UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA - CCET
DEPARTAMENTO DE COMPUTAÇÃO - DC

Gabriel Mari Tararam

Um Estudo sobre Testes Flaky em Projetos Python

SÃO CARLOS - SP
2024

Gabriel Mari Tararam

Um Estudo sobre Testes Flaky em Projetos Python

Trabalho de conclusão de curso apresentado ao Departamento de Computação da Universidade Federal de São Carlos, para obtenção do título de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. André Takeshi Endo

SÃO CARLOS - SP
2024

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia - CCET
Departamento de Computação

Comissão avaliadora

Membros da comissão examinadora que avaliou e aprovou a Defesa do Trabalho de Conclusão de Curso do candidato Gabriel Mari Tararam, realizada em 06/09/2024

Prof. Dr. André Takeshi Endo
Instituição: Universidade Federal de São Carlos

Prof. Dr. Auri Marcelo Rizzo Vincenzi
Instituição: Universidade Federal de São Carlos

Prof. Dr. Fabiano Cutigi Ferrari
Instituição: Universidade Federal de São Carlos

Dedicatória

Dedico este trabalho aos meus pais, Marcos e Sílvia, minha avó Idalina, meu avô Daniel e todos os meus amigos e familiares que me apoiaram nessa difícil e longa caminhada.

AGRADECIMENTO

Gostaria de expressar minha profunda gratidão aos meus avós, Idalina e Daniel, que sempre acreditaram em mim e me incentivaram a seguir em frente, mesmo nos momentos mais desafiadores.

Agradeço também à minha querida gata Misty, cuja companhia constante e afetuosa iluminou meus dias bons e me confortou nos dias ruins, tornando cada momento mais leve.

Finalmente, expresso minha profunda gratidão ao meu orientador de TCC, Prof. Dr. André Takeshi Endo, por ser extremamente prestativo e sempre se voluntariar a me auxiliar, proporcionando a orientação e o suporte necessários para a conclusão deste trabalho.

Não sou nada. Nunca serei nada. Não posso querer ser nada. À parte isso, tenho em mim todos os sonhos do mundo. - Álvaro de Campos

RESUMO

A compreensão do comportamento dos testes automatizados é crucial para o aprimoramento das práticas de desenvolvimento de software. Este estudo foca na investigação da incidência de testes *flaky* em projetos Python. A pesquisa envolveu a análise de um conjunto diversificado de projetos Python para responder às questões de pesquisa propostas. Os resultados indicaram que 19,44% dos projetos analisados apresentaram testes *flaky*, uma taxa significativamente maior do que os 4,5% relatados por Gruber et al. (2021). A análise mostrou que a maioria (85,1%) dos testes *flaky* identificados dependiam da ordem de execução, enfatizando a necessidade de um cuidado com a independência entre os testes para minimizar a chance de ocorrer *flakiness*. Testes *flaky* não dependentes de ordem foram menos frequentes, mas mostraram inconsistências principalmente devido à dependência de APIs externas e limite de tempo de execução de bibliotecas. Esses achados sugerem que a estabilidade dos testes pode ser comprometida por fatores externos, como a disponibilidade de serviços externos e a variabilidade nas respostas dessas APIs.

Palavras-chave: Testes *flaky*, Testes automatizados, Python, Confiabilidade de testes.

ABSTRACT

Understanding the behavior of automated tests is crucial for improving software development practices. This study focuses on investigating the incidence of *flaky* tests in Python projects. The research involved analyzing a diverse set of Python projects to address the proposed research questions. The results indicated that 19.44% of the analyzed projects exhibited *flaky* tests, a significantly higher rate than the 4.5% reported by Gruber et al. (2021). The analysis showed that the majority (85.1%) of the identified *flaky* tests depended on the order of execution, emphasizing the need to ensure independence between tests to minimize the chance of *flakiness*. Non-order-dependent *flaky* tests were less frequent but exhibited inconsistencies mainly due to dependencies on external APIs and execution time limits of libraries. These findings suggest that test stability can be compromised by external factors, such as the availability of external services and the variability in responses from these APIs.

