UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ
CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

PEDRO ANDERSON COSTA MARTINS

UM MAPEAMENTO SISTEMÁTICO SOBRE FERRAMENTAS PARA DETECÇÃO DE *FLAKY TESTS*

QUIXADÁ
2023

PEDRO ANDERSON COSTA MARTINS

UM MAPEAMENTO SISTEMÁTICO SOBRE FERRAMENTAS PARA DETECÇÃO DE *FLAKY TESTS*

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

Orientadora: Profa. Dra. Carla Ilane Moreira Bezerra

QUIXADÁ

2023

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

M345m Martins, Pedro Anderson Costa.
Um mapeamento sistemático sobre ferramentas para detecção de flaky tests / Pedro Anderson Costa Martins. – 2023.
65 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2023.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Flaky tests. 2. Ferramentas - Detecção. 3. Teste de software – Automação. I. Título.
CDD 005.1

PEDRO ANDERSON COSTA MARTINS

UM MAPEAMENTO SISTEMÁTICO SOBRE FERRAMENTAS PARA DETECÇÃO DE *FLAKY TESTS*

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

Aprovada em: ___/___/___

BANCA EXAMINADORA

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

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

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

Dedico este trabalho aos meus pais, irmão, tia, noiva e avós, que estiveram ao meu lado e me deram forças em todos os momentos dessa jornada. Especialmente a minha mãe e meu pai, que trabalharam incansavelmente durante toda a minha vida para me proporcionar esse momento, obrigado por tudo!

AGRADECIMENTOS

Primeiramente agradeço a minha família. Minha mãe, meu pai, meu irmão e minha tia que me incentivaram e me apoiaram desde o início dessa jornada e me deram forças durante todo o período, sendo meu porto seguro em todos os momentos difíceis e me ajudando a sempre se manter firme. A minha noiva, que esteve presente no dia a dia dessa caminhada, nos momentos felizes ou tristes, sempre me apoiando em cada adversidade. Ao meu avô e minha avó, que cuidaram tanto de mim e me ensinaram sobre a importância dos estudos, mas que não puderam estar ao meu lado no dia da minha conquista.

À Profª, Drª Carla Ilane Moreira Bezerra, que me ensinou diversas lições ao longo das disciplinas e atividades acadêmicas, contribuindo diretamente com a minha visão sobre os estudos e dedicação pessoal, adquirindo a mentalidade de sempre fazer o melhor.

"O sucesso é a soma de pequenos esforços repetidos dia após dia."

(Robert Collier)

RESUMO

Flaky tests são testes que se comportam de forma não determinística, ou seja, possuem resultados incertos, podendo retornar resultados de aprovação e falha mesmo quando forem executados em um mesmo código de testes que não sofreu alteração. Uma parte considerável das falhas em suítes de testes são ocasionadas por *flaky tests*. Se considerar que o modelo de integração contínua é amplamente adotado em projetos industriais, com milhares de testes, a presença de *flaky tests* podem ser mais críticas ainda, visto que durante os testes de regressão, um único *flaky test* pode paralisar todo o processo de criação de builds e lançamentos de atualizações, ocasionando atraso nas entregas. Neste estudo, foi executado um mapeamento sistemático para investigar as ferramentas relacionadas a *Flaky tests* em 37 artigos, com o objetivo de responder a 3 questões de pesquisa. A primeira questão de pesquisa, busca mostrar as ferramentas encontradas e disponíveis para a comunidade, onde foram retornadas 30 ferramentas e que foram organizadas de acordo com suas datas de publicação ou criação. A segunda questão de pesquisa tem o intuito de fornecer uma visão geral de cada uma das 30 ferramentas coletadas, englobando os objetivos, técnicas ou abordagens e o estudo da qual cada ferramenta foi coletada, onde foi possível observar, por exemplo, que a reexecução é a técnica mais utilizada como apoio à detecção. A terceira e última questão de pesquisa tem como intuito fornecer as características de mais alto nível das ferramentas e as causas que elas abordam, mostrando, por exemplo, que um total 46% das ferramentas abordam a causa de dependência da ordem de teste e que 70% das ferramentas são analisadas na linguagem java. Com as descobertas deste estudo, espera-se contribuir com atuantes da área de Testes de Software, pesquisadores e desenvolvedores, fornecendo insumos para estudos futuros e uma fonte mais centralizada das ferramentas disponíveis, para que os interessados possam selecionar uma ou mais ferramentas para estudos futuros ou alguma necessidade profissional.

Palavras-chave: *Flaky Tests*. Ferramentas de Detecção. Testes de Software Automatizados.

ABSTRACT

Flaky tests are tests that behave in a non-deterministic way, that is, they have uncertain results, and may return pass and fail results even when executed in the same code of tests that have not changed. A considerable part of test suite failures are caused by flaky tests. If we consider that the continuous integration model is widely adopted in industrial projects, with thousands of tests, the presence of flaky tests can be even more critical, since during regression tests, a single flaky test can paralyze the entire creation process of builds and releases of updates, causing delays in deliveries. In this study, a systematic mapping was carried out to investigate the tools related to Flaky tests in 37 articles, with the objective of answering 3 research questions. The first research question seeks to show the tools found and available to the community, where 30 tools were returned and which were organized according to their publication or creation dates. The second research question is intended to provide an overview of each of the 30 tools collected, encompassing the objectives, techniques or approaches and the study from which each tool was collected, where it was possible to observe, for example, rerun is the most used technique to support detection. The third and final research question is intended to provide the highest level features of the tools and the causes they address, showing, for example, that a total of 46% of the tools address the test order dependency cause and that 70% of the tools are analyzed in the java language. With the discoveries of this study, it is expected to contribute to players in the Software Testing area, researchers and developers, providing inputs for future studies and a more centralized source of available tools, so that those interested can select one or more tools for future studies or some professional necessity.

Keywords: Flaky Tests. Detection Tools. Software Testing (Software Engineering). Automated Software Testing.

LISTA DE FIGURAS

Figura 1 – Pirâmide de testes 19
Figura 2 – Resultados de uma suíte de testes 23
Figura 3 – Processo de filtragem de publicações 35
Figura 4 – Diagrama de fluxo dos procedimentos metodológicos 37
Figura 5 – Atividades da etapa de planejamento 38
Figura 6 – String de pesquisa 39
Figura 7 – Estudo por bibliotecas digitais 40
Figura 8 – Atividades da etapa de execução 41
Figura 9 – Filtragem de artigos após etapa de execução 42

LISTA DE TABELAS

Tabela 1 – Causas gerais 24
Tabela 2 – Ferramentas para detecção de *flaky tests* 27
Tabela 3 – Técnicas de correção de *flaky tests* para as principais categorias 28
Tabela 4 – Catálogo das ferramentas de detecção de *flaky tests* e suas características . . 56
Tabela 5 – Conjunto final dos estudos selecionados 65

LISTA DE QUADROS

Quadro 1 – Comparativo entre os trabalhos relacionados e o trabalho proposto 33

Quadro 2 – Bibliotecas digitais utilizadas na pesquisa 39

Quadro 3 – Critérios de inclusão e exclusão 40

Quadro 4 – Filtragem por biblioteca digital 42

Quadro 5 – Publicação das ferramentas de detecção de *flaky tests* para a comunidade . 43

Quadro 6 – Ferramentas de detecção de *flaky tests* 45

Quadro 7 – Ferramentas de detecção de *flaky tests* em testes dependentes de ordem . . 47

Quadro 8 – Ferramentas de mitigação de *flaky tests* 50

Quadro 9 – Ferramentas de reparo de *flaky tests* 51

Quadro 10 – Causas de *flaky tests* abordadas pelas ferramentas 55

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Classe de testes de uma Biblioteca 21
Código-fonte 2 – Ordernação de testes com xUnit 23

LISTA DE ABREVIATURAS E SIGLAS

CI	*Continuous integration*
API	Interface de programação de aplicações
JSON	*JavaScript Object Notation*
IDE	*Integrated Development Environment*
CUT	*Code under test*
Snowballing	*Snowball sampling*
ID	Identificador
GPS	*Global Positioning System*
Detec. OD	Testes Dependentes de Ordem
CLI	*Command-line Interface*
Doc	Documentação
UNK	*Unknown*

SUMÁRIO

1 INTRODUÇÃO . . . 16

1.1 Objetivos . . . 18

1.2 Organização . . . 18

2 FUNDAMENTAÇÃO TEÓRICA . . . 19

2.1 Testes de software automatizados . . . 19

2.2 *Flaky tests* . . . 20

2.2.1 Causas gerais . . . 24

2.2.1.1 Principais causas . . . *24*

2.2.2 Estratégias de detecção . . . 25

2.2.2.1 Detecção nas principais categorias . . . *26*

2.2.2.2 Ferramentas de detecção . . . *26*

2.2.3 Estratégias de mitigação . . . 27

2.2.4 Técnicas de correção . . . 28

3 TRABALHOS RELACIONADOS . . . 29

3.1 *Test Smell Detection Tools: A Systematic Mapping Studys* . . . 29

3.2 *A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests* . . . 30

3.3 *Root Causing, Detecting, and Fixing Flaky Tests: State of The Art and Future Roadmap* . . . 31

3.4 *A Survey of Flaky Tests* . . . 31

3.5 Análise comparativa . . . 32

4 PROCEDIMENTOS METODOLÓGICOS . . . 34

4.1 Planejamento . . . 34

4.2 Execução . . . 34

4.3 Análise . . . 35

5 MAPEAMENTO SISTEMÁTICO DE FERRAMENTAS PARA DETECÇÃO DE *FLAKY TESTS* . . . 37

5.1 Introdução . . . 37

5.2 Planejamento . . . 38

5.3 Execução . . . 41

5.4 Análise . . . 43
5.4.1 QP1 - Quais ferramentas de detecção de flaky tests estão disponíveis para a comunidade? . . . 43
5.4.2 QP2 - Quais os objetivos das ferramentas de detecção de flaky tests e quais técnicas elas utilizam? . . . 44
5.4.2.1 Ferramentas e técnicas para detecção de flaky tests . . . *44*
5.4.2.2 Ferramentas e técnicas para detecção de flaky tests em testes dependentes de ordem . . . *46*
5.4.2.3 Ferramentas e técnicas para mitigação de flaky tests . . . *50*
5.4.2.4 Ferramentas e técnicas para reparo de flaky tests . . . *51*
5.4.3 QP3 - Quais são as principais características das ferramentas de detecção de flaky tests e quais causas elas abordam? . . . 53
5.5 Discussão . . . 57
6 AMEAÇAS À VALIDADE . . . 58
7 CONCLUSÕES E TRABALHOS FUTUROS . . . 59
REFERÊNCIAS . . . 60
APÊNDICE A–CONJUNTO FINAL DOS ARTIGOS SELECIONADOS NO MAPEAMENTO . . . 65

1 INTRODUÇÃO

O processo de teste de software constitui uma importante etapa do desenvolvimento de um software e acompanhando sua constante evolução, a implementação de testes automatizados se tornou algo bastante útil para obtenção de *feedbacks* mais ágeis para os desenvolvedores, considerando que os ciclos de desenvolvimentos estão mais curtos (MYERS *et al.*, 2011). Porém, esses *feedbacks* podem ter sua confiabilidade questionada à medida que seu histórico de entregas são testes com resultados ambíguos e incertos (PARRY *et al.*, 2021). Esses testes com resultados imprecisos, conhecidos como *flaky tests*, são testes que podem ser aprovados e reprovados quando executados em uma mesma versão do software, ou seja, sem alterações no código (PINTO *et al.*, 2020; PARRY *et al.*, 2021).

Alguns estudos fornecem dados que ampliam a percepção dos problemas que os *flaky tests* podem ocasionar (ECK *et al.*, 2019), apresentando que falhas em suítes de testes causados por *flaky tests* podem ocorrer com frequência em softwares, causando problemas como atraso na entrega de um produto ou diminuindo a confiabilidade dos testes automatizados (BELL *et al.*, 2018). Por exemplo, no estudo de Luo *et al.* (2014), os autores abordaram um escopo de 51 projetos *open-source* analisando de 201 *commits*, eles relataram que os *flaky tests* foram responsáveis por 73 mil de 1,6 milhão (4,6%) dos casos de falha de testes do sistema Google TAP, um valor que apesar de parecer pequeno, pode causar grandes problemas, visto que apenas um único *flaky test* pode chegar a paralisar todo um *deploy*. Em outro trabalho, Labuschagne *et al.* (2017) estudaram 61 projetos que utilizam uma ferramenta de integração contínua hospedada em nuvem, o *Travis CI*[1], onde descobriram que 13% das suítes de testes que tiveram falha são causadas por *flaky tests* (BELL *et al.*, 2018).

O estudo das causas e estratégias de detecção de *flaky tests* são um grande desafio, considerando a natureza não determinística do mesmo e vem se tornando uma área de pesquisa mais ativa (LAM *et al.*, 2020). De acordo com Luo *et al.* (2014), as principais causas de *flaky tests* são (1) espera assíncrona, (2) simultaneidade e (3) testes com dependência de ordem (LAM *et al.*, 2020). Além de outras causas que também foram inseridas com o tempo por (ECK *et al.*, 2019), como (1) tempo Limite do caso de teste e (2) dependência de plataforma, por exemplo. Os *flaky tests* causados por dependência de ordem, são frequentemente agrupados em uma categoria à parte dos que não são causados por dependência de ordem (esta que se refere a todas outras causas), onde podemos considerar então 2 categorias principais, as que são causadas por

[1] https://www.travis-ci.com/

dependência de ordem e as que não são (GRUBER *et al.*, 2021). Entre as estratégias de detecção, a mais popular é a reexecução dos testes, onde é possível verificar se o seu resultado é alterado a cada execução. Porém, por ser uma técnica que pode consumir muito tempo de acordo com a quantidade de testes, outras propostas podem ser abordadas, como identificar *flaky tests* de acordo com a diferença de cobertura entre versões consecutivas de um software, registrando se o resultados dos testes muda quando nenhum código é modificado (PARRY *et al.*, 2021; BELL *et al.*, 2018).

Ao assimilar as principais causas e estratégias de detecção, é possível direcionar os esforços para identificar e selecionar ferramentas que possam atuar de fato no processo de detecção de *flaky tests*, de maneira eficiente e automatizada (PARRY *et al.*, 2021; GRUBER *et al.*, 2021). Algumas ferramentas que atuam no processo de reexecução de testes ou no processo de identificação de *flaky tests* em testes dependentes de ordem, por exemplo, foram listadas por Parry *et al.* (2021) e Gruber *et al.* (2021). Considerando que a indústria e comunidade adotaram o modelo de *Continuous integration* (CI) de desenvolvimento e lançamento de software, consequentemente os testes de regressão, que executam uma suíte de testes para verificar se as alterações recentes ao software impactaram em alguma funcionalidade existente, são executados automaticamente por meio de um *pipeline* automatizado (LAM *et al.*, 2019; LABUSCHAGNE *et al.*, 2017). Diante disso, também é necessário identificar e selecionar produtos e serviços de CI que possuam ferramentas que possibilitem a detecção de *flaky tests*, visto que a manifestação de falsas falhas podem podem impedir criação de *builds* e lançamentos de atualizações, consequentemente ocasionando atrasos nas entregas (LAM *et al.*, 2020; LABUSCHAGNE *et al.*, 2017).

Portanto, este trabalho visa identificar e analisar um grupo de ferramentas que trabalhem na detecção, mitigação e reparo de *flaky tests*, onde as mesmas serão apresentadas com um conjunto de características em alto nível. Para isso, será realizado um mapeamento sistemático, adaptado dos processos realizados por Petersen *et al.* (2015) e Keele *et al.* (2007), para buscar uma maior compreensão sobre as ferramentas de detecção, mitigação e reparo de *flaky tests* e responder às questões de pesquisa abordadas durante o estudo. A primeira questão de pesquisa, busca mostrar as ferramentas encontradas e disponíveis para a comunidade, onde foram retornadas 30 ferramentas e que foram organizadas de acordo com suas datas de publicação ou criação. A segunda questão de pesquisa tem o intuito de fornecer uma visão geral de cada uma das 30 ferramentas coletadas, englobando os objetivos, técnicas ou abordagens e o estudo da

qual cada ferramenta foi coletada, onde foi possível observar, por exemplo, que a técnica de reexecução é a mais utilizada como apoio à detecção. A terceira e última questão de pesquisa tem como intuito fornecer as características de mais alto nível das ferramentas e as causas que elas abordam, mostrando, por exemplo, que um total 46% das ferramentas abordam a causa de dependência da ordem de teste e que 70% das ferramentas são analisadas na linguagem java.