Future work could be carried out to expand the FlaPy tool to enable the installation and verification of a broader range of projects.

Keyword: *flaky* tests, Automated testing, Python, Software stability, Test reliability.

Lista de Figuras

Lista de Tabelas

Conteúdo

1 INTRODUÇÃO

Os testes automatizados são de fundamental importância para a avaliação da qualidade e confiabilidade do software, envolvendo a utilização de técnicas automatizadas para a identificação de erros antes de entrarem em produção. De acordo com Kumar e Mishra (2016), essa prática adquire uma relevância incontestável no ciclo de vida do software.

Conforme citado por Garousi e Felderer (2016), em diversos projetos é notável que os testes automatizados proporcionam diversos benefícios, como, por exemplo, a execução de testes repetíveis, previsíveis e eficientes. Contudo, é um processo de desenvolvimento tedioso, com possíveis erros de implementação, além de demandar um investimento significativo. Mesmo não considerando esse custo inicial de implementação, ainda é necessária a manutenção dos testes automatizados, que devem coevoluir com o código de produção. Por conta disso, são necessárias técnicas, ferramentas e métodos que abranjam todo o ciclo de vida do código de teste.

Um exemplo citado por Garousi e Felderer (2016) é que para o software Jedit, foram executados os testes automatizados desenvolvidos para a versão 3.2.2 do software, nas versões 4.0 e 4.1 do mesmo software, obtendo-se que dos 71 casos de teste, 19 (26%) e 27 (38%) passaram nas duas versões, enquanto os 52 (74%) e 44 (62%) restantes falharam, evidenciando a necessidade da manutenção e atualização dos testes. Nesses casos o custo de manutenção é maior, pois gera a necessidade de os desenvolvedores revisitarem os testes toda vez que modificam uma funcionalidade no sistema, o que é ainda mais crítico em sistemas com entregas altamente incrementais.

Dentre os problemas que surgiram com os testes automatizados, existem os testes *flaky*; como descrito por Habchi et al. (2022), um mesmo código de teste pode falhar ou passar em diferentes execuções, gerando um resultado não determinístico, o que confunde os desenvolvedores na interpretação do resultado do teste. Segundo Listfield (2017), esse se tornou um problema relevante, a partir do momento que ignorar estes testes, poderia ser ignorar também um bug em produção, não tendo o código de fato coberto por aquele teste *flaky*.

Listfield (2017) afirma possuir cerca de 4,2 milhões de testes executados no sistema de CI da empresa Google, sendo aproximadamente 63 mil testes *flaky* identificados semanalmente. Mesmo sendo somente 2% do total de testes, isso ainda é relevante e gera um problema para os desenvolvedores envolvidos. Segundo Habchi et al. (2022), além de implicar em um tempo maior de desenvolvimento e interromper a integração contínua, os testes *flaky*, afetam as boas práticas de teste e pioram a qualidade do sistema em si.

Os testes *flaky* se tornam um problema relevante a partir do momento em que, conforme evidenciado por uma pesquisa recente analisada por Eck et al. (2019) envolvendo desenvolvedores da indústria, revelaram-se a prevalência e a proeminência dos testes *flaky* no processo de testes automatizados. Os resultados apontam que a instabilidade nos testes

é uma questão recorrente, afetando 20% dos participantes mensalmente, 24% semanalmente e 15% diariamente. Essa frequência de ocorrência destaca a natureza persistente do problema, evidenciando sua presença constante no ciclo de desenvolvimento de software. Além disso, a pesquisa revela a gravidade do impacto, com 91% dos desenvolvedores alegando enfrentar testes *flaky* pelo menos algumas vezes por ano. Entre esses, 56% descreveram o problema como moderado, enquanto 23% consideraram-no sério.

Diante do impacto considerável dos testes *flaky*, diversas abordagens são propostas com o intuito de mitigar, visualizar, quantificar e isolar, ou até mesmo corrigir, os testes *flaky* (Gruber et al., 2023). Contudo, todas essas estratégias enfrentam um desafio comum, que é a dependência de técnicas eficazes para a detecção precisa dos testes *flaky*. Conforme ressaltado por Gruber et al. (2023), a inspeção manual torna-se impraticável em projetos maiores, nos quais ocorrem centenas de alterações diárias, resultando na execução de milhares de testes de regressão.