Assim, espera-se contribuir com atuantes da área de Testes de Software, pesquisadores e desenvolvedores que não possuam conhecimento ou consenso entre as ferramentas existentes e principais maneiras para detecção, mitigação e reparo de *flaky tests*, além de fornecer insumos para estudos futuros.

1.1 Objetivos

O objetivo principal deste trabalho é identificar ferramentas adotadas na detecção, mitigação e reparo de *flaky tests* e fornecer dados qualitativos sobre as soluções encontradas a fim de gerar um catálogo das ferramentas.

Como objetivos específicos estão:

a) Realizar um mapeamento sistemático para coletar ferramentas de detecção, mitigação e reparo de *flaky tests* existentes.
b) Fornecer dados qualitativos das ferramentas selecionadas.
c) Elaborar um catálogo com ferramentas para detecção, mitigação e reparo de *flaky tests*.

1.2 Organização

O restante deste trabalho está organizado da seguinte maneira: No capítulo 2 são apresentados os três principais conceitos para o desenvolvimento deste trabalho. No capítulo 3 realiza-se a apresentação e discussão dos trabalhos relacionados, com suas semelhanças e diferenças ao trabalho aqui proposto. No capítulo 4 a metodologia a ser empregada para o desenvolvimento deste trabalho é explicada. No capítulo 5 é demonstrado o processo de mapeamento sistemático que foi realizado. O Capítulo 6 mostra as ameaças à validade deste trabalho e no capítulo 7 é apresentada a conclusão do trabalho e alguns trabalhos futuros que podem ser beneficiados por esta pesquisa.

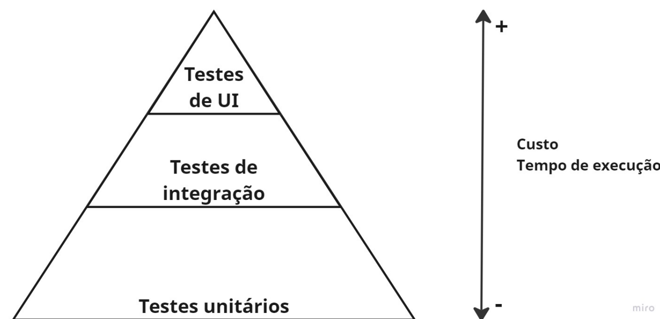
2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, serão apresentados alguns dos conceitos mais pertinentes para o entendimento e desenvolvimento deste trabalho. Na Seção 2.1 são apresentados os conceitos básicos de automação de testes de software. Na Seção 2.2, o conceito de *flaky tests* será aprofundado, assim como suas principais causas, estratégias de detecção e mitigação, técnicas de correção e impacto no processo de desenvolvimento de um software.

2.1 Testes de software automatizados

Os testes automatizados são *scripts* codificados com a utilização de *frameworks* e são executados cada vez que o software em desenvolvimento é testado, onde muitas vezes ocorre durante os testes de regressão no processo de CI (SOMMERVILLE, 2011). Diferente dos testes manuais, onde é necessário que um testador execute o software com alguns dados de teste e então compare os resultados com suas expectativas, os testes automatizados não necessitam de uma interferência humana (SOMMERVILLE, 2011). De acordo com Cohn (2010), para que uma estratégia de automação seja eficaz, é necessário que a automação de testes englobe 3 níveis diferentes: Interface ou Sistema, Serviços ou Integração e Unidades, conforme mostrado na Figura 1.

Figura 1 – Pirâmide de testes



Fonte: Elaborado pelo autor

Na camada mais abaixo da Pirâmide de Testes, estão os testes unitários, onde a funcionalidade dos métodos das classes de objetos são testadas individualmente, ou seja, são testadas as menores unidades do sistema (SOMMERVILLE, 2011). Justamente por abranger as menores partes do sistema, o escopo dos testes é menor, ocasionando em uma implementação mais rápida e consequentemente diminuindo seu custo. Na maioria das vezes, os testes unitários são escritos na mesma linguagem do sistema, onde os desenvolvedores possuem mais

familiaridade, amplificando as característica citadas por Cohn (2010). É importante ressaltar que não apenas os cenários de sucesso devem ser testados, mas também cenários alternativos, onde usam-se entradas anormais para verificar se elas também são processadas e tratadas da maneira correta (SOMMERVILLE, 2011). Esse processo deve ser realizado não apenas em testes unitários, mas também em testes de integração e de interface.

Na camada intermediária, situam-se os Testes de Integração ou Testes de Serviço, onde os módulos individuais são integrados a fim de serem testados como componentes, sendo possível validar se funcionam corretamente em conjunto (SOMMERVILLE, 2011). Esses testes possuem uma maior complexidade de implementação, visto que a criação do código envolve justamente a configuração de todos os componentes que serão percorridos pelo teste, consequentemente aumentando o escopo e custo do mesmo. Um exemplo nesse contexto de integração são os testes de Interface de programação de aplicações (API), que é um trecho de código que permite que dois componentes de software interajam, onde os testes validam diferentes cenários e garantem que o conteúdo da resposta do *JavaScript Object Notation* (JSON) esteja correto (ISHA *et al.*, 2018). A partir do momento que esses testes dependem da comunicação de várias camadas, como a API com o banco de dados, por exemplo, maior a chance de manifestação de *flaky tests*, visto que qualquer falha momentânea em uma das camadas pode impactar todo o teste e gerar falsos negativos.

Na camada mais acima, temos os Testes de Interface ou Testes de Sistema, onde as interações entre os componentes são testadas após integradas, logo, o sistema é testado como um todo (SOMMERVILLE, 2011). Visto que esses testes possuem mais camadas integradas, maior ainda o escopo e custo de implementação. Os Testes de Sistema envolvem também a interface do usuário, e por se tratar de uma camada que pode sofrer alterações com mais recorrência, diversos testes podem quebrar em um curto intervalo de tempo (COHN, 2010). Ao percorrer todo o sistema durante a execução de um cenário de teste, o processo pode se tornar bastante demorado, principalmente considerando a quantidade de cenários que será executada (COHN, 2010). Por ter mais camadas ainda envolvidas no processo e considerando os fatores citados, é bastante comum que os Testes de Interface estejam em menor quantidade em um projeto.

2.2 *Flaky tests*

Flaky Tests são testes que se comportam de forma não determinística, portanto, podem gerar resultados de aprovação e falha quando executados repetidamente no mesmo código

em teste (GRUBER; FRASER, 2022). Esses testes já são identificados com frequência na indústria, como apontado por Eck *et al.* (2019), que em sua pesquisa indicaram que 20% dos entrevistados afirmaram encontrar *flaky tests* mensalmente, 24% encontram semanalmente e 15% diariamente. Em termos de severidade, dos 91% dos desenvolvedores que lidam com *flaky tests* pelo menos alguma vez no ano, 56% o consideram um problema moderado e 23% consideram um problema sério (PARRY *et al.*, 2021; ECK *et al.*, 2019).

Para exemplificar melhor o que são *flaky tests* na prática, temos no trecho de Código-fonte 1 um exemplo de Teste de API desenvolvido na linguagem CSharp com auxílio do *framework* xUnit[1]. O código consiste na classe *LibraryServiceWorkFlow* com 2 métodos de testes, representando os testes de um sistema fictício de uma Biblioteca e que possui 2 casos de testes. O primeiro caso de teste, denominado *ValidatePostBookResponse*, representa uma requisição do tipo *POST*, onde o objetivo é realizar uma requisição para adicionar um objeto Livro, criado nas linhas 7 a 10, no banco de dados através de um endpoint, na linha 13, e verificar se a resposta do endpoint corresponde ao objeto adicionado por meio do método *Assert.Equal*. O segundo caso de teste, denominado *ValidateGetBookResponse*, representa uma requisição do tipo *GET*, onde o objetivo é buscar uma lista de livros do banco de dados, na linha 30, e então verificar se o Livro adicionado anteriormente consta na lista por meio do método *Assert.Contains*. Ambos métodos de teste, nas linhas 3 e 20, são marcados com o atributo *Fact* para que sejam reconhecidos como métodos de testes. Os atributos podem conter propriedades como o *DisplayName*, por exemplo, para definir qual nome do método irá aparecer na interface do *Integrated Development Environment* (IDE).

Código-fonte 1 – Classe de testes de uma Biblioteca

```
1  public class LibraryServiceWorkFlow {
2
3      [Fact(DisplayName = "Validate Post Book")]
4      public async void Validate_PostBook_Response()
5      {
6          //Arrange
7          Book postBook = new();
8
9          postBook.Title = "The lord of the rings";
10         postBook.Author = "Tolkien";
11
12         //Act
13         Book returnBook = await new BookAPI().Post_Book(postBook);
```

[1] https://xunit.net

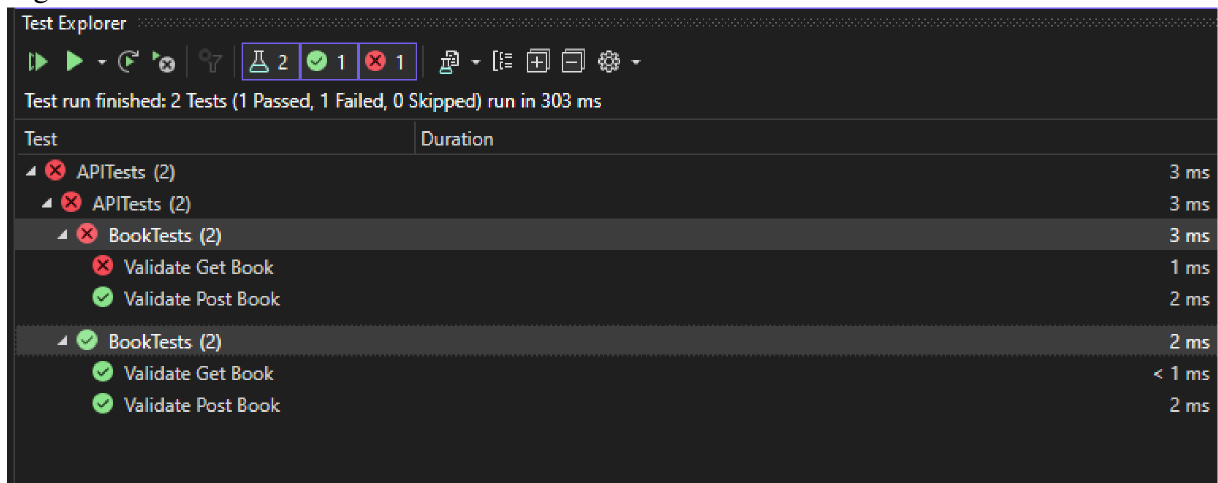
```
14
15         //Assert
16         Assert.Equal(postBook.Title, returnBook.Title);
17         Assert.Equal(postBook.Author, returnBook.Author);
18     }
19
20     [Fact(DisplayName = "Validate Get Book")]
21     public async void Validate_GetBook_Response()
22     {
23         //Arrange
24         Book requestedBook = new();
25
26         requestedBook.Title = "The lord of the rings";
27         requestedBook.Author = "Tolkien";
28
29         //Act
30         List<Book> response_allBooks = await new BookAPI().Get_Books();
31
32         //Assert
33         Assert.Contains(response_allBooks, book =>
34                 book.Title == requestedBook.Title &&
35                 book.Author == requestedBook.Author
36             );
37     }
38 }
```

Fonte: Elaborado pelo autor

Ao executar a suíte de testes demonstrada no código, é possível obter 2 resultados, um de falha e outro de sucesso, conforme mostrado na Figura 2. Se o caso de teste representado pela requisição *POST* for executado primeiro, o objeto Livro será adicionado no banco de dados e quando o caso de teste representado pela requisição *GET* for executado, encontrará o objeto na lista de Livros, dessa forma o resultado da suíte será de sucesso. Mas caso os papéis se invertam, a requisição *GET* pode ocorrer antes da requisição *POST*, então o resultado da suíte será de falha, visto que o caso de teste representado pela requisição *GET* buscará um objeto que ainda não foi adicionado ao banco.

A suíte de testes demonstrada no código é dependente da ordem de execução dos casos de testes, visto que a requisição *GET* só será bem-sucedida quando executada após a requisição *POST*. Para resolver esse tipo de problema, é necessário usar artifícios do *framework* para executar os testes em uma ordem definida. No exemplo citado, seria necessário implementar um código de ordenação de métodos de testes e adicionar a propriedade *TestPriority* em cada método de teste, como pode ser visto nas linhas 3 e 6 do Código-fonte 2. Dessa forma, a

Figura 2: Resultados de uma suíte de testes



Fonte: Elaborado pelo autor

requisição *POST* teria prioridade 1 e a requisição *GET* teria prioridade 2, sendo executadas na ordem de prioridade definida e retornando um resultado de sucesso para essa suíte de testes.

Código-fonte 2: Ordernação de testes com xUnit

```
1 public class LibraryServiceWorkFlow {
2
3     [Fact, TestPriority(1)]
4     public async void Validate_PostBook_Response(){}
5
6     [Fact, TestPriority(2)]
7     public async void Validate_GetBook_Response(){}
8 }
```

Fonte: Elaborado pelo autor

Para compreender melhor os *flaky tests* e a motivação deste trabalho, é necessário se aprofundar um pouco mais nas questões que compõem esse assunto, como as causas, estratégias de detecção e também as ferramentas já desenvolvidas para estes fins e técnicas de mitigação e correção que podem ser utilizadas para reduzir ou evitar os danos. Além disso, essas questões podem ser de grande serventia para aqueles que têm em vista entender mais sobre os *flaky tests*, como desenvolvedores, analistas de testes, pesquisadores e atuantes da área.

2.2.1 *Causas gerais*

De acordo com a pesquisa de (LUO *et al.*, 2014), um dos primeiros estudos empíricos realizados sobre *flaky tests*, foi realizada uma inspeção em 201 *commits* em códigos *open-source* que trabalharam na correção de *flaky tests*. Logo, algumas causas gerais de *flaky tests* foram identificadas e classificadas, gerando então 10 categorias de causas que podem ser visualizadas na Tabela 1. Esse trabalho visa coletar ferramentas que atuem na detecção de *flaky tests* e informar quais causas elas podem detectar.

É também possível visualizar que Espera Assíncrona, Simultaneidade e Dependência da Ordem de Teste foram identificadas em 125 dos 201 *commits* inspecionados, representando aproximadamente 62% dos commits. Portanto, são consideradas como as principais causas de *flaky tests*.

Tabela 1: Causas gerais

***Commits* inspecionados**	**201**
Espera Assíncrona	74
Simultaneidade	32
Dependência da Ordem de Teste	19
Vazamento de recursos	11
Rede	10
Tempo	5
E/S	4
Aleatoriedade	4
Operações com Ponto Flutuante	3
Coleções Não Ordenadas	1

Fonte: Adaptado de Luo *et al.* (2014)

2.2.1.1 *Principais causas*

A categoria de Espera Assíncrona (*Async Wait*) é caracterizada por um teste, que em execução, realiza uma requisição assíncrona e não espera adequadamente que o seu resultado seja retornado (ECK *et al.*, 2019). Em um teste de API, por exemplo, quando o teste realiza uma requisição para algum serviço, onde o seu resultado é necessário para validações futuras, caso o mesmo não seja retornando em um tempo hábil, o teste provavelmente irá falhar.

A categoria de Simultaneidade (*Concurrency*) é caracterizada por um teste onde diferentes *threads* interagem de uma maneira imprevista ou não desejável, onde o problema pode ocorrer devido a Condições de Corrida, por exemplo (PARRY *et al.*, 2021; LUO *et al.*, 2014). Essa condição ocorre quando o resultado de um processo depende de outros recursos e existe a tentativa de manipulação de dados antes desse resultado. Diferente da categoria de Espera

Assíncrona (*Async Wait*), esta categoria se refere principalmente a problemas de sincronização local (ECK *et al.*, 2019).

Visto que a Espera Assíncrona (*Async Wait*) e a Simultaneidade (*Concurrency*) demandam a utilização de vários *threads* simultaneamente, a ordem de execução pode não seguir a ordem pretendida e o resultado pode ser impreciso (ZOLFAGHARI *et al.*, 2021).

A categoria de Dependência da Ordem de Teste (*Test Order Dependency*) é caracterizada por testes onde seus resultados dependem da ordem em que os testes são executados, como o seu próprio nome sugere (LUO *et al.*, 2014). Esse problema geralmente surge quando os testes partilham de um mesmo conjunto de dados ou de um estado (LUO *et al.*, 2014). Por exemplo, se uma suíte de testes tem como objetivo testar as requisições de uma API, caso um objeto precisasse ser criado primeiro, para servir de dado para outros testes posteriores, se o mesmo fosse executado pelo *framework* após os outros testes, todos que foram executados anteriormente seriam falhos, gerando dados inconsistentes. Em sua pesquisa, Parry *et al.* (2021) consideram também fatores específicos associados aos testes dependentes da ordem em particular, visto que várias fontes consideram que a Dependência da Ordem de Teste possui fatores e dependências que podem estar ligadas diretamente aos *flaky tests* (PARRY *et al.*, 2021).

2.2.2 Estratégias de detecção

Detectar a origem de uma falha é o primeiro passo para corrigí-la. A detecção automática de falhas e que potencialmente podem ser *flaky tests*, podem ser bastante úteis para desenvolvedores e analistas de testes, destacando testes que podem precisar ser refatorados, permitindo o uso de estratégias de mitigação e consequentemente evitando tempo desperdiçado (PARRY *et al.*, 2021).

A maneira mais tradicional para detecção de *flaky tests* é a reexecução do teste com falha para verificar se o mesmo passará ou não, na tentativa de identificar inconsistências em seus resultados (ZOLFAGHARI *et al.*, 2021; PARRY *et al.*, 2021). Porém, apesar de ser uma técnica válida e útil em alguns casos, a probabilidade de um *flaky test* ter seu resultado alterado não é tão alta, além de que a reexecução pode ser bastante demorada dependendo da quantidade de vezes aplicada e tamanho da suíte de testes automatizados (LUO *et al.*, 2014).

No estudo de Zolfaghari *et al.* (2021), foi observado que alguns métodos para cobertura de código são utilizadas para detecção de *flaky tests*, visto que um código com alta cobertura possui mais chances de ter uma falha detectada, pois aumenta a quantidade de linhas

de código cobertas. Inclusive, algumas pesquisas tentam melhorar a cobertura de código para tentar diminuir as reexecuções citadas anteriormente (ZOLFAGHARI *et al.*, 2021).

2.2.2.1 Detecção nas principais categorias

Além das técnicas já citadas, para cada uma das três principais categorias de *flaky tests* serão levantadas algumas maneiras de manipular os testes para que seja possível ocorrer a manifestação de *flaky tests* e assim identificá-los e planejar técnicas de mitigação (LUO *et al.*, 2014).

Dos testes classificados na categoria de Espera Assíncrona (*Async Wait*), Luo *et al.* (2014) descobriram que 34% utilizaram um atraso simples ou algum método de espera nos testes, ou seja, os *flaky tests* podem ser detectados diminuindo esse tempo de atraso. Além disso, também alegaram que os *flaky tests* podem ser detectados ao adicionar um tempo de atraso em alguma parte do código sem a necessidade de depender de recursos externos.

De acordo com Luo *et al.* (2014), a manifestação de *flaky tests* na categoria de Simultaneidade (*Concurrency*) está atrelada a quantidade, onde descobriram que 13 testes envolvem mais de 2 threads. Além disso, 97% dos testes não dependiam de recursos externos, como o sistema de arquivos, logo, as falhas se manifestam devido à acessos simultâneos aos objetos na memória principal.

Na categoria de Dependência da Ordem de Teste, os *flaky tests* podem se manifestar de algumas maneiras, como quando vários testes acessam o mesmo campo estático que foi declarado no código de teste ou quando o campo estático que é compartilhado entre casos de testes seja declarado no *Code under test* (CUT) ao invés de no próprio código de teste, isto é, um teste que dependia de um estado de outro teste foi iniciado sem receber as informações necessárias ou não se iniciou com um estado limpo (LUO *et al.*, 2014). Porém, a maioria dos *flaky tests* dessa categoria se manifesta por dependência de recursos externos, que exigem técnicas mais aprimoradas para traçar o estado do ambiente ou para realizar a reexecução dos testes com ordem diferente (LUO *et al.*, 2014).

2.2.2.2 Ferramentas de detecção

Em seu trabalho, Lam *et al.* (2020) realizaram uma investigação para encontrar a melhor maneira de aplicar ferramentas de detecção de *flaky tests*. Ferramentas como iDFlakies e NonDex foram executadas nos commits coletados por eles para detectar *flaky tests*, onde para

cada *flaky test* identificado, eles executavam as ferramentas no commit inicial que o introduziu no conjunto de testes. Caso o teste não fosse de fato um *flaky test*, então o primeiro commit onde o *flaky test* era identificado era procurado até descobrir o commit que introduziu o problema (PARRY *et al.*, 2021; LAM *et al.*, 2020). Dessa forma, isso os levou a sugerir que a execução de detectores de *flaky tests* acontecesse logo após a introdução dos testes.

Além das ferramentas citadas anteriormente, outras ferramentas também foram citadas em trabalhos de Lam *et al.* (2020), entre outros. Algumas delas foram listadas na pesquisa de Parry *et al.* (2021), servindo de insumo para esse trabalho. A lista contendo algumas ferramentas para detecção de *flaky tests* pode ser visualizada na Tabela 2.

Tabela 2: Ferramentas para detecção de *flaky tests*

Ferramenta	**Descrição**	**Fonte**
DTDetector	Detecta testes dependentes de ordem revertendo a ordem do conjunto de testes, embaralhando-a, executando cada k-permutação isoladamente ou apenas executando permutações que provavelmente irão expor uma dependência de ordem de teste, com base na análise conservadora de acesso de campo estático entre testes e monitoramento do uso de arquivos.	Zhang *et al.* (2014a)
OrcalePolish	Pode identificar indiretamente testes dependentes de ordem ou testes que têm o potencial de se tornarem dependentes da ordem.	Huo e Clause (2014)
Shaker	Introduz estresse de CPU e memória durante execuções repetidas do conjunto de testes em uma tentativa de aumentar a probabilidade de manifestar *flaky tests* nas categorias de espera assíncronas e simultaneidade.	Silva *et al.* (2020)
FlakeFlagger	Uma técnica para detectar *flaky tests* sem exigir reexecuções do conjunto de testes, usando uma máquina modelo de aprendizagem. Requer uma combinação de dados de teste dinâmicos, como cobertura de linha e dados estáticos, como recursos do código-fonte do teste.	Alshammari *et al.* (2021)

Fonte: Adaptado de Parry *et al.* (2021)

2.2.3 Estratégias de mitigação

Os *flaky tests*, bem como os bugs, também são considerados problemas que não podem ser totalmente eliminados de um software, dessa forma, muitos estudos apresentaram e avaliaram algumas estratégias e técnicas de mitigação que ajudam no controle desse problema (PARRY *et al.*, 2021).

De acordo com Gruber e Fraser (2022), alguns exemplos de abordagens foram propostas para realização da mitigação de *flaky tests*, como: reexecutar testes que já teve seus problemas citados anteriormente, desabilitar testes, reexecutar automaticamente testes por meio de notações, utilizar ferramentas para detecção automática, entre outras maneiras. Esse trabalho também visa analisar como as ferramentas e técnicas de detecção podem contribuir na mitigação de *flaky tests*.

2.2.4 *Técnicas de correção*

Apesar de não ser possível eliminar totalmente a presença de *flaky tests*, desenvolvedores utilizam algumas técnicas para corrigir suas várias causas em diferentes cenários e de acordo com a situação. Em seu trabalho, Luo *et al.* (2014) listaram algumas das técnicas de correção mais utilizadas para as principais categorias citadas anteriormente: Espera Assíncrona (*Async Wait*), Simultaneidade (*Concurrency*) e Dependência da Ordem de Teste (*Test Order Dependency*).

É possível visualizar algumas dessas técnicas na Tabela 3. Este trabalho pretende verificar quais técnicas estão presente nas ferramentas e quais podem ser utilizadas em benefício da reparação de *flaky tests*.

Tabela 3: Técnicas de correção de *flaky tests* para as principais categorias

Categorias	**Tipo de Correção**
Espera Assíncrona	Adicionar/modificar *waitFor* Adicionar/modificar *sleep* Reordenar execução
Simultaneidade	Bloquear operação atômica Tornar determinístico Alterar condição Alterar asserção
Dependência da Ordem de Teste	Configurar/alterar estado Remover dependência Mesclar testes

Fonte: Adaptado de Luo *et al.* (2014)

3 TRABALHOS RELACIONADOS

Neste capítulo, serão apresentados quatro trabalhos relacionados ao projeto proposto neste trabalho. A seção 3.1 discorre sobre o trabalho de Aljedaani *et al.* (2021), que tem como objetivo principal fornecer uma visão abrangente das ferramentas de detecção de *tests smells*. A seção 3.2 apresenta o trabalho de Habchi *et al.* (2022), que tem como objetivo fornecer dados qualitativos sobre as fontes, impactos e estratégias de mitigação de *flaky tests*. A seção 3.3 aborda o trabalho de Zolfaghari *et al.* (2021), onde foi realizada uma revisão sistemática sobre os feitos mais recentes em relação à detecção e mitigação de *flaky tests*. E por fim, a seção 3.5 apresenta uma comparação entre os trabalhos apresentados e o trabalho proposto.

3.1 *Test Smell Detection Tools: A Systematic Mapping Studys*

Em (ALJEDAANI *et al.*, 2021), os autores propuseram fornecer uma fonte completa englobando ferramentas de detecção de *tests smells*, onde pesquisadores poderão selecionar a ferramenta mais apropriada para a sua tarefa de pesquisa. Neste estudo, os pesquisadores contribuíram com um catálogo de 22 publicações de ferramentas de detecção de *tests smells* e publicações que utilizam essas ferramentas, um acervo de experimentos destacando o crescimento de comparação das principais particularidades das ferramentas e uma discussão de como as descobertas da pesquisa podem contribuir para futuras pesquisas no campo.

No trabalho foi realizado um estudo de mapeamento sistemático a fim de evidenciar uma compreensão sobre a existência, características e adoção em estudos acadêmicos das ferramentas de detecção de *tests smells*. A pesquisa consiste em três fases: Planejamento, execução e síntese.

A primeira fase consiste em extrair uma considerável quantidade de publicações referentes ao tema. Para isso, Aljedaani *et al.* (2021) aplicaram uma estratégia para busca de publicações que consistiu em utilizar um conjunto específico de palavras chave relacionadas a *tests smells*, para pesquisar em bibliotecas digitais populares. Além disso, foram definidos critérios de inclusão e exclusão para serem aplicados na etapa seguinte.

A segunda fase consistiu em filtrar as publicações que foram obtidas na pesquisa nas bibliotecas digitais. Nesse processo, foram realizadas 4 etapas manuais para realizar a filtragem, onde na primeira etapa foram removidas publicações duplicadas, na segunda etapa foram aplicadas os critérios de inclusão e exclusão definidos, na terceira etapa foi realizada uma

análise por completo de cada publicação selecionada e na quarta e última etapa a técnica de *Snowball sampling (Snowballing)* foi utilizada para coletar mais publicações. No final dessa fase, 47 publicações foram selecionadas.

A terceira fase consiste em sintetizar os resultados obtidos para então evidenciar quais as ferramentas para detecção de *tests smells* estão disponíveis e quais as características dessas ferramentas. Para isso, as publicações foram divididas entre dois tipos: Publicações que destacam o desenvolvimento de ferramentas e publicações que destacam a adoção de ferramentas. Além disso, as publicações coletadas foram classificadas por ano de publicação e o local. As publicações que destacam o desenvolvimento de ferramentas foram revisadas manualmente e para cada ferramenta, foram extraídas os tipos de *tests smells* que a ferramenta detecta e outros recursos da ferramenta.

3.2 *A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests*

Em (HABCHI *et al.*, 2022), os autores realizaram um estudo empírico focado no contexto industrial onde os *flaky tests* se manifestam, visto que os profissionais necessitam identificar e investigar esses testes devido aos impactos que os mesmos causam no processo de desenvolvimento de software. Para isso, 14 profissionais foram entrevistados e com os dados coletados nesse processo foi realizada uma análise qualitativa a fim de responder às seguintes questões de pesquisa: (1) Onde podemos localizar os *flaky tests*? (2) Como os profissionais percebem o impacto dos *flaky tests*? (3) Como os profissionais lidam com os *flaky tests*? (4) Como as medidas de mitigação podem ser melhoradas com ferramentas de automação?

Habchi *et al.* (2022) optaram por realizar entrevistas semiestruturadas, que se baseia em algumas perguntas iniciais, e de acordo com as respostas obtidas do entrevistado vão desenvolvendo outras perguntas sobre o assunto que exploram outros pontos. Foram desenvolvidas 17 perguntas divididas entre assuntos sobre o contexto geral da infraestrutura de testes, contexto geral sobre *flaky tests* e sobre as ações tomadas pelos participantes para prevenir e resolver ou mitigar *flaky tests*. Essas perguntas foram aplicadas em profissionais com experiência em lidar com *flaky tests*.

Após o processo de entrevistas, os dados qualitativos em relação ao estudo foram fornecidos. Para isso, cada questão de pesquisa mencionada anteriormente foi dividida em diversas subseções, onde foram analisadas individualmente de acordo com as respostas obtidas. No fim desse estudo, foi possível observar que a análise de *flaky tests* precisa considerar todo o

ecossistema de testes e não deve se limitar apenas ao teste e código em teste, além de destacar um impacto mais amplo de *flaky tests* nas práticas de teste.

3.3 *Root Causing, Detecting, and Fixing Flaky Tests: State of The Art and Future Roadmap*

Em (ZOLFAGHARI *et al.*, 2021), os autores apresentaram uma revisão sistemática dos feitos mais recentes da indústria e da academia em relação à detecção e mitigação de *flaky tests*, contribuindo para abrir o caminho para pesquisas que podem ser realizadas futuramente sobre esse assunto, podendo ajudar a área acadêmica a acompanhar os avanços industriais.

Visto que Zolfaghari *et al.* (2021) adotaram uma abordagem sistemática para conduzir a pesquisa, primeiramente definiram 6 questões de pesquisa e criaram uma estratégia para extração de publicações, onde foram definidas algumas bibliotecas digitais populares, onde seriam realizadas a busca de publicações, palavras-chave referentes ao tema para construir uma string de busca e critérios de inclusão e exclusão para o processo de filtragem. Após esse processo, as publicações obtidas foram filtradas em mais algumas etapas, onde os critérios definidos anteriormente foram aplicados, publicações duplicadas foram removidas e por último foi realizado uma análise manual em cada publicação.

Por último, foram utilizados os resultados obtidos com a revisão dos trabalhos mais relevantes filtrados para responder às seguintes questões de pesquisa: (1) Quais abordagens e técnicas foram usadas para detectar *flaky tests*? (2) Quais abordagens e técnicas foram usadas para corrigir *flaky tests*? (3) Quais estudos empíricos foram feitos no domínio de *flaky tests* e na qualidade geral dos estudos? (4) Quais são as causas gerais de *flaky tests*? (5) Quais são as ferramentas automatizadas de suporte para detectar *flaky tests*? (6) Quais são as possíveis direções de pesquisa com base no atual problema/desafios neste domínio?

3.4 *A Survey of Flaky Tests*

Em (PARRY *et al.*, 2021), os autores apresentaram um levantamento sistemático do corpo de literatura relevante para a pesquisa de *flaky tests*, contribuindo com uma pesquisa abrangente que cobre a literatura para entender como os *flaky tests* influenciam o campo da engenharia de software.

Para o processo de levantamento sistemático, realizaram uma estratégia onde selecionaram termos de consulta que pudessem abranger os *flaky tests* e então aplicaram em

bibliotecas digitais definidas anteriormente. Depois avaliaram os artigos de acordo com os critérios de inclusão e exclusão também definidos e por fim, aplicaram a técnica de Snowballing. Além da avaliação da literatura, Parry *et al.* (2021) tinham como objetivo fornecer uma análise comparativa sobre as causas de *flaky tests*, um resumo dos custos e impactos dos *flaky tests* sobre a confiabilidade dos testes, uma comparação das ferramentas e estratégias disponíveis para detecção de *flaky tests*, uma análise das técnicas de mitigação e reparo e por fim a identificação de linhas de pesquisa, tendências e direções futuras no campo de *flaky tests*.

Os resultados obtidos foram utilizados para responder às seguintes questões de pesquisa: (1) Quais são as causas e fatores associados aos *flaky tests*? (2) Quais são os custos e consequências associados aos *flaky tests*? (3) Quais *insights* e técnicas podem ser aplicadas para detectar *flaky tests*? (4) Quais *insights* e técnicas podem ser aplicadas para mitigar ou reparar *flaky tests*?

3.5 Análise comparativa

Nesta seção é apresentado uma comparação entre os trabalhos relacionados descritos anteriormente e o trabalho aqui proposto. No Quadro 1, é possível observar um resumo das características de cada trabalho.