Para a linguagem Python, o estudo de Gruber et al. (2021), visando ampliar a análise de *flakiness* além da linguagem Java, desenvolveu o software denominado FlaPy, utilizado para analisar empiricamente a prevalência, causas e grau de instabilidade em projetos Python. Nesse estudo foram analisados 22.352 projetos Python contidos no indexador de pacotes PyPI, e 876.186 casos de teste. Os resultados identificaram que o problema é tão predominante em Python quanto em Java, mas com razões distintas. Em Python, a dependência da ordem é mais dominante (59% dos 7.571 casos), seguida por problemas de infraestrutura (28%) e o uso de rede e APIs de aleatoriedade (13%). A ferramenta FlaPy foi desenvolvida em Python e utiliza a biblioteca PyTest como base, para identificação de testes *flaky*. Essa ferramenta recebe como entrada o diretório ou URL do seu repositório, e executa os testes em uma ordem constante ou aleatória, além disso é possível definir o número de vezes que os testes são executados e o nível de granularidade (classe, módulo, pacote, etc.) em que os testes são "embaralhados". A metodologia utilizada se baseia em utilizar uma grande base de dados, e o número de execuções necessárias para identificar um teste *flaky*.

Inicialmente existem duas estratégias para criar um conjunto de dados contendo testes *flaky*. A primeira estratégia é procurar por commits que corrigem testes *flaky*, ou por issues reportando testes *flaky*. A segunda estratégia consiste em executar os testes um determinado número de vezes e verificar se o resultado mudou, ou seja, se um teste falhou e passou em diferentes execuções.

No caso da ferramenta FlaPy foi utilizado como base o segundo método com 400 re-execuções, um número maior que estudos anteriores, mas com o objetivo também de identificar quantas execuções são necessários para se identificar um teste *flaky*.

Tendo como foco o problema de identificação de *flakiness* em projetos Python, este trabalho se propõe a aplicar e extrair dados de *flakiness* utilizando a ferramenta FlaPy em projetos distintos dos estudados anteriormente por Gruber et al. (2021).

1.1 Objetivos

O objetivo deste trabalho foi a aplicação da ferramenta FlaPy, proveniente do estudo de Gruber et al. (2021), em um conjunto de projetos com características distintas dos originalmente utilizados. Para isso, foram selecionados projetos de repositórios/indexadores distintos em relação aos utilizados no estudo de forma a verificar se o padrão observado se repete, ou apresenta características diferentes. Como repositório distinto em relação ao PyPi, foi escolhido o conjunto de projetos Awesome Python. Este conjunto de projetos passa por uma curadoria dos mais de 400 contribuidores, sendo necessária a aprovação de mais de 20 contribuidores para um novo projeto ser incluído. Este projeto contém mais de 24,6 mil *forks* na plataforma GitHub, e 196 mil *stars*[1], o que aumenta a credibilidade deste repositório.

1.2 Estrutura do Trabalho

Este trabalho está dividido da seguinte forma: o Capítulo 2 discute a revisão bibliográfica, abordando os testes automatizados, testes *flaky* e trabalhos relacionados; o Capítulo 3 detalha a metodologia utilizada para a condução do estudo; no Capítulo 4 são apresentados e analisados os resultados obtidos; e, finalmente, no Capítulo 5, são apresentadas as conclusões e sugestões para trabalhos futuros.

[1]Forma dos usuários da plataforma indicarem que aprovam determinado projeto.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo, são discutidos os principais conceitos e abordagens relacionados aos testes automatizados, com ênfase nos testes *flaky* em projetos Python. A organização do capítulo segue a seguinte estrutura: na Seção 2.1, são explorados os conceitos fundamentais de testes automatizados; na Seção 2.2, o foco é nos testes *flaky*, abordando suas características e implicações; e a Seção 2.3 apresenta uma revisão dos principais trabalhos relacionados ao tema.