O trabalho de Aljedaani *et al.* (2021) fornece uma visão mais detalhada sobre o processo de coleta de ferramentas que serão reutilizadas, com adaptações, no trabalho aqui proposto. É necessário salientar que o trabalho de Aljedaani *et al.* (2021) aplica sua metodologia para coletar ferramentas de detecção de *tests smells*, o que o diferencia do trabalho aqui proposto, que propõe coletar ferramentas de detecção, mitigação e reparo de *flaky tests*.

Em seu trabalho, Habchi *et al.* (2022) forneceram dados qualitativos sobre as fontes, impactos e estratégias de mitigação de *flaky tests* baseados nos dados obtidos de 14 entrevistas realizadas com profissionais da área. A análise qualitativa também será utilizada no trabalho aqui proposto, porém, o objetivo é analisar os dados obtidos das etapas de planejamento e execução do mapeamento sistemático e então fornecer dados qualitativos sobre as ferramentas obtidas que abordam *flaky tests*, como seus objetivos e técnicas utilizadas.

Assim como o trabalho de Aljedaani *et al.* (2021), a pesquisa de Zolfaghari *et al.* (2021) também realiza um mapeamento sistemático. Porém, apesar da sua busca pelos feitos mais recentes da indústria e da academia em relação à detecção e mitigação de *flaky tests*, uma de suas questões de pesquisa trata diretamente de ferramentas de detecção de *flaky tests*, onde é

elaborada uma lista com algumas ferramentas e poucas características. Apesar de também conter a etapa de mapeamento sistemático citada, o trabalho aqui proposto fornecerá não só uma lista de ferramentas de detecção de *flaky tests*, mas um catálogo dessas ferramentas, contendo suas principais características.

O trabalho de Parry *et al.* (2021) realiza um levantamento sistemático, que além de buscar fornecer dados sobre as causas, impactos e custos dos *flaky tests*, também tem como objetivo apresentar ferramentas de *flaky tests* e dados qualitativos sobre as mesmas, fornecendo até mesmo uma lista de ferramentas e suas descrições. O trabalho aqui proposto, além de fornecer dados qualitativos e discorrer sobre as ferramentas, também irá apresentar um catálogo com as ferramentas e suas características de alto nível.

O Quadro 1 compara todos os 4 trabalhos relacionados especificados anteriormente com esse trabalho, considerando os seus principais fatores:

1. Realiza uma revisão da literatura.
2. Apresenta ferramentas que abordam *flaky tests*.
3. Fornece dados qualitativos.
4. Cataloga as ferramentas de detecção e suas características.

Quadro 1: Comparativo entre os trabalhos relacionados e o trabalho proposto

Trabalhos	**Fator 1**	**Fator 2**	**Fator 3**	**Fator 4**
Aljedaani *et al.* (2021)	X			X
Habchi *et al.* (2022)			X	
Zolfaghari *et al.* (2021)	X	X		
Parry *et al.* (2021)	X	X	X	
Trabalho proposto	X	X	X	X

Fonte: Elaborado pelo autor.

4 PROCEDIMENTOS METODOLÓGICOS

Neste capítulo, será apresentado um conjunto de etapas necessárias para a realização do objetivo do trabalho proposto, que consiste na realização de um mapeamento sistemático adaptado dos processos realizados por Petersen *et al.* (2015) e Keele *et al.* (2007). Esta etapa de mapeamento visa amplificar a compreensão existente sobre a existência e utilização de ferramentas de detecção de *flaky tests*, assim como suas características e adesão em pesquisas e mercado. Dessa forma, as ferramentas serão identificadas, analisadas e fornecidas. O mapeamento sistemático que será realizado consiste em três etapas: Planejamento, Execução e Análise ou Síntese.

4.1 Planejamento

Na fase de planejamento, serão detalhadas as estratégias de busca de publicações. De acordo com o mapeamento sistemático, foram utilizadas palavras chave relacionadas ao domínio de *flaky tests*, para então pesquisar em determinadas bibliotecas digitais, por publicações que atendam aos requisitos do trabalho proposto. Além disso, foram definidos alguns critérios de inclusão e exclusão para serem implementados, durante a fase de execução, nas publicações extraídas.

Para agrupar os resultados obtidos durante a fase de Planejamento e das outras duas etapas citadas anteriormente, a ferramenta *Parsifal*[1] será utilizada, visto que a mesma é bastante completa, fornecendo meios de documentar todo o processo do mapeamento sistemático, proporcionando um maior controle sobre cada etapa do processo.

4.2 Execução

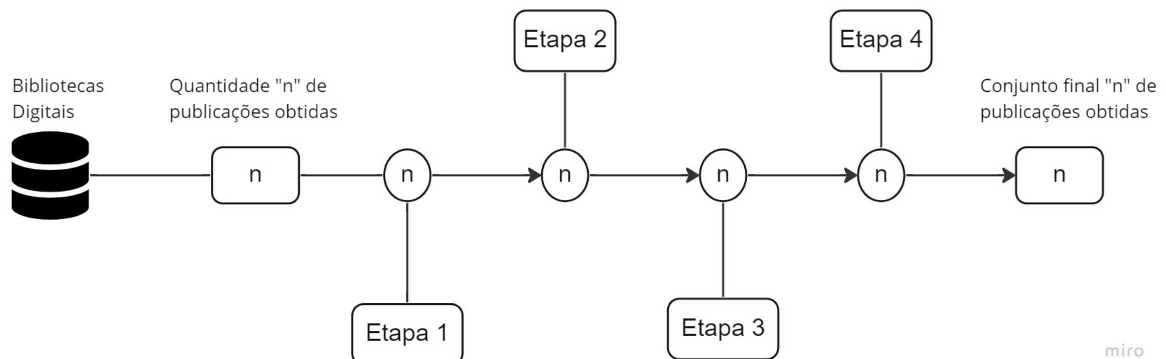
A fase de execução ocorrerá após a realização da primeira seleção de artigos publicados. A partir dos resultados obtidos, serão aplicadas manualmente as seguintes quatro etapas para realizar uma nova filtragem:

- **Etapa 1**: Remoção de publicações duplicadas
- **Etapa 2**: Aplicação de critérios de inclusão e exclusão
- **Etapa 3**: Leitura completa da publicação
- **Etapa 4**: Aplicação de *Snowballing*

[1] https://parsif.al

A administração dos dados obtidos será realizada por meio da ferramenta *Parsifal*, onde todas as informações relacionadas aos artigos serão documentadas, assim como em todas as outras etapas do mapeamento sistemático. O processo de execução pode ser visualizado na Figura 3.

Figura 3: Processo de filtragem de publicações



Fonte: Elaborado pelo autor.

Na primeira etapa do processo de filtragem, serão removidas da lista todas publicações que estejam duplicadas. Na segunda etapa, mais publicações serão filtradas de acordo com os critérios de inclusão e exclusão definidos, onde o objetivo é excluir todos os estudos que não sejam relevantes para responder às questões de pesquisa definidas anteriormente.

Na terceira etapa, cada publicação será explorada individualmente, a fim de garantir que seu conteúdo apresenta informações pertinentes para o trabalho. Na quarta e última etapa, será implementada a técnica de *Backward Snowballing* que consiste em percorrer a lista de referências das publicações coletadas, selecionando artigos que atendam aos critérios básicos de inclusão e exclusão para obter novas publicações que podem não ter sido encontradas durante o processo (JALALI; WOHLIN, 2012). Ao fim dessa etapa, é esperado que a quantidade de publicações seja reduzida, contendo apenas estudos que contribuam com o assunto do trabalho.

4.3 Análise

Na fase de análise ou síntese, todas as publicações obtidas após o processo de filtragem da etapa de execução serão analisadas a fim de obter informações pertinentes para responder às questões de pesquisa citadas anteriormente. Nesse processo, as ferramentas filtradas serão analisadas e agrupadas de acordo com suas atribuições no contexto de *flaky tests*, como detecção, mitigação e reparo.

Durante a etapa de análise, também é esperado obter, para cada ferramenta coletada, os tipos de técnicas que a ferramenta em questão utiliza, entre outros recursos da ferramenta, sendo possível construir um catálogo de acordo com todas as características de alto nível que foram obtidas durante o mapeamento sistemático, como linguagem de programação suportada, *framework* de teste suportados, etc, a fim de obter respostas para as questões de pesquisa definidas que foram citadas anteriormente, com o objetivo de fornecer uma base inicial para futuras pesquisas nesta área.

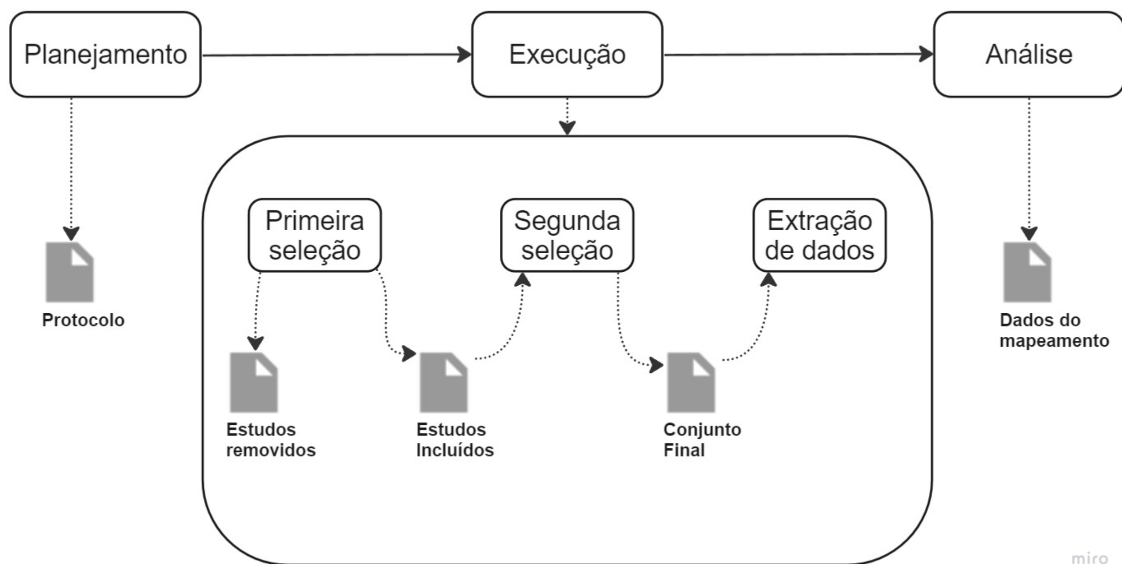
5 MAPEAMENTO SISTEMÁTICO DE FERRAMENTAS PARA DETECÇÃO DE *FLAKY TESTS*

Nesta seção, serão apresentados os resultados de cada etapa do processo de mapeamento sistemático realizado neste trabalho. A subseção 5.1 discorre sobre a concepção de um mapeamento sistemático e seus passos. A subseção 5.2 demonstra os resultados obtidos na etapa de planejamento. Na subseção 5.3 é apresentada a etapa de execução do mapeamento sistemático e a subseção 5.4 apresenta uma síntese dos resultados obtidos durante as etapas. A subseção 5.5 apresenta uma discussão dos resultados obtidos ao fim do mapeamento sistemático.

5.1 Introdução

De acordo com Petersen *et al.* (2015), os mapeamentos sistemáticos fornecem uma visão geral de uma área de pesquisa. Dessa maneira, é possível então que seja realizado uma coleta e síntese dos elementos encontrados a fim de responder questões de pesquisa que refletem esse objetivo.

Figura 4: Diagrama de fluxo dos procedimentos metodológicos



Fonte: Elaborado pelo autor.

O mapeamento sistemático consiste nas etapas de planejamento, execução e análise. Na etapa de planejamento é organizada uma estrutura, com especificação de alguns elementos, para que seja aplicada na etapa de execução. Na etapa de execução ocorre uma triagem de estudos, para extração de seus dados e então retornando uma seleção de estudos pertinentes ao

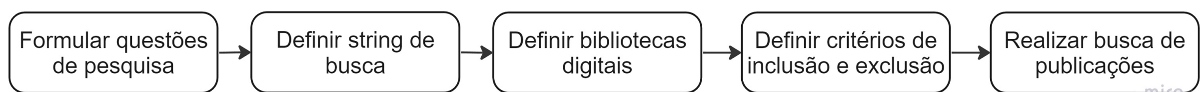
assunto. Por último, na etapa de análise, os resultados obtidos são sintetizados (KEELE *et al.*, 2007). A Figura 4 demonstra essas etapas e quais atividades presentes em cada uma delas, que serão descritas nas subseções subsequentes.

O mapeamento sistemático utilizado neste trabalho foi adaptado dos processos realizados por Petersen *et al.* (2015) e Keele *et al.* (2007). A ferramenta *Parsifal* foi utilizada para a documentação e organização dos estudos obtidos.

5.2 Planejamento

As atividades da etapa de planejamento podem ser visualizadas de maneira mais detalhada na Figura 5.

Figura 5: Atividades da etapa de planejamento



Fonte: Elaborado pelo autor.

Visto que será realizado um mapeamento sistemático, o trabalho proposto examina a literatura a fim de coletar informações sobre um tema específico de engenharia de software, para então fornecer uma maior compreensão sobre as ferramentas de detecção de *flaky tests* e responder três perguntas de pesquisa exploratória (BARN *et al.*, 2017). Para isso, na primeira atividade, foram definidas as seguintes questões:

- **QP1**: Quais ferramentas de detecção de *flaky tests* estão disponíveis para a comunidade?
- **QP2**: Quais os objetivos das ferramentas de detecção de *flaky tests* e quais técnicas elas utilizam?
- **QP3**: Quais são as principais características das ferramentas de detecção de *flaky tests* e quais causas elas abordam?

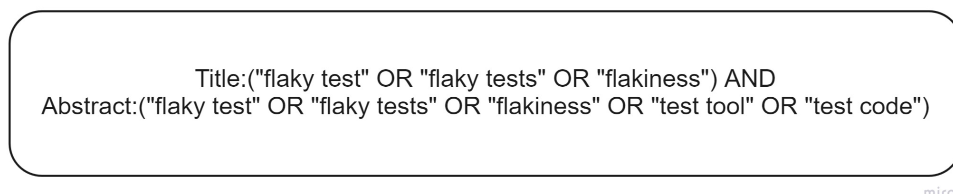
A primeira questão de pesquisa foi escolhida com intuito de centralizar as ferramentas de detecção de *flaky tests* que estão disponíveis atualmente, visto que ao longo dos anos novas ferramentas são criadas e publicadas, dessa forma, é possível atualizar os pesquisadores e profissionais da área. A segunda questão de pesquisa tem o objetivo de explicar a função de cada ferramenta coletada, isto é, em que contexto de *flaky tests* ela atua, quais técnicas utilizadas, e quando possível, uma comparação com outras ferramentas que possuem funções parecidas. A terceira e última questão de pesquisa tem o objetivo de fornecer uma lista com todas as

ferramentas e suas características em alto nível e de maneira centralizada, além de mostrar as causas de *flaky tests* que elas abordam, ajudando os pesquisadores e profissionais da área a ter uma visão geral das ferramentas atuais e conseguir encontrar informações de uma maneira mais ágil e facilitada, podendo, por exemplo, escolher uma ou mais ferramentas para um estudo relacionado.

Para determinar as palavras chave mais pertinentes para extração de publicações, na segunda atividade, foi realizada uma pesquisa piloto nas bibliotecas digitais IEEE e ACM, bastante renomadas na área de pesquisa. Durante esse processo manual, foram consultadas publicações que contivessem o termo *flaky test* ou *flaky tests* em seu título ou em seu resumo, a fim de identificar palavras e termos que pudessem ser utilizadas na pesquisa principal.

Após a pesquisa piloto descrita, algumas palavras chave foram determinadas: *Flaky Test*, *Flaky Tests*, *Flakiness*, *Test Tool* e *Test Code*. Com essas palavras, foi definida uma string de busca para a realização da pesquisa principal, que será aplicada apenas no título e resumo das publicações a fim de evitar resultados falso positivos. A string definida pode ser visualizada na Figura 6.

Figura 6: String de pesquisa



Fonte: Elaborado pelo autor

Na terceira atividade e após a pesquisa piloto, além do IEEE e ACM, outras bibliotecas digitais, que contém publicações da área de Engenharia de Software e Ciência da Computação, foram selecionadas para a pesquisa principal e podem ser visualizadas no Quadro 2.