2.1 Testes Automatizados

Atualmente, testes automatizados são utilizados para verificar a funcionalidade de softwares comerciais ou de código aberto. Isso é especialmente relevante para projetos de software que evoluem a partir de muitas versões, já que os testes automatizados são mais eficazes em testes de regressão e repetitivos (Garousi e Felderer, 2016).

Segundo ISO/IEC/IEEE... (2022) o principal propósito do teste é proporcionar dados sobre a qualidade do item avaliado e os riscos que ainda possam existir, detectando falhas antes de sua disponibilização para uso. Esses dados auxiliam na melhoria do item, sustentam as decisões gerenciais com base em risco e qualidade, além de otimizar os processos internos ao identificar onde as falhas ocorrem ou permanecem ocultas.

Uma das estratégias propostas para implementação de testes automatizados, é a descrita por Pressman e Maxim (2010), em que o processo começa com testes unitários, que se concentram na verificação de cada unidade do software, como componentes ou classes, conforme implementadas no código-fonte. Em seguida, são realizados os testes de integração, cujo foco está na arquitetura do software e no seu design. Em seguida, ocorrem os testes de validação, nos quais os requisitos definidos anteriormente são verificados em relação ao software desenvolvido. Por fim, realizam-se os testes de sistema, nos quais o software, juntamente com outros elementos do sistema, é testado como um todo.

Alguns números que destacam a importância dos testes automatizados são trazidos por Garousi e Felderer (2016). Um exemplo notável é fornecido pelos engenheiros de teste da Microsoft, que relataram ter escrito mais de um milhão de casos de teste automatizados para o Microsoft Office 2007, destacando a escala e a relevância desses testes na indústria. A complexidade e a extensão das suítes de testes podem exceder as do próprio código de produção. No caso da versão 2.1 do sistema operacional Android, a base de código de testes JUnit compreendia 17,1% do total de código Java.

A biblioteca requests (REITZ, 2024) é amplamente utilizada em projetos Python. Como exemplo de teste automatizado, o trecho de código da Figura 1 na linha 2 realiza uma requisição do tipo get para o endereço obtido pela biblioteca Httpbin e com o parametro redirect (Atua como um serviço de requisição e resposta HTTP para testes e

depuração). Na linha 3 é avaliado se o status_code é 200, ou seja, se a requisição foi realizada com sucesso, nas linhas 4 e 5 verificam-se o status code é 302 (indica redirecionamento) e o atributo is_redirect da primeira requisição intermediária (o atributo history da resposta, é uma lista de respostas de cada requisição intermediaria feita até chegar ao destino final) é o esperado, ou seja, que a requisição foi feita por meio de redirecionamento.

Figura 1: Função de teste `test_HTTP_302_ALLOW_REDIRECT_GET` da biblioteca requests

```
1 def test_HTTP_302_ALLOW_REDIRECT_GET(self, httpbin):
2     r = requests.get(httpbin("redirect", "1"))
3     assert r.status_code == 200
4     assert r.history[0].status_code == 302
5     assert r.history[0].is_redirect
```

Fonte: Biblioteca requests (REITZ, 2024)

2.2 Testes *flaky*

Testes *flaky* são aqueles que apresentam resultados inconsistentes, passando ou falhando sem nenhuma modificação no código. Isso dificulta a identificação de problemas novos no código, pois não é claro se uma falha é um indicativo real de um problema ou apenas uma manifestação da instabilidade do próprio teste (Listfield, 2017).

Um exemplo ilustrativo é o da Figura 2 em que o teste não apresenta resultados consistentes. Se executado diversas vezes, tende a ter 50% dos resultados aprovados e 50% reprovados. Isso pode ser observado na linha 2, em que é utilizada a função random nativa do Python para obter aleatoriamente ou o valor True, ou False, com probabilidades iguais para ambos. Na linha 7 se obtém ou o valor True ou o valor False, via chamada da função `get_random_boolean()`, logo em seguida na linha 8 é verificado se o resultado obtido é True, se for, irá resultar em um teste aprovado, se não for, o teste irá falhar.