Quadro 2: Bibliotecas digitais utilizadas na pesquisa

Biblioteca Digital	**URL**
ACM Digital Library	https://dl.acm.org/
IEEE Xplore	https://ieeexplore.ieee.org/
Science Direct	https://www.sciencedirect.com/
Scopus	https://www.scopus.com/
Springer Link	https://link.springer.com/

Fonte: Elaborado pelo autor

Após a primeira filtragem de publicações utilizando a string de busca definida anteriormente, será realizado uma segunda filtragem por meio de 4 etapas, dessa vez manualmente, que será descrita na subseção da atividade de execução. Para direcionar melhor o processo de filtragem manual e conseguir coletar publicações que atendam melhor o objetivo do trabalho proposto, alguns critérios de inclusão e exclusão foram definidos na quarta atividade e podem ser visualizados no Quadro 3.

Quadro 3: Critérios de inclusão e exclusão

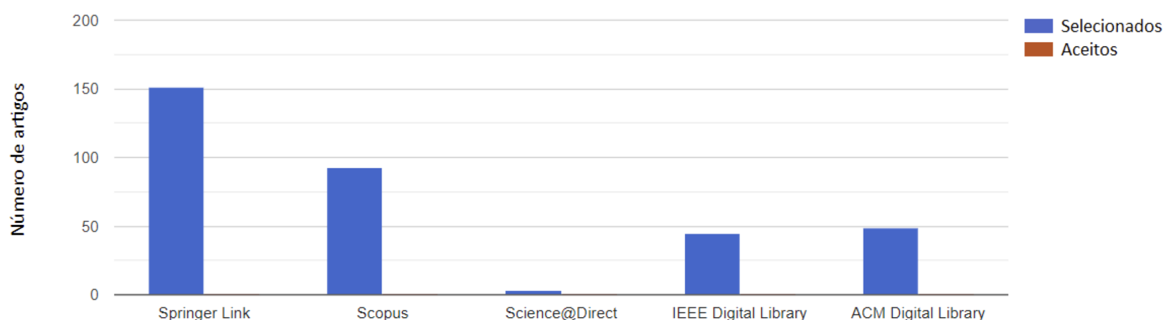
Inclusão	**Exclusão**
O artigo propõe, cita ou utiliza uma ferramenta de detecção de *flaky tests*	Trabalho fora do escopo de detecção de flaky tests
O resumo cita detecção de flaky tests	Trabalhos com menos de duas páginas
O resumo cita alguma ferramenta de detecção de *flaky tests*	Trabalhos que não estão em inglês
	Sites, livros, folhetos e literatura cinzenta
	Texto completo não disponível online
	Trabalhos que não citam ferramentas de detecção de *flaky tests*

Fonte: Elaborado pelo autor

A quinta e última atividade dessa etapa consiste em utilizar a *string* de busca definida para coletar um conjunto de publicações que serão avaliadas durante o mapeamento. Logo, a *string* foi aplicada nas 5 bibliotecas digitais: ACM Library, SpringerLink, IEEE Xplore, ScienceDirect e Scopus.

Após a realização da busca nas bibliotecas, todos os 341 trabalhos obtidos foram registrados e organizados na ferramenta Parsifal, que será utilizado para apoiar o gerenciamento de dados das etapas do mapeamento. Na Figura 7 é possível visualizar a divisão de artigos por biblioteca digital.

Figura 7: Estudo por bibliotecas digitais

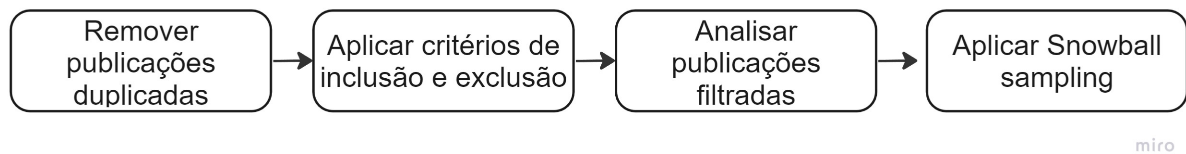


Fonte: Elaborado pelo autor.

5.3 Execução

A etapa de execução foi iniciada em 11 de fevereiro de 2023 e finalizada no dia 26 de março de 2023 e abordou a filtragem de todos os 341 artigos obtidos anteriormente, durante a etapa de planejamento, das bibliotecas digitais ACM Library, SpringerLink, IEEE Xplore, ScienceDirect e Scopus. As atividades realizadas durante essa etapa podem ser visualizadas na Figura 8.

Figura 8: Atividades da etapa de execução



Fonte: Elaborado pelo autor.

Na primeira atividade, foram constatados 95 trabalhos duplicados, portanto, os mesmos foram removidos. Após a remoção, restaram um total de 246 trabalhos para serem analisados na atividade seguinte. Na segunda atividade, a filtragem foi executada de acordo com os critérios de inclusão e exclusão definidos anteriormente, utilizando a ferramenta Parsifal para atribuição dos critérios de inclusão ou exclusão para cada estudo avaliado, permitindo o gerenciamento dos dados obtidos. Os critérios foram aplicados no resumo e corpo das publicações.

Foram removidos 173 artigos durante essa atividade, resultando em um total de 73 artigos obtidos para serem utilizados no processo de filtragem seguinte do mapeamento. É possível visualizar abaixo uma relação entre a quantidade de artigos removidos e os critérios de exclusão em questão:

- **EXC1**: Trabalho fora do escopo de detecção de *flaky tests*: 116
- **EXC2**: Trabalhos com menos de duas páginas: 2
- **EXC3**: Trabalhos que não estão em inglês: 0
- **EXC4**: Sites, livros, folhetos e literatura cinzenta: 23
- **EXC5**: Texto completo não disponível online: 1
- **EXC6**: Trabalhos que não citam ferramentas de detecção de *flaky tests*: 31

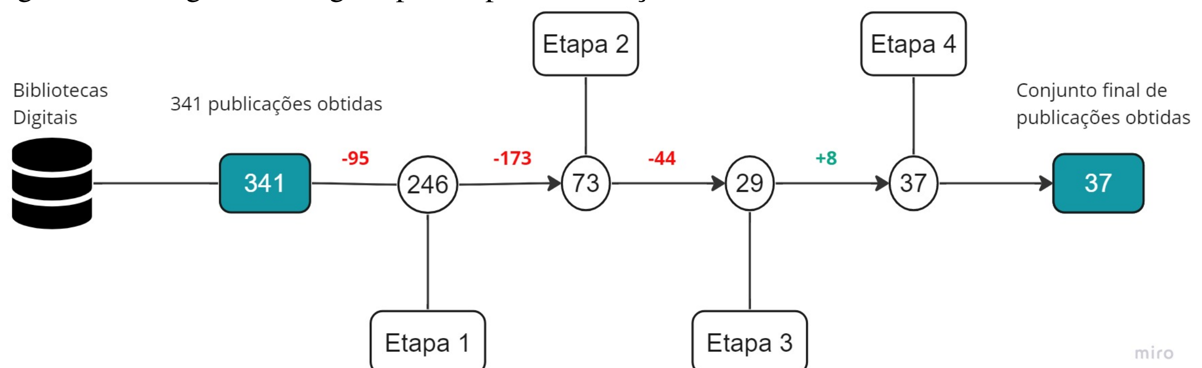
Após essa atividade, foi realizado um processo de leitura mais detalhado nos 73 artigos restantes, com o objetivo de descorir se os trabalhos realmente apresentavam informações

pertinentes para o tema do trabalho. Durante a atividade foram removidas 44 publicações, resultando em um total de 29 artigos.

Com o objetivo de expandir possíveis referências não identificadas durante o processo, a técnica de *Backward* Snowballing foi aplicada nos 29 artigos restantes, obtendo-se 8 novos artigos com dados úteis para a pesquisa, resultando em um total de 37 artigos válidos para a extração de dados durante a etapa de análise do mapeamento sistemático.

A etapa de execução foi iniciada com 341 artigos e finalizada com 37 artigos filtrados, no qual 29 artigos foram obtidos ao fim das 3 primeiras atividades e mais 8 artigos foram obtidos durante a atividade de Snowballing. O resumo do processo pode ser visualizado na Figura 9.

Figura 9: Filtragem de artigos após etapa de execução



Fonte: Elaborado pelo autor.

No Quadro 4 é possível observar a quantidade de publicações filtradas por biblioteca digital em cada etapa do processo de execução.

- **Etapa 1**: Remoção de publicações duplicadas
- **Etapa 2**: Aplicação de critérios de inclusão e exclusão
- **Etapa 3**: Leitura completa da publicação
- **Etapa 4**: Aplicação de *Snowballing*

Quadro 4: Filtragem por biblioteca digital

Bibliotecas	**Artigos Retornados**	**Etapa 1**	**Etapa 2**	**Etapa 3**	**Etapa 4**
ACM Library	49	10	0	0	7
SpringerLink	151	139	1	0	0
IEEE Xplore	45	5	2	1	2
ScienceDirect	3	3	1	0	0
Scopus	93	89	69	28	28
Total	**341**	**246**	**73**	**29**	**37**

Fonte: Elaborado pelo autor.

5.4 Análise

Durante a etapa de análise, que teve seu início em 30 de março e foi finalizada no dia 21 de junho, todas as 37 publicações filtradas durante a etapa de execução foram inspecionadas a fim de obter informações em relação às ferramentas de detecção de *flaky tests* e então responder às questões de pesquisa definidas anteriormente. A seção foi dividida em 3 subseções, onde a primeira aborda quais as ferramentas de detecção de *flaky tests* estão disponíveis para a comunidade, a segunda apresenta uma visão geral das ferramentas e quais técnicas elas utilizam e a terceira apresenta as principais características das ferramentas coletadas. O Apêndice A possui a Tabela 5, que apresenta todos os 37 estudos que foram retornados na fase de execução.

5.4.1 QP1 - Quais ferramentas de detecção de flaky tests estão disponíveis para a comunidade?

Essa primeira questão de pesquisa busca mostrar as ferramentas encontradas e disponíveis para a comunidade durante o estudo dos artigos na fase de execução do mapeamento sistemático. Foram encontradas um total de 30 ferramentas que tiveram seus respectivos artigos publicados entre os anos 2014 e 2022. No Quadro 5 é possível visualizar quais ferramentas foram publicadas e a quantidade total em cada ano.

Quadro 5: Publicação das ferramentas de detecção de *flaky tests* para a comunidade

Data (Ano)	**Ferramenta(s)**	**Total**
2014	DTDetector, VmVm e OrcalePolish	3
2015	ElectricTest e PolDet	2
2016	NonDex	1
2018	DeFlaker e PraDet	2
2019	iDFlakies, RootFinder, IFixFlakies e TEDD	4
2020	MOKA, FaTB, Shaker, FLASH e FlakyLoc	5
2021	FlakeFlagger, DexFix, FLAST, FlakeScanner e MDFlaker	5
2022	FlakiMe, ODRepair, FLAKE16, Flakify, iPFlakies, PEELER, FlakeRepro e IncIDFlakies	8

Fonte: Elaborado pelo autor

É possível observar que a partir do ano de 2019 houve um crescimento mais estável em relação a quantidade de ferramentas publicadas por ano e obtendo uma quantidade máxima de 8 ferramentas em 2022, demonstrando um iminente crescimento de pesquisas na área. Além disso, 4 das 13 ferramentas publicadas em 2021 e 2022 não utilizam a técnica de reexecução de testes, que já foi citada anteriormente como a maneira mais tradicional para apoio na detecção

de *flaky tests*, e isso pode ser interessante visto que a reexecução pode ser bastante demorada dependendo da quantidade de vezes aplicada e tamanho da suíte de testes automatizados, logo, com o crescimento das pesquisas em um período recente, os estudos estão buscando novos caminhos para a detecção de *flaky tests*.

Durante a segunda questão de pesquisa, as ferramentas listadas no Quadro 5 serão abordadas com mais detalhes, fornecendo os objetivos de cada uma e os estudos das quais foram coletadas.

5.4.2 *QP2 - Quais os objetivos das ferramentas de detecção de flaky tests e quais técnicas elas utilizam?*

A segunda questão de pesquisa tem o intuito de fornecer uma visão geral de cada uma das 30 ferramentas coletadas durante a pesquisa. Essa visão geral engloba os objetivos, técnicas ou abordagens e o estudo da qual cada ferramenta foi coletada.

A técnica de reexecução de testes, como seu próprio nome sugere, consiste em reexecutar a suíte de testes 1 ou mais vezes a fim de verificar se o teste passará ou não, na tentativa de identificar inconsistências em seus resultados. Essa técnica é a maneira mais tradicional de apoio à detecção de *flaky tests* e esse dado foi constatado durante a pesquisa, visto que 25 das 30 ferramentas, ou seja, 83% das ferramentas, a utilizam.

A visão geral das ferramentas será fornecida nas 4 subseções a seguir, de acordo com suas atribuições no contexto de *flaky tests*, que são: detecção, mitigação e reparo. Alguns estudos informam que suas ferramentas abordam especificamente detecção de *flaky tests* em testes dependentes de ordem, que é uma das principais causas de *flaky tests*, portanto, as ferramentas que participam desse contexto serão explicadas em uma subseção individual.

5.4.2.1 *Ferramentas e técnicas para detecção de flaky tests*

As ferramentas de detecção tem como objetivo identificar *flaky tests* ou apoiar nesse processo, fornecendo margem para um processo de mitigação dos problemas ou reparo dos testes. Nos estudos analisados, foi possível observar os objetivos gerais das ferramentas, assim como algumas técnicas e abordagens utilizadas nesse processo. No Quadro 6 é possível visualizar as ferramentas que atuam no contexto de detecção de *flaky tests*, suas técnicas, abordagens, estudos nas quais foram encontradas e seu Identificador (ID).

A ferramenta DeFlaker, estudada no **S1**, utiliza a técnica cobertura diferencial para

detectar *flaky tests*, que consiste em verificar o código modificado, de acordo com o sistema de controle de versão. Então, após uma nova versão do software em teste, ou seja, após um *commit*, se o resultado do teste mudar mesmo sem cobrir nenhum código modificado o mesmo é considerado um *flaky test*. O MDFlaker, abordado no estudo **S9**, também utiliza a técnica de cobertura diferencial, porém, ao contrário do DeFlaker, não necessita da reexecução dos testes, economizando tempo de teste e custo.

Quadro 6: Ferramentas de detecção de *flaky tests*

Ferramenta	**Técnica(s) / Abordagem**	**Estudo / Ano**	**ID**
DeFlaker	Cobertura diferencial Reexecução	Bell *et al.* (2018)	S1
FlakeFlagger	Aprendizado de máquina na detecção	Alshammari *et al.* (2021)	S2
Shaker	Ambiente de Execução com Estresse Reexecução	Silva *et al.* (2020) Cordeiro *et al.* (2021)	S3 S31
FLASH	Aprendizado de máquina na detecção Reexecução	Dutta *et al.* (2020)	S4
NonDex	Especificações não determinísticas Reexecução	Shi *et al.* (2016) Gyori *et al.* (2016)	S5 S32
FLAST	Aprendizado de máquina na detecção	Verdecchia *et al.* (2021)	S6
Flakify	Aplicando previsão em código-fonte	Fatima *et al.* (2022)	S7
PEELER	Aprendizado de máquina na detecção	Qin *et al.* (2022)	S8
MDFlaker	Cobertura diferencial	Ahmad *et al.* (2021)	S9

Fonte: Elaborado pelo autor

A ferramenta FlakeFlagger foi estudada no **S2** e é uma abordagem que utiliza aprendizado de máquina para detecção, onde a mesma coleta um conjunto de recursos dinâmicos e estáticos como cobertura de linha e código-fonte, por exemplo, e em seguida prevê testes que provavelmente serão *flaky tests* com base no comportamento semelhante às características coletadas. A ferramenta PEELER, estudada no **S8** também aplica aprendizado de máquina para detecção, porém, diferente do FlakeFlagger, utiliza uma abordagem totalmente estática, onde captura os logs de falha de teste e os compara com os valores envolvidos nas declarações de asserção dos testes. A ferramenta FLAST, estudada no **S6**, também utiliza um modelo de aprendizado de máquina e tem como objetivo classificar testes como *flaky tests* ou não, baseados puramente em recursos estáticos. Assim, o objetivo é um método que possa detectar a tempo um *flaky test* antes mesmo de ser executado. A ferramenta FLASH, estudada no **S4**, ao contrário das 3 ferramentas citadas anteriormente, utiliza a reexecução de testes aliado ao aprendizado de máquina. Basicamente, essa ferramenta executa os testes diversas vezes com diferentes entradas aleatórias, retornando sequências diferentes de resultados. O FLASH então verifica as diferenças

entre os valores que deveriam ser retornados e os valores dos resultados dos testes aleatórios, identificando *flaky tests*.