Figura 2: Exemplo fictício de teste *flaky*

```
1 def get_random_boolean():
2     return random.choice([True, False])
3
4 class TestFlakyRandomFunction(unittest.TestCase):
5
6     def test_random_number(self):
7         result = get_random_boolean()
8         self.assertEqual(result, True)
```

Fonte: Autoria própria

Listfield (2017) ainda ressalta alguns números que mostram a relevância dos teste *flaky* dentro da empresa Google, que possuem cerca de 4,2 milhões de testes executados em seu

sistema de integração contínua, com aproximadamente 63 mil testes *flaky* identificados semanalmente. Apesar de ser apenas 2%, gera um impacto relevante para os engenheiros responsáveis pelo desenvolvimento do software. O autor afirma que em geral, quanto maior o teste (medido pelo tamanho do binário, uso de RAM ou número de bibliotecas construídas), mais provável é que ele seja instável. Mostrando que o impacto em grandes projetos pode ser maior ainda.

Por conta da relevância deste problema, diversos trabalhos são realizados voltados para mitigar, visualizar, quantificar, isolar e corrigir os testes *flaky*, porém isso depende de técnicas que identifiquem de forma efetiva os teste *flaky*.

O estudo de Gruber et al. (2021) foi feito com essa proposta, analisando a prevalência, causas e graus de *flakiness* em projetos Python, uma das linguagens mais populares atualmente. Os pesquisadores examinaram 22.352 projetos de código aberto do repositório PyPI, contendo 876.186 casos de teste. Destes, identificaram 7.571 testes *flaky*, representando 0,86% dos testes analisados. As principais causas de *flakiness* foram a dependência de ordem dos testes (59%) e problemas de infraestrutura de teste (28%). Além disso, APIs de rede e aleatoriedade foram responsáveis por 13% dos casos.

Para detectar e mitigar testes *flaky*, os pesquisadores sugerem que é necessário realizar um número significativo de reexecuções dos testes. Para uma confiança de 95% de que um teste não é *flaky*, são necessárias, em média, 170 reexecuções. Este estudo não apenas destacou a prevalência de testes *flaky* em Python, mas também forneceu *insights* sobre as principais causas e sugeriu estratégias para sua detecção e mitigação, contribuindo assim para a melhoria da qualidade dos testes de software (Gruber et al., 2021).

2.3 Trabalhos Relacionados

Gruber et al. (2021) realizaram um estudo empírico com o objetivo de investigar a prevalência, causas e graus de *flakiness* em projetos Python. Para isso, analisaram 22.352 projetos de código aberto do índice PyPI, contendo 876.186 casos de teste. A metodologia incluiu a execução de 400 repetições de cada teste para identificar aqueles que apresentavam comportamento instável. Destes 876.186 casos de teste, foram identificados 7.571 testes *flaky*, representando 0,86% dos testes analisados. As principais causas de *flakiness* foram a dependência de ordem dos testes (59%) e problemas de infraestrutura de teste (28%). Além disso, APIs de rede e aleatoriedade foram responsáveis por 13% dos casos.

Para detectar e mitigar testes *flaky*, os pesquisadores sugerem que é necessário realizar um número significativo de reexecuções dos testes. Para uma confiança de 95% de que um teste não é *flaky*, são necessárias, em média, 170 reexecuções. Este estudo não apenas destacou a prevalência de testes *flaky* em Python, mas também forneceu *insights* sobre as principais causas e sugeriu estratégias para sua detecção e mitigação, contribuindo assim

para a melhoria da qualidade dos testes de software (Gruber et al., 2021).

Parry et al. (2021) realizaram uma pesquisa abrangente que analisou 76 artigos sobre teste *flaky*, dividindo a análise nas seguintes partes: causas, custos, consequências, estratégias de detecção e métodos de mitigação e reparo. Eck et al. (2019) identificaram que 59% dos desenvolvedores lidam com teste *flaky* regularmente, além disso, com a dependência de ordem sendo a principal causa, responsável por até 59% dos casos, segundo Gruber et al. (2021). As estratégias de mitigação incluem técnicas de isolamento de testes e métodos robustos de *setup* e *teardown*. As consequências dos teste *flaky* incluem a perda de tempo e recursos dos desenvolvedores, comprometendo a confiabilidade das suítes de teste (PARRY et al., 2021).

Habchi et al. (2022) realizaram uma análise qualitativa das fontes, impactos e estratégias de mitigação dos testes *flaky*, essenciais para o desenvolvimento de software. Através de entrevistas com 14 profissionais da indústria, identificaram que as principais fontes de *flakiness* incluem a falta de integração entre componentes do sistema, infraestrutura de teste instável e fatores externos que não se tinha controle. O estudo revelou que a *flakiness* não apenas exige certo tempo dos desenvolvedores e dificulta a integração contínua, mas também afeta negativamente as práticas de teste e a qualidade do sistema, frequentemente resultando em características não determinísticas que são erroneamente rotuladas como testes *flaky*. Para mitigar esses efeitos, os profissionais adotam estratégias de prevenção, como a construção de infraestruturas estáveis e a imposição de diretrizes de teste, enquanto a detecção ainda depende principalmente de *reruns* e inspeções manuais. O estudo enfatiza a importância do monitoramento e da análise de logs para mitigar a *flakiness*, e sugere a necessidade de ferramentas automatizadas para prever, depurar e avaliar com precisão os testes *flaky*.

Os autores identificaram 16 medidas de mitigação adotadas pelos profissionais e destacaram várias oportunidades de automação para aprimorar essas práticas. Como medidas de detecção, foram listadas: reexecutar testes falhos várias vezes, analisar manualmente os resultados e rastros de falhas, verificar o histórico de execução dos testes, expor a *flakiness* de testes antes que ela se manifeste na integração contínua, e utilizar a cobertura de teste para identificar falhas *flaky* Já para medidas de tratamento, foram listadas: corrigir a causa raiz da *flakiness*, ignorar testes *flaky* que não sejam comuns ou custosos, colocar em quarentena os testes *flaky* isolando-os da *pipeline* principal da integração contínua, remover testes *flaky* permanentemente da suíte de testes e documentar testes *flaky* em bancos de dados, relatórios internos ou alertas. Por fim, como medidas de suporte foram levantadas, monitorar e registrar interações do sistema e resultados de testes e estabelecer fluxos de trabalho de teste que protejam a integração contínua.

Garousi e Felderer (2016) conduziram um estudo sobre o desenvolvimento, verificação e manutenção de scripts de teste automatizados de alta qualidade. O artigo destaca a importância de uma gestão holística de ponta a ponta do código de teste, incluindo

desenvolvimento, avaliação de qualidade, melhoria de qualidade e co-manutenção com o código de produção. Os autores analisaram múltiplos casos de projetos recentes e destacaram que embora scripts de teste automatizados ofereçam benefícios como execuções de teste eficientes e repetíveis, seu desenvolvimento é complexo e propenso a erros, exigindo um investimento significativo. O estudo enfatiza a necessidade de técnicas, ferramentas e métodos apropriados para todo o ciclo de vida dos scripts de teste, evidenciando os benefícios das abordagens de Engenharia de Código de Teste de Software (STCE) para a construção de software de alta qualidade de forma eficiente.

2.4 Considerações Finais

Testes automatizados são essenciais para garantir a qualidade e confiabilidade de software, especialmente em projetos que evoluem por meio de muitas versões. Eles oferecem vantagens como execuções de teste repetíveis e eficientes. No entanto, a presença de testes *flaky* pode comprometer tais benefícios, dificultando a identificação de problemas reais no código. Os estudos destacaram que a dependência da ordem de execução, problemas de infraestrutura, e o uso de APIs de rede e aleatoriedade são as principais causas de *flakiness*. Estratégias como isolamento de testes, melhorias na infraestrutura e ferramentas automatizadas são essenciais. O estudo de Gruber et al. (2021) forneceu dados importantes sobre a prevalência de testes *flaky* em projetos Python.