A ferramenta NonDex, estudada no **S5** e **S32**, realiza a detecção explorando implementações não determinísticas, ou seja, busca aleatoriamente diferentes comportamentos de APIs subdeterminadas durante a execução do teste. Quando um teste falha durante a busca, o NonDex procura a instância de invocação da API que causou a falha. Por exemplo, a ordem de iteração de um algoritmo de ordenação é subdeterminada e o código de teste, que assume alguma ordem de iteração específica da implementação, pode falhar. O NonDex explora aleatoriamente diferentes ordens de iteração e quando um teste falha durante essa exploração, essa falha demonstra que o código faz algumas suposições erradas.

A ferramenta Flakify, publicada em 2022 no estudo **S7**, utiliza uma solução de caixa preta para prever casos de *flaky tests*, utilizando apenas o código-fonte dos casos de teste, em oposição ao sistema em teste e não requer a reexecução de casos de teste várias vezes. A ferramenta Shaker, abordada nos estudos **S3** e **S31** e, tenta melhorar a eficiência da reexecução dos testes. Para isso, o Shaker adiciona tarefas de estresse para competir pela CPU ou memória, por exemplo, enquanto reexecuta o conjunto de testes, a fim de aumentar a probabilidade de manifestar *flaky tests* de espera assíncrona e simultaneidade. O estudo da ferramenta se baseia nas observações de que a simultaneidade é uma importante fonte de instabilidade e que a adição de estresse no ambiente pode interferir na ordenação dos eventos e, consequentemente, influenciar nas saídas dos testes.

5.4.2.2 Ferramentas e técnicas para detecção de flaky tests em testes dependentes de ordem

As ferramentas de detecção de testes dependentes de ordem possuem fatores e dependências que podem estar ligadas diretamente aos *flaky tests*, como por exemplo, quando os testes partilham de um mesmo conjunto de dados ou de um estado, ocasionando no problema. Nos estudos analisados, alguns informaram que suas ferramentas abordam especificamente detecção de *flaky tests* em testes dependentes de ordem, então essa subseção tem como intuito informar os objetivos gerais dessas ferramentas, assim como algumas técnicas e abordagens utilizadas durante o processo. Um total de 11 das 11 ferramentas que serão apresentadas, ou seja, 100% das ferramentas, utilizam a reexecução dos testes. No Quadro 7 é possível visualizar as ferramentas que atuam no contexto de detecção de *flaky tests* em testes dependentes de ordem, suas técnicas, abordagens e estudos nas quais foram encontradas.

A ferramenta OrcalePolish, estudada no **S17**, utiliza uma abordagem focada em detectar asserções frágeis, ou seja, asserções que dependem de valores derivados de entradas não controladas inicialmente e entradas fornecidas pelo teste que não são verificadas por uma asserção. Por exemplo, podemos considerar um teste que assume que um determinado campo está em seu valor padrão e faz uso dele em uma declaração de asserção. Esta é uma entrada não controlada, pois o próprio teste não define o valor padrão, e outra fonte, como um teste executado anteriormente, pode tê-lo modificado. Ou seja, asserções frágeis podem criar uma oportunidade para que ocorra o surgimento de testes dependentes de ordem, pois fazem com que o resultado de um teste dependa de entradas que ele não controla, mas que podem ser definidas anteriormente por outro teste.

Quadro 7: Ferramentas de detecção de *flaky tests* em testes dependentes de ordem

Ferramenta	Técnica(s) / Abordagem	Estudo / Ano	ID
iDFlakies	Repetição de testes com falha Reexecução	Lam *et al.* (2019)	S10
PraDet	Validação de dependência Reexecução	Gambi *et al.* (2018)	S11
ElectricTest	Validação de dependência Reexecução	Bell *et al.* (2015)	S12
DTDetector	Validação de dependência Reexecução	Zhang *et al.* (2014b)	S13
PolDet	Poluição de Teste Reexecução	Gyori *et al.* (2015)	S14
FLAKE16	Aplicativos de aprendizado de máquina Reexecução	Parry *et al.* (2022)	S15
FlakeScanner	Repetição de testes com falha Reexecução	Dong *et al.* (2021)	S16
OrcalePolish	Asserções frágeis Reexecução	Huo e Clause (2014)	S17
iPFlakies	Repetição de testes com falha Reexecução	Wang *et al.* (2022)	S18
TEDD	Validação de dependência Reexecução	Biagiola *et al.* (2019)	S19
IncIDFlakies	Repetição de testes com falha Reexecução	Li e Shi (2022)	S20

Fonte: Elaborado pelo autor

A ferramenta PolDet, abordada no estudo **S14**, tem como objetivo detectar testes que modificam o programa compartilhado ou o estado ambiente, criando potencialmente dependência de ordem de teste, ou seja, teste que poluem o estado de outros testes. Dessa forma, o PolDet encontra testes que poluem o estado, permitindo que os desenvolvedores consertem os testes imediatamente, em vez de posteriormente, quando a poluição possivelmente já tenha se

manifestado em uma falha de teste.

A ferramenta DTDetector, estudada no **S13**, tem como abordagem validar a dependência de ordem dos testes, logo, é necessário inverter a ordem do conjunto de testes, realizando permutações e então reexecutando-as a fim de identificar os testes dependentes de ordem. A ferramenta ElectricTest, estudada no **S12**, também utiliza a mesma abordagem da ferramenta DTDetector, de validar a dependência de ordem, mas de acordo com os autores a ferramenta ElectricTest fornece mais informações de rastreamento, como o nome da exceção em uma mensagem mais descritiva que indica o problema e a pilha de chamadas de método, diferente de seu antecessor que só poderia relatar que dois testes eram dependentes. Além disso, o ElectricTest se concentra em uma definição mais geral de dependência, por exemplo, se um teste 2 lê algum valor que foi escrito por último por um teste 1, pode-se dizer que teste 2 depende do teste 1 (ou seja, há uma dependência de dados). Se algum teste posterior, um teste 3, escrever sobre esses mesmos dados, dizemos que há uma antidependência entre os testes 1 e 3, logo, o teste 3 nunca deve ser executado entre os testes 1 e 2. Baseado nessa definição, a ferramenta diminui o escopo de possíveis dependências e consequentemente diminui a quantidade de reexecuções, a tornando mais rápida.

A ferramenta PraDet, abordada no estudo **S11**, utiliza a mesma abordagem das outras 2 citadas anteriormente, porém, combina a precisão do DTDetector e a velocidade do ElectricTest. Para isso, ela monitora os padrões de acesso de objetos na memória entre execuções de teste para identificar instâncias de possíveis dependências de ordem de teste. Ou seja, ela consegue identificar os testes envolvidos nessa dependência de ordem e então executa-as fora de ordem para tentar descobrir as dependências que são manifestas, reduzindo a quantidade de execuções de teste necessárias para expor as dependências manifestas.

O iDFlakies é uma ferramenta publicada em 2019 e estudada no **S10**, onde tem como objetivo classificar os testes em dependentes de ordem ou não, de acordo com a comparação dos resultados de reexecuções dos testes com falha em uma ordem de testes que foi modificada da ordem original. Basicamente, a ferramenta reexecuta a ordem original dos testes várias vezes para verificar se o resultado de algum teste muda e caso qualquer teste passe e falhe em uma mesma versão de código na mesma ordem de teste é, por definição, considerando como um *flaky test* não dependente de ordem. Os testes com falha são reexecutados e caso falhem novamente na ordem de falha e passem novamente na ordem original, serão consideradas como um *flaky test* dependente de ordem. A ferramenta iPFlakies, publicada em 2022 e estudada no **S18**, também

tem como objetivo classificar e categorizar os testes dependente de ordem repetindo os testes com falha em projetos desenvolvidos em Python, diferente do iDFlakies que tem como foco projetos em Java. O iPFlakies está categorizado junto com as ferramentas focadas no contexto de detecção de *flaky tests* em testes dependentes de ordem e também nas ferramentas do contexto de reparação, visto que ela basicamente é composta pelas funcionalidades das ferramentas iDFlakies e iFixFlakies. A ferramenta iFixFlakies será explicada na subseção 5.4.2.4.

A ferramenta FlakeScanner, estudada no **S16**, utiliza a técnica de repetição de testes com falha para identificar *flaky tests* de simultaneidade em aplicativos Android. A técnica explora possíveis ordens de execução dos eventos em testes assíncronos que falharam, de forma que cada execução de teste explore uma ordem diferente de execução dos eventos. Dessa forma, a ferramenta busca detectar *flaky tests* em poucas execuções de teste. A ferramenta IncIDFlakies, abordada no estudo **S20**, utiliza uma técnica para detectar *flaky tests* dependentes de ordem que foram recém introduzidos após uma mudança de código. O IncIDFlakies se baseia no iDFlakies, detectando *flaky tests* dependentes de ordem, executando novamente testes em ordens aleatórias. No entanto, IncIDFlakies também leva em consideração as mudanças de código desde a última vez que o IncIDFlakies foi executado, executando ordens de teste que consistem apenas no subconjunto de testes que podem se tornar *flaky tests* dependentes de ordem após a mudança do código.

A ferramenta TEDD, estudada no **S19**, tem como objetivo detectar *flaky tests* por meio de validação de dependência de teste presente nas suítes de testes de interface. A ferramenta detecta testes dependentes de ordem em aplicativos da web considerando dependências facilitadas por dados persistentes armazenados no lado do servidor, em vez de dependências internas, como o PraDet, que monitora os padrões de acesso de objetos na memória. A ferramenta FLAKE16, estudada no **S15**, apresenta uma técnica de detecção que utiliza aprendizado de máquina a fim de categorizar os *flaky tests* detectados. A ferramenta instala o projeto dentro de um ambiente virtual e cria uma instância para cada execução do conjunto de testes. Caso os testes demonstrem resultados inconsistentes durante as execuções em uma ordem consistente, são considerados *flaky tests* não dependentes de ordem. Caso contrário, ou seja, se tiverem resultados inconsistentes durante ordens aleatórias, são considerados *flaky tests* dependentes de ordem.

5.4.2.3 *Ferramentas e técnicas para mitigação de flaky tests*

Algumas ferramentas têm como objetivo utilizar técnicas para minimizar os custos e impactos negativos dos *flaky tests*, além da detecção já comentada. Essas ferramentas atuam em um contexto de mitigação. Um total de 3 ferramentas serão apresentadas e todas utilizam reexecução como técnica de apoio. No Quadro 8 é possível visualizar as ferramentas que atuam no contexto de mitigação de *flaky tests*, suas técnicas, abordagens e estudos nas quais foram encontradas.

Quadro 8: Ferramentas de mitigação de *flaky tests*

Ferramenta	**Técnica(s) / Abordagem**	**Estudo / Ano**	**ID**
FlakiMe	Geração automática de testes Reexecução	Cordy *et al.* (2022)	S21
VmVm	Geração automática de testes Testes dependentes de ordem Reexecução	Bell e Kaiser (2014)	S22
FlakyLoc	Teste da interface do usuário Reexecução	Barbón *et al.* (2020)	S23

Fonte: Elaborado pelo autor

A ferramenta VmVm, estudada no **S22**, apresenta uma abordagem para mitigar testes dependente de ordem. Basicamente, ela executa os casos de teste de maneira isolada, cada uma dentro de seu próprio processo. Dessa forma, ela consegue evitar que algum efeito colateral baseado no estado de cada caso de teste executado afete os testes posteriores, eliminando então dependências de ordem de teste que geralmente ocorrem por recursos compartilhados na memória. Baseando-se nesse processo, a ferramenta reinicializa as classes que contém campos estáticos que podem facilitar os efeitos colaterais baseados no estado de cada caso de teste explicado anteriormente. Em outras palavras, a ferramenta basicamente gera um novo estado para os testes em tempo de execução, diminuindo as chances de um possível ocasionamento de *flaky test*.

O FlakiMe é uma ferramenta publicada em 2022 e abordada no estudo **S21**, onde tem como objetivo fornecer aos desenvolvedores maneiras de simular um conjunto de cenários e condições para a ocorrência de *flaky tests*. Basicamente, a ferramenta permite gerar exceções durante a compilação dos casos de testes para que seja possível prever se o teste em questão falhará ou não, ajudando a prever se o teste pode desencadear um *flaky test*. A ferramenta FlakyLoc, estudada no **S23**, tem como objetivo identificar a causa raiz de falhas em aplicações web. Para isso, reexecuta os testes focados na interface do usuário de diferentes maneiras,

com base em testes combinatórios e analisa a execução do teste usando diferentes métricas de classificação. Os testes são feitos mudando o ambiente, como os núcleos da CPU, quantidade de memória RAM, navegador web e resolução de tela.

5.4.2.4 Ferramentas e técnicas para reparo de flaky tests

Em vez de utilizar técnicas para ajudar na mitigação de *flaky tests*, algumas ferramentas foram desenvolvidas e utilizam técnicas para apoiar os desenvolvedores na remoção dos *flaky tests* do conjunto de testes dos projetos em questão. Todas as 8 ferramentas que serão apresentadas utilizam a técnica de reexecução de testes como apoio. No Quadro 9 é possível visualizar as ferramentas que atuam no contexto de reparação de *flaky tests*, suas técnicas, abordagens e estudos nas quais foram encontradas.

Quadro 9: Ferramentas de reparo de *flaky tests*

Ferramenta	**Técnica(s) / Abordagem**	**Estudo / Ano**	**ID**
MOKA	Abordagens automáticas para gerar ou melhorar reparos Reexecução	Fazzini *et al.* (2020)	S24
RootFinder	Identificando as causas raiz para auxiliar no reparo Reexecução	Lam *et al.* (2019)	S25
FaTB	Abordagens automáticas para gerar ou melhorar reparos Reexecução	Lam *et al.* (2020)	S26
iFixFlakies	Abordagens automáticas para gerar ou melhorar reparos Reexecução	Shi *et al.* (2019)	S27
DexFix	Abordagens automáticas para gerar ou melhorar reparos Reexecução	Zhang *et al.* (2021)	S28
iPFlakies	Abordagens automáticas para gerar ou melhorar reparos Reexecução	Wang *et al.* (2022)	S18
FlakeRepro	Identificando as causas raiz para auxiliar no reparo Reexecução	Leesatapornwongsa *et al.* (2022)	S29
ODRepair	Abordagens automáticas para gerar ou melhorar reparos Reexecução	Li *et al.* (2022)	S30

Fonte: Elaborado pelo autor

A ferramenta MOKA, estudada no **S24**, é uma técnica que fornece geração automática de mocks de teste em aplicativos móveis. Basicamente, a ferramenta coleta dados que

são utilizados nas execuções dos testes e então gera simulações de testes a partir desses dados, substituindo as interações reais entre o aplicativo e seu ambiente, como câmera, microfone, *Global Positioning System* (GPS), etc. A cada reexecução, a ferramenta incrementa novos dados simulados, ajudando por exemplo que o teste seja executado com entradas inéditas, podendo reduzir ou indicar os *flaky tests* da suíte de testes. A ferramenta FaTB, estudada no **S26**, tem como objetivo aliviar o impacto negativo dos testes de espera assíncrona, fornecendo reparos automáticos para os desenvolvedores. Durante a reexecução de testes, a ferramenta identifica as chamadas de método do código que estão relacionadas com tempos limite ou esperas e em seguida calcula a frequência com o que o teste inconsistente pode falhar. Com base nessa frequência, o FaTB tentará reparar o teste com vários valores de tempo diferentes e então irá fornecer para os desenvolvedores o tempo mínimo que os mesmos devem utilizar.

A ferramenta DexFix, abordada no estudo **S28**, pretende fornecer automaticamente correções para *flaky tests* dependentes de uma implementação. Basicamente, a ferramenta foi criada para ser utilizada em conjunto com a ferramenta NonDex, explicada anteriormente na subseção 5.4.2.1. A entrada para o DexFix consiste em identificar o código-fonte do projeto e receber o teste que possui inconsistências, que foi detectado previamente pelo NonDex. Após esse processo, se aplica o DexFix para consertar o código-fonte e então o teste deve ser aprovado quando o NonDex for executado novamente.

O iFixFlakies é uma ferramenta que foi publicada em 2019 e abordada no estudo **S27**, e indica para os desenvolvedores correções de maneira automática para um teste dependente de ordem a partir das declarações de outros testes no conjunto de testes, permitindo que o teste em questão seja aprovado quando executado novamente. Após a execução dos testes de maneira isolada, os mesmo são classificados de acordo com o resultado obtido, sendo o primeiro os testes que são aprovados mesmo sendo executados de maneira isolada, mas falham quando executados com alguns outros testes e o segundo os testes que falham quando executados isoladamente, mas que passam quando executados com outros testes. Existe um terceiro tipo de teste, que são chamados de auxiliares, onde a lógica desses testes pode definir ou redefinir o estado do ambiente de uma forma que os testes dependente de ordem sejam aprovados. Logo, a ferramenta indica a correção para que seja implementada, resultando na aprovação dos testes mesmo em ordens que antes falharam. Como citado na subseção 5.4.2.2, a ferramenta iPFlakies, estudada no **S18**, também é considerada uma ferramenta que atua no contexto de reparação, visto que ela é composta também pelas funcionalidades do iFixFlakies, além das técnicas de deteção de teste

dependente de ordem do iDFlakies. A diferença é que o iPFlakies é aplicado em projetos Python, diferente do iFixFlakies que tem como foco projetos em Java.

A ferramenta ODRepair, estudada no **S30**, tem como objetivo reparar os testes dependente de ordem de maneira automática, mas, sem exigir que a lógica que define ou redefine o estado do ambiente seja implementado, diferente do que faz o iFixFlakies. Primeiramente, o ODRepair identifica o estado poluído do ambiente entre a execução dos testes e então gera automaticamente uma correção que realiza a chamada dos métodos que modificam o estado, redefinindo o ambiente para os testes que dependem de ordem.

A ferramenta RootFinder, abordada no estudo **S25**, tem como objetivo identificar as possíveis causas de manifestação de *flaky tests* e então auxiliar o desenvolvedor no reparo. Basicamente, a ferramenta analisa os logs de execuções de testes aprovados e reprovados de um mesmo teste para sugerir quais as chamadas de métodos possivelmente são responsáveis pela falha. A ferramenta tem como entrada o nome de um método que provavelmente seja a causa da falha e observa o comportamento do método em tempo de execução durante os testes que falham. A ferramenta FlakeRepro, estudada no **S29**, também tem como objetivo auxiliar o desenvolvedor no reparo a partir da identificação da causa raiz. O FlakeRepro busca analisar os logs com mensagens detalhadas sobre o local de erro e então reproduzir o teste exatamente da mesma maneira que ele falhou, sem depender da aleatoriedade, ajudando o desenvolvedor a consertar o erro dentro daquele cenário.

5.4.3 QP3 - Quais são as principais características das ferramentas de detecção de flaky tests e quais causas elas abordam?

Essa terceira questão de pesquisa tem o intuito de fornecer um catálogo com as características de mais alto nível das ferramentas de detecção, mitigação e reparo de *flaky tests* que foram coletadas após a análise dos artigos. Essas características serão explicadas, a fim de conceder um melhor entendimento sobre a lista das ferramentas e seus atributos. É possível visualizar na Tabela 4 o catálogo com todas as ferramentas coletadas e suas respectivas características.

Além dos objetivos e técnicas de cada ferramenta de detecção de *flaky tests* explicadas durante a Questão de Pesquisa 2 na subseção 5.4.2, existem outras informações pertinentes que o trabalho aqui proposto irá fornecer a fim de centralizar os dados sobre as ferramentas coletadas. Durante a análise das ferramentas, suas características foram organizadas nos seguintes tópicos

que compõem as colunas da Tabela 4:

- **Ferramenta**: A primeira coluna contém o nome de todas as ferramentas coletadas.
- **Linguagem de Implementação**: A segunda coluna fornece a lista com uma ou mais linguagens de programação na qual a ferramenta foi ou pode ser implementada. A coluna foi denominada como "Implementação".
- **Linguagem de Análise**: A terceira coluna fornece a linguagem de programação na qual a ferramenta foi implementada e analisada durante o estudo. A coluna foi denominada como "Análise".
- **Framework**: A quarta coluna fornece a lista com um ou mais *frameworks* de desenvolvimento de testes que podem ser utilizados com a ferramenta em questão.
- **Contexto**: A quinta coluna fornece o contexto na qual a ferramenta atua, seja detecção, mitigação ou reparo. As ferramentas de Testes Dependentes de Ordem (Detec. OD) foram abreviadas.
- **Interface**: A sexta coluna indica a forma que os desenvolvedores podem interagir com a ferramenta. As opções encontradas foram *Command-line Interface* (CLI), *pipeline* e *plugin*.
- **Documentação**: A sétima coluna indica se alguma documentação externa sobre a ferramenta está disponível online, seja um guia, repositório de aplicação, página do *plugin*, etc. O título Documentação (Doc) foi abreviado.
- **Site**: A oitava e última coluna fornece um ou mais links para os tipos de documentação da ferramenta, caso a mesma possua.

Caso alguma coluna da Tabela 4 não tenha sido descoberta durante a análise, será preenchida com *Unknown* (UNK), indicando que a mesma é desconhecida.

Das 30 ferramentas coletadas, um total de 21 ferramentas, ou seja, 70% das ferramentas foram analisadas em java e consequentemente os *frameworks* mais utilizados como apoio são JUnit[1] e Maven[2], visto que foram propriamente construídas para integração com java. Em seguida, as linguagens mais analisadas foram python, com 20%, C# com 6,6%. Apenas a ferramenta FlakyLoc não teve informação sobre sua linguagem de análise adquirida.

Um total de 73,3%, ou seja, 22 ferramentas possuem alguma documentação e ao menos 1 link para alguma documentação sobre a mesma. A grande maioria é do repositório de aplicação da ferramenta, onde algumas possuem um guia e outras não e os artigos em si

[1] https://junit.org
[2] https://maven.apache.org

funcionam como uma visão geral das ferramentas. Um total de 20 ferramentas atuam no contexto de detecção de *flaky tests*, onde 11 são focadas na detecção de testes dependentes de ordem. Apenas 3 ferramentas atuam no contexto de mitigação e 8 de reparo. Lembrando que a ferramenta iPFlakies é contada como uma ferramenta de detecção de testes dependentes de ordem e reparo.

No Quadro 10 é possível observar quais as causas de *flaky tests* foram abordadas pelas ferramentas. Apesar das ferramentas atuarem dentro de algum contexto, seja detecção, detecção de testes dependentes de ordem, mitigação ou reparo, nem todas as ferramentas abordam uma causa específica. As ferramentas de detecção, por exemplo, são focadas em detectar *flaky tests* de maneira geral, apenas buscando os padrões de inconsistências e realizando a detecção de fato. Como exceção, temos as ferramentas Shaker, do estudo S3 e NonDex, do estudo S5, que abordam Espera Assíncrona (*Async Wait*), Simultaneidade (*Concurrency*) e Dependência de Implementação (*Implementation Dependency*).

Todas as ferramentas que abordam *flaky tests* em testes dependentes de ordem, sugestivamente, abordam a Dependência da Ordem de Teste (*Test Order Dependency*), pois como citado anteriormente, essa causa possui fatores e dependências que podem estar ligadas diretamente aos *flaky tests*, como por exemplo, quando os testes partilham de um mesmo conjunto de dados ou de um estado, ocasionando no problema. No total, 14 das 30 ferramentas, ou seja, 46% das ferramentas, abordam a dependência da ordem de teste.

As ferramentas de mitigação e reparo tem um fator comum, que é trabalhar em cima da resolução do problema, seja diminuindo os efeitos, indicando ao desenvolvedor como resolver ou consertando o problema de maneira automática. E apesar de nem todas as ferramentas abordarem uma causa específica, somando todas as ferramentas de mitigação e reparo, é possível observar que todas as causas são abordadas por pelo menos uma ferramenta desses contextos.

Quadro 10: Causas de *flaky tests* abordadas pelas ferramentas

Causas	**Estudos**
Async Wait	S3, S16, S26
Concurrency	S3, S11, S16, S17, S22
Test Order Dependency	S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S22, S27, S30
External Data Dependency	S23, S24
Implementation Dependency	S5, S27, S28

Fonte: Elaborado pelo autor

Tabela 4: Catálogo das ferramentas de detecção de *flaky tests* e suas características

Ferramenta	Implementação	Análise	Framework	Contexto	Interface	Doc.	Site
MOKA	UNK	Java	JUnit Mockito	Reparo	UNK	Não	-
DeFlaker	Java	Java	JUnit TestNG	Detecção	CLI	Sim	Link 1
iDFlakies	Java	Java	JUnit Surefire Plugin	Detec. OD	CLI	Sim	Link 1 Link 2
FlakeFlagger	UNK	Java	UNK	Detecção	UNK	Sim	Link 1
RootFinder	C#	C#	MsTest Maven	Reparo	UNK	Sim	Link 1
FaTB	UNK	Python	UNK	Reparo	UNK	Sim	Link 1
Shaker	Java Python	Java	Maven Pytest	Detecção	CLI Pipeline	Sim	Link 1 Link 2
FLASH	Python	Python	UnitTest Pytest	Detecção	CLI	Sim	Link 1
PraDet	Java	Java	JUnit	Detec. OD	CLI	Sim	Link 1
iFixFlakies	Java	Java	JUnit Maven	Reparo	CLI Plugin	Sim	Link 1
NonDex	Java	Java	JUnit Maven	Detecção	CLI Plugin	Sim	Link 1 Link 2
FlakiMe	Java	Java	JUnit Maven	Mitigação	CLI Plugin	Sim	Link 1 Link 2
DexFix	Java	Java	JUnit Maven	Reparo	UNK	Não	-
ElectricTest	Java	Java	JUnit	Detec. OD	CLI	Não	-
DTDetector	Java	Java	JUnit	Detec. OD	CLI	Sim	Link 1
VmVm	Java	Java	JUnit Maven Ant	Mitigação	CLI	Sim	Link 1
PolDet	Java	Java	JUnit	Detec. OD	UNK	Não	-
FLAKE16	Python	Python	Pytest	Detec. OD	CLI	Sim	Link 1
FLAST	Todas	Python	IDE Pipeline	Detecção	UNK	Sim	Link 1
Flakify	Todas	Java	Pipeline	Detecção	UNK	Sim	Link 1
FlakeScanner	Scala	Java	JUnit	Detec. OD	UNK	Sim	Link 1
FlakyLoc	UNK	UNK	UNK	Mitigação	UNK	Não	-
OrcalePolish	Java	Java	JUnit	Detec. OD	CLI	Não	-
iPFlakies	Python	Python	Pytest	Detec. OD Reparo	CLI Plugin	Sim	Link 1 Link 2
PEELER	UNK	Java	UNK	Detecção	UNK	Sim	Link 1
TEDD	Java	Java	JUnit	Detec. OD	CLI	Sim	Link 1
FlakeRepro	C#	C#	UNK	Reparo	UNK	Não	-
IncIDFlakies	Java	Java	UNK	Detec. OD	UNK	Não	-
ODRepair	Java	Java	JUnit	Reparo	CLI	Sim	Link 1
MDFlaker	Python	Python	UNK	Detecção	UNK	Não	-

Fonte: Elaborado pelo autor.

5.5 Discussão

As 30 ferramentas de detecção de *flaky tests* extraídas, foram divididas de acordo com o seu contexto de atuação: 9 ferramentas de detecção, 11 ferramentas detecção de testes dependente de ordem, 3 ferramentas de mitigação e 8 ferramentas de reparo. Onde uma ferramenta pode atuar tanto na detecção de testes dependente de ordem como no reparo. Foi possível observar um crescimento contínuo de criação e publicação das ferramentas, onde em 2022 foram publicadas a maior quantidade delas.

Ao analisar as ferramentas, foi possível observar que um total de 83%, ou seja, 25 ferramentas utilizam a técnica de reexecução como apoio à detecção de flaky tests, isto é, além de outras técnicas. A reexecução está bastante presente nas ferramentas de detecção de *flaky tests*, sendo assim a técnica mais tradicional encontrada. Além disso, foi observado que um total de 70%, ou seja, 21 ferramentas foram analisadas em projetos com linguagens em java, subentendendo-se que a maioria dos *flaky tests* são manifestados em java. A maioria das ferramentas possuem alguma documentação que pode ou não demonstrar um guia de uso da mesma, porém, mesmo com os projetos disponibilizados online, nem sempre detalhes mais específicos são tornados públicos.

Mesmo atuando em diversos contextos, seja detecção, detecção de testes dependentes de ordem, mitigação ou reparo, nem todas as ferramentas deixaram claro abordar uma causa específica. Por exemplo, a maioria das ferramentas do contexto de detecção, são focadas em detectar *flaky tests* de maneira geral, apenas buscando os padrões de inconsistências e realizando a detecção de fato. Porém, foi possível constatar que um total de 14 das 30 ferramentas, ou seja, 46% das ferramentas, abordam a causa de dependência da ordem de teste e que somando todas as ferramentas de mitigação e reparo, todas as causas são abordadas por pelo menos uma ferramenta desses contextos.

Das 30 ferramentas coletadas durante este trabalho, 18 estavam contidas no trabalho de Parry *et al.* (2021), citado como um trabalho relacionado a este. Ou seja, 12 novas ferramentas foram coletadas e abordadas neste trabalho, reforçando a ideia de que mais estudos sobre ferramentas que abordam *flaky tests* estão sendo desenvolvidos. Das 12 ferramentas encontradas, 10 delas foram criadas ou publicadas após a publicação do estudo de Parry *et al.* (2021).

6 AMEAÇAS À VALIDADE

O trabalho aqui proposto possui algumas possíveis ameaças à validade. Primeiramente, existe a possibilidade de que não foram considerados todos os trabalhos existentes sobre ferramentas de detecção de *flaky tests*, o que pode resultar em um estudo com resultados incompletos. Portanto, uma maneira de mitigar esse problema, foi utilizado a técnica de Snowballing durante o período de execução do mapeamento sistemático, a fim de recuperar ou expandir possíveis referências não identificadas durante o processo. Além disso, o estudo não realizou busca de artigos na literatura cinza, mas para mitigar essa ação, foram selecionados os repositórios mais famosos dentro do contexto de Engenharia de Software e foi realizada uma revisão mais apurada nos critérios de inclusão e exclusão da etapa de planejamento.

Além disso, o mapeamento sistemático foi realizado apenas por um único pesquisador, onde o mesmo realizou todos os passos de planejamento, extração e análise dos dados, o que pode ser considerado um grande risco, observando o escopo do estudo. A fim de minimizar possíveis erros de seleção de artigos e análise de dados, o processo foi realizado com a premissa de que sempre que houvesse alguma dúvida sobre a aplicação de algum critério de pesquisa, o artigo em questão não seria eliminado, passando-o para a fase posterior do processo.

Apesar de terem sido encontradas 30 ferramentas disponíveis para a comunidade através de estudos, não significa que as mesmas estão disponíveis para implementação e utilização em projetos reais, para isso seria necessário um estudo focado na implementação das ferramentas para saber de fato quais estão disponíveis para implementação e uso em projetos reais.

7 CONCLUSÕES E TRABALHOS FUTUROS

Nesta pesquisa foi realizado um mapeamento sistemático a fim de analisar os estudos primários e fornecer todos os resultados encontrados para facilitar o entendimento da área de *flaky tests* e responder as 3 questões de pesquisa definidas durante o processo.

Durante o estudo foram extraídas 30 ferramentas de detecção de *flaky tests* que foram divididas de acordo com o seu contexto de atuação. Foi possível observar que houve um crescimento de publicações de ferramentas ao longo dos anos, sendo 2022 o ano com mais ferramentas publicadas.

Ao analisar as ferramentas, foi possível observar que a técnica de reexecução foi a mais utilizada como apoio à detecção de flaky tests. Além disso, foi observado que a maioria das ferramentas foram analisadas em projetos com linguagens em java, subentendendo-se que a maioria dos *flaky tests* são manifestados em java. Apesar de atuar em diferentes contextos, nem todas as ferramentas deixaram claro o tipo de *flaky test* detectado, porém, dentre os tipos coletados, foi visto que a maioria aborda o tipo de dependência da ordem de teste.

As questões de pesquisa visaram fornecer as ferramentas existentes na área de detecção *flaky tests*, quando cada ferramenta foi criada ou publicada, quais os objetivos e técnicas de cada ferramentas e suas características de mais alto nível de uma maneira centralizada. As descobertas desse trabalho entregam uma fonte mais centralizada das ferramentas disponíveis, sendo possível que um pesquisador, desenvolvedor ou profissional da área de Testes de Software possa selecionar uma ferramenta de acordo com seu estudo, necessidade profissional ou simplesmente para obter uma visão geral dos *flaky tests*.