3 METODOLOGIA DO ESTUDO

Este capítulo apresenta o passo a passo da realização do estudo proposto. Como o objetivo deste trabalho foi a aplicação da ferramenta FlaPy (Gruber et al., 2021) em um conjunto de projetos distintos dos originalmente utilizados pelo estudo, foram formuladas as seguintes Questões de Pesquisa (QP):

- **QP1:** Os resultados identificados por Gruber et al. (2021) são similares em projetos distintos?

  O propósito desta questão é selecionar projetos com características distintas aos de de Gruber et al. (2021) e verificar se os resultados são similares.

- **QP2:** Para os testes *flaky* identificados, quais foram as causas?

  Essa questão tem o intuito de classificar as causas dos testes *flaky* identificados, realizando uma análise qualitativa.

Assim, o capítulo está organizado da seguinte maneira: a Seção 3.1 apresenta a seleção dos projetos avaliados e a extração das URLs utilizadas por Gruber et al. (2021). Na Seção 3.2, detalha-se a comparação entre as duas listas de URLs extraídas e a exclusão das equivalentes. A Seção 3.3 descreve a etapa de verificação dos projetos executáveis pela ferramenta FlaPy, bem como o aprimoramento da ferramenta. A Seção 3.4 descreve a análise qualitativa dos testes *flaky* identificados. Em seguida a Seção 3.5 aborda as possíveis ameaças à validade do trabalho proposto. Por fim a Seção 3.6 cita os artefatos gerados e modificados.

3.1 Seleção de projetos e extração das URLs

Para iniciar este estudo, foi selecionada a lista de projetos do repositório Awesome Python[2]. A partir desta lista, todas as URLs dos projetos nela contidos foram extraídas. A Figura 3 mostra o código implementado para extrair do arquivo HTML de readme das URLs dos projetos listados no Awesome Python. Na linha 4 é criada uma lista vazia para armazenar as URLs obtidas, a função da `get_url` declarada na linha 5, itera de forma recursiva pelos elementos de link da pagina html, para todos elementos que contenham uma URL iniciada com "https://github.com/", armazena na lista inicial, como sendo uma URL válida. A linha 26 verifica se o arquivo declarado na linha 19 e aberto na linha 20 existe, para iniciar o procedimento de verificação das URLs por meio da função descrita anteriormente, `get_url`. Como resultado, foram obtidas 527 URLs de projetos.

[2]https://github.com/vinta/awesome-python

Figura 3: Script para extração das URLs dos repositórios da lista Awesome Python

```python
from bs4 import BeautifulSoup

count = 0
links_list = []
def get_url(html__element, output_file):
    links = html__element.find_all("a", href=True)
    global count, links_list
    for link in links:
        url = link["href"]

        if not url.startswith("#") and (url not in links_list) and url.startswith("https://github.com/"):
            print(f"Link: {link}")
            output_file.write(f"{url}\n")
            count += 1
            links_list.append(url)
            get_url(link, output_file)


readme_html_page = "/home/gabrieltararam/PycharmProjects/AwesomeHtmlProcess/awesome_readme_html_clean.html"
with open(readme_html_page, "r", encoding="utf-8") as html_file:
    html_content = html_file.read()

soup = BeautifulSoup(html_content, "html.parser")
div_readme = soup.find("div", {"id": "readme"})
urls_lists = div_readme.find_all("ul")
if div_readme:
    with open("urls_list_links.txt", "w", encoding="utf-8") as output_file:
        for urls_list in urls_lists:
            get_url(urls_list, output_file)
    print("Urls found ", count)
```

Fonte: Autoria própria

Após a obtenção da lista URLs identificou-se que as próximas etapas envolveriam uniformizar as URLs do estudo de Gruber et al. (2021) com a lista de URLs obtida fim de filtrar as não equivalentes, verificar em quais projetos seria possível executar a ferramenta FlaPy e, em seguida, armazenar os resultados obtidos. Desta forma, tornou-se essencial armazenar de forma persistente as seguintes informações: URLs, equivalência entre projetos, projetos executáveis pela ferramenta FlaPy e o estado de execução de cada projeto. Além do armazenamento e organização das informações em uma base de dados, também foi identificado que seria interessante integrá-las em uma ferramenta única, para acesso e edição da base de dados de forma unificada, evitando a repetição de códigos e permitindo reaproveitar trechos de código. Optou-se por utilizar o framework Django por ser desenvolvido em Python assim como a ferramenta FlaPy e os projetos analisados, além de possuir um sistema nativo de ORM utilizando PostgreeSQL. Para o front-end foi selecionado, JavaScript, CSS e jQuery pela popularidade e familiaridade, por fim

Figura 4: Interface da ferramenta e log de extração das URLs.