Esse trabalho fornece uma visão geral dos *flaky tests* e das ferramentas de detecção disponíveis para a comunidade e suas características, além de fornecer insumo para estudos futuros. Como trabalhos futuros, propõe-se: (i) tentar implementar e avaliar as ferramentas encontradas para descobrir se elas são implementáveis em projetos reais; (ii) realizar uma comparação mais técnica entre ferramentas de um mesmo contexto, a fim de coletar dados mais específicos sobre elas e indicar quais as melhores ferramentas para utilização; (iii) realizar uma pesquisa focada em ferramentas de CI para detecção de *flaky tests* e compará-las com as ferramentas fornecidas neste trabalho; (iv) desenvolver uma ferramenta de detecção de *flaky tests* baseado em alguma ferramenta proposta neste trabalho e compará-las.

REFERÊNCIAS

AHMAD, A.; NETO, F. G. de O.; SHI, Z.; SANDAHL, K.; LEIFLER, O. A multi-factor approach for flaky test detection and automated root cause analysis. In: IEEE. **2021 28th Asia-Pacific Software Engineering Conference (APSEC)**. [*S. l.*], 2021. p. 338–348.

ALJEDAANI, W.; PERUMA, A.; ALJOHANI, A.; ALOTAIBI, M.; MKAOUER, M. W.; OUNI, A.; NEWMAN, C. D.; GHALLAB, A.; LUDI, S. Test smell detection tools: A systematic mapping study. In: **Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (EASE 2021), p. 170–180. ISBN 9781450390538. Disponível em: https://doi.org/10.1145/3463274.3463335. Acesso em: 21 jun. 2023.

ALSHAMMARI, A.; MORRIS, C.; HILTON, M.; BELL, J. Flakeflagger: Predicting flakiness without rerunning tests. In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [*S. l.*: *s. n.*], 2021. p. 1572–1584.

BARBÓN, J. M.; ALONSO, C. A.; BERTOLINO, A.; ÁLVAREZ, C. A. R.; GONZÁLEZ, P. J. T. *et al.* Flakyloc: flakiness localization for reliable test suites in web applications. **Journal of Web Engineering, 2**, River Publishers, 2020.

BARN, B.; BARAT, S.; CLARK, T. Conducting systematic literature reviews and systematic mapping studies. In: **Proceedings of the 10th Innovations in Software Engineering Conference**. New York, NY, USA: Association for Computing Machinery, 2017. (ISEC ’17), p. 212–213. ISBN 9781450348560. Disponível em: https://doi.org/10.1145/3021460.3021489. Acesso em: 21 jun. 2023.

BELL, J.; KAISER, G. Unit test virtualization with vmvm. In: **Proceedings of the 36th International Conference on Software Engineering**. [*S. l.*: *s. n.*], 2014. p. 550–561.

BELL, J.; KAISER, G.; MELSKI, E.; DATTATREYA, M. Efficient dependency detection for safe java test acceleration. In: **Proceedings of the 2015 10th joint meeting on foundations of software engineering**. [*S. l.*: *s. n.*], 2015. p. 770–781.

BELL, J.; LEGUNSEN, O.; HILTON, M.; ELOUSSI, L.; YUNG, T.; MARINOV, D. Deflaker: Automatically detecting flaky tests. In: **2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)**. [*S. l.*: *s. n.*], 2018. p. 433–444.

BIAGIOLA, M.; STOCCO, A.; MESBAH, A.; RICCA, F.; TONELLA, P. Web test dependency detection. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2019. p. 154–164.

COHN, M. **Succeeding with Agile**: Software development using scrum. Addison-Wesley, 2010. (A Mike Cohen signature book). ISBN 9780321579362. Disponível em: https://books.google.com.br/books?id=IdT6AgAAQBAJ. Acesso em: 21 jun. 2023.

CORDEIRO, M.; SILVA, D.; TEIXEIRA, L.; MIRANDA, B.; D’AMORIM, M. Shaker: a tool for detecting more flaky tests faster. In: IEEE. **2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [*S. l.*], 2021. p. 1281–1285.

CORDY, M.; RWEMALIKA, R.; FRANCI, A.; PAPADAKIS, M.; HARMAN, M. Flakime: laboratory-controlled test flakiness impact assessment. In: **Proceedings of the 44th International Conference on Software Engineering**. [*S. l.*: *s. n.*], 2022. p. 982–994.

DONG, Z.; TIWARI, A.; YU, X. L.; ROYCHOUDHURY, A. Flaky test detection in android via event order exploration. In: **Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2021. p. 367–378.

DUTTA, S.; SHI, A.; CHOUDHARY, R.; ZHANG, Z.; JAIN, A.; MISAILOVIC, S. Detecting flaky tests in probabilistic and machine learning applications. In: **Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis**. [*S. l.*: *s. n.*], 2020. p. 211–224.

ECK, M.; PALOMBA, F.; CASTELLUCCIO, M.; BACCHELLI, A. Understanding flaky tests: The developer's perspective. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 830–840. ISBN 9781450355728. Disponível em: https://doi.org/10.1145/3338906.3338945. Acesso em: 21 jun. 2023.

FATIMA, S.; GHALEB, T. A.; BRIAND, L. Flakify: A black-box, language model-based predictor for flaky tests. **IEEE Transactions on Software Engineering**, IEEE, 2022.

FAZZINI, M.; GORLA, A.; ORSO, A. A framework for automated test mocking of mobile apps. In: **Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering**. [*S. l.*: *s. n.*], 2020. p. 1204–1208.

GAMBI, A.; BELL, J.; ZELLER, A. Practical test dependency detection. In: IEEE. **2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*], 2018. p. 1–11.

GRUBER, M.; FRASER, G. A survey on how test flakiness affects developers and what support they need to address it. In: **2022 IEEE Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*: *s. n.*], 2022. p. 82–92.

GRUBER, M.; LUKASCZYK, S.; KROIß, F.; FRASER, G. An empirical study of flaky tests in python. In: **2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*: *s. n.*], 2021. p. 148–158.

GYORI, A.; LAMBETH, B.; SHI, A.; LEGUNSEN, O.; MARINOV, D. Nondex: A tool for detecting and debugging wrong assumptions on java api specifications. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2016. p. 993–997.

GYORI, A.; SHI, A.; HARIRI, F.; MARINOV, D. Reliable testing: Detecting state-polluting tests to prevent test dependency. In: **Proceedings of the 2015 international symposium on software testing and analysis**. [*S. l.*: *s. n.*], 2015. p. 223–233.

HABCHI, S.; HABEN, G.; PAPADAKIS, M.; CORDY, M.; TRAON, Y. L. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In: **2022 IEEE Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*: *s. n.*], 2022. p. 244–255.

HUO, C.; CLAUSE, J. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 621–631. ISBN 9781450330565. Disponível em: https://doi.org/10.1145/2635868.2635917. Acesso em: 21 jun. 2023.

ISHA; SHARMA, A.; REVATHI, M. Automated api testing. In: **2018 3rd International Conference on Inventive Computation Technologies (ICICT)**. [*S. l.*: *s. n.*], 2018. p. 788–791.

JALALI, S.; WOHLIN, C. Systematic literature studies: database searches vs. backward snowballing. In: **Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement**. [*S. l.*: *s. n.*], 2012. p. 29–38.

KEELE, S. *et al.* **Guidelines for performing systematic literature reviews in software engineering**. [*S. l.*]: Technical report, ver. 2.3 ebse technical report. ebse, 2007.

LABUSCHAGNE, A.; INOZEMTSEVA, L.; HOLMES, R. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 821–830. ISBN 9781450351058. Disponível em: https://doi.org/10.1145/3106237.3106288. Acesso em: 21 jun. 2023.

LAM, W.; GODEFROID, P.; NATH, S.; SANTHIAR, A.; THUMMALAPENTA, S. Root causing flaky tests in a large-scale industrial setting. In: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2019. (ISSTA 2019), p. 101–111. ISBN 9781450362245. Disponível em: https://doi.org/10.1145/3293882.3330570. Acesso em: 21 jun. 2023.

LAM, W.; MUsLU, K.; SAJNANI, H.; THUMMALAPENTA, S. A study on the lifecycle of flaky tests. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE ’20), p. 1471–1482. ISBN 9781450371216. Disponível em: https://doi.org/10.1145/3377811.3381749. Acesso em: 21 jun. 2023.

LAM, W.; OEI, R.; SHI, A.; MARINOV, D.; XIE, T. idflakies: A framework for detecting and partially classifying flaky tests. In: IEEE. **2019 12th ieee conference on software testing, validation and verification (icst)**. [*S. l.*], 2019. p. 312–322.

LAM, W.; WINTER, S.; WEI, A.; XIE, T.; MARINOV, D.; BELL, J. A large-scale longitudinal study of flaky tests. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. OOPSLA, nov 2020. Disponível em: https://doi.org/10.1145/3428270. Acesso em: 21 jun. 2023.

LEESATAPORNWONGSA, T.; REN, X.; NATH, S. Flakerepro: automated and efficient reproduction of concurrency-related flaky tests. In: **Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2022. p. 1509–1520.

LI, C.; SHI, A. Evolution-aware detection of order-dependent flaky tests. In: **Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis**. [*S. l.*: *s. n.*], 2022. p. 114–125.

LI, C.; ZHU, C.; WANG, W.; SHI, A. Repairing order-dependent flaky tests via test generation. In: **Proceedings of the 44th International Conference on Software Engineering**. [*S. l.*: *s. n.*], 2022. p. 1881–1892.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. An empirical analysis of flaky tests. In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 643–653. ISBN 9781450330565. Disponível em: https://doi.org/10.1145/2635868.2635920. Acesso em: 21 jun. 2023.

MYERS, G.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. Wiley, 2011. (ITPro collection). ISBN 9781118133156. Disponível em: https://books.google.com.br/books?id=GjyEFPkMCwcC. Acesso em: 21 jun. 2023.

PARRY, O.; KAPFHAMMER, G. M.; HILTON, M.; MCMINN, P. A survey of flaky tests. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 1, oct 2021. ISSN 1049-331X. Disponível em: https://doi.org/10.1145/3476105. Acesso em: 21 jun. 2023.

PARRY, O.; KAPFHAMMER, G. M.; HILTON, M.; MCMINN, P. Evaluating features for machine learning detection of order-and non-order-dependent flaky tests. In: IEEE. **2022 IEEE Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*], 2022. p. 93–104.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. **Information and software technology**, Elsevier, v. 64, p. 1–18, 2015.

PINTO, G.; MIRANDA, B.; DISSANAYAKE, S.; D'AMORIM, M.; TREUDE, C.; BERTOLINO, A. What is the vocabulary of flaky tests? In: **Proceedings of the 17th International Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2020. (MSR '20), p. 492–502. ISBN 9781450375177. Disponível em: https://doi.org/10.1145/3379597.3387482. Acesso em: 21 jun. 2023.

PONTILLO, V. Static test flakiness prediction. In: **Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings**. [*S. l.*: *s. n.*], 2022. p. 325–327.

QIN, Y.; WANG, S.; LIU, K.; LIN, B.; WU, H.; LI, L.; MAO, X.; BISSYANDÉ, T. F. Peeler: Learning to effectively predict flakiness without running tests. In: IEEE. **2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S. l.*], 2022. p. 257–268.

SHI, A.; GYORI, A.; LEGUNSEN, O.; MARINOV, D. Detecting assumptions on deterministic implementations of non-deterministic specifications. In: IEEE. **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*], 2016. p. 80–90.

SHI, A.; LAM, W.; OEI, R.; XIE, T.; MARINOV, D. ifixflakies: A framework for automatically fixing order-dependent flaky tests. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2019. p. 545–555.

SILVA, D.; TEIXEIRA, L.; D'AMORIM, M. Shake it! detecting flaky tests caused by concurrency with shaker. In: **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S. l.*: *s. n.*], 2020. p. 301–311.

SOMMERVILLE, I. **Engenharia de software**. Pearson Prentice Hall, 2011. ISBN 9788579361081. Disponível em: https://books.google.com.br/books?id=H4u5ygAACAAJ. Acesso em: 21 jun. 2023.

VERDECCHIA, R.; CRUCIANI, E.; MIRANDA, B.; BERTOLINO, A. Know you neighbor: Fast static prediction of test flakiness. **IEEE Access**, IEEE, v. 9, p. 76119–76134, 2021.

WANG, R.; CHEN, Y.; LAM, W. ipflakies: a framework for detecting and fixing python order-dependent flaky tests. In: **Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings**. [*S. l.*: *s. n.*], 2022. p. 120–124.

ZHANG, P.; JIANG, Y.; WEI, A.; STODDEN, V.; MARINOV, D.; SHI, A. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In: IEEE. **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [*S. l.*], 2021. p. 50–61.

ZHANG, S.; JALALI, D.; WUTTKE, J.; MUsLU, K.; LAM, W.; ERNST, M. D.; NOTKIN, D. Empirically revisiting the test independence assumption. In: **Proceedings of the 2014 International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2014. (ISSTA 2014), p. 385–396. ISBN 9781450326452. Disponível em: https://doi.org/10.1145/2610384.2610404. Acesso em: 21 jun. 2023.

ZHANG, S.; JALALI, D.; WUTTKE, J.; MUŞLU, K.; LAM, W.; ERNST, M. D.; NOTKIN, D. Empirically revisiting the test independence assumption. In: **Proceedings of the 2014 International Symposium on Software Testing and Analysis**. [*S. l.*: *s. n.*], 2014. p. 385–396.

ZOLFAGHARI, B.; PARIZI, R. M.; SRIVASTAVA, G.; HAILEMARIAM, Y. Root causing, detecting, and fixing flaky tests: state of the art and future roadmap. **Software: Practice and Experience**, Wiley Online Library, v. 51, n. 5, p. 851–867, 2021.

APÊNDICE A – CONJUNTO FINAL DOS ARTIGOS SELECIONADOS NO MAPEAMENTO

Tabela 5: Conjunto final dos estudos selecionados

ID	Título	Referência
S1	DeFlaker: Automatically Detecting Flaky Tests	Bell *et al.* (2018)
S2	FlakeFlagger: Predicting Flakiness Without Rerunning Tests	Alshammari *et al.* (2021)
S3	Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker	Silva *et al.* (2020)
S4	Detecting Flaky Tests in Probabilistic and Machine Learning Applications	Dutta *et al.* (2020)
S5	Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications	Shi *et al.* (2016)
S6	Know You Neighbor: Fast Static Prediction of Test Flakiness	Verdecchia *et al.* (2021)
S7	Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests	Fatima *et al.* (2022)
S8	Peeler: Learning to Effectively Predict Flakiness without Running Tests	Qin *et al.* (2022)
S9	A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis	Ahmad *et al.* (2021)
S10	iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests	Lam *et al.* (2019)
S11	Practical Test Dependency Detection	Gambi *et al.* (2018)
S12	Efficient dependency detection for safe Java test acceleration	Bell *et al.* (2015)
S13	Empirically revisiting the test independence assumption	Zhang *et al.* (2014b)
S14	Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency	Gyori *et al.* (2015)
S15	Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests	Parry *et al.* (2022)
S16	Flaky Test Detection in Android via Event Order Exploration	Dong *et al.* (2021)
S17	Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests	Huo e Clause (2014)
S18	iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests	Wang *et al.* (2022)
S19	Web Test Dependency Detection	Biagiola *et al.* (2019)
S20	Evolution-Aware Detection of Order-Dependent Flaky Tests	Li e Shi (2022)
S21	FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment	Cordy *et al.* (2022)
S22	Unit Test Virtualization with VMVM	Bell e Kaiser (2014)
S23	FlakyLoc: Flakiness Localization for Reliable Test Suites in Web Applications	Barbón *et al.* (2020)
S24	A Framework for Automated Test Mocking of Mobile Apps	Fazzini *et al.* (2020)
S25	Root causing flaky tests in a large-scale industrial setting	Lam *et al.* (2019)
S26	A Study on the Lifecycle of Flaky Tests	Lam *et al.* (2020)
S27	IFixFlakies: A framework for automatically fixing order-dependent flaky tests	Shi *et al.* (2019)
S28	Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications	Zhang *et al.* (2021)
S29	FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests	Leesatapornwongsa *et al.* (2022)
S30	Repairing order-dependent flaky tests via test generation	Li *et al.* (2022)
S31	Shaker: A Tool for Detecting More Flaky Tests Faster	Cordeiro *et al.* (2021)
S32	NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications	Gyori *et al.* (2016)
S33	A Large-Scale Longitudinal Study of Flaky Tests	Lam *et al.* (2020)
S34	A Survey of Flaky Tests	Parry *et al.* (2021)
S35	Root causing, detecting, and fixing flaky tests: state of the art and future roadmap	Zolfaghari *et al.* (2021)
S36	Static test flakiness prediction	Pontillo (2022)
S37	A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests	Habchi *et al.* (2022)

Fonte: Elaborado pela autor.