Fonte: Autoria própria

WebSocket para manter a atualização em tempo real do processamento dos dados entre o front-end e back-end.

Prontamente, iniciou-se o desenvolvimento da ferramenta. Nesta ferramenta, foi implementado o recurso de percorrer a lista de projetos utilizada por Gruber et al. (2021) cadastrando em modelos do Django os pacotes utilizados, como pode ser observado nas Figuras 4 e 5, que exibem a interface da ferramenta e o resultado da extração dos pacotes no painel admin do framework Django.

3.2 Comparação entre os repositórios dos projetos

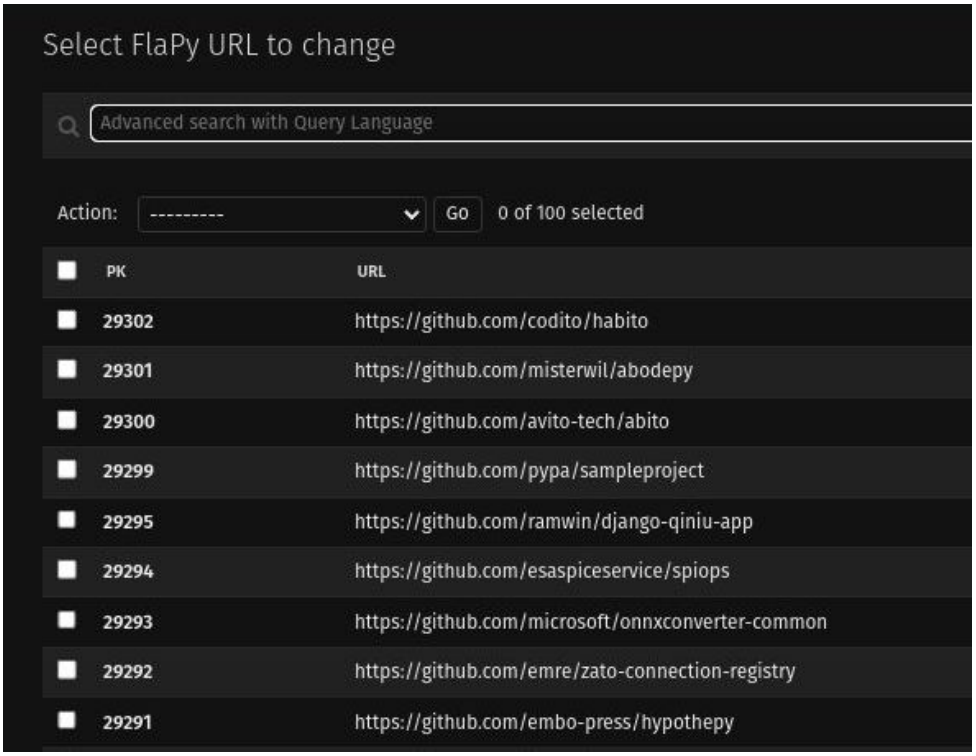
Em seguida, um segundo recurso foi implementado para comparar a lista de projetos do estudo de Gruber et al. (2021) com a lista Awesome Python, com a finalidade de excluir da lista projetos que já haviam sido analisados anteriormente, como pode ser observado na Figura 6.

Primeiramente, foram comparadas somente as URLs com equivalência exata, e depois as URLs similares. Porém essa opção não foi utilizada posteriormente, por conta de mesmo que similares na URL, eram projetos distintos, como pode ser observado nas Figuras 7 e 8, nas quais, por mais que ambas as URLs fazem referência a projetos que utilizam Gunicorn, são projetos distintos. Então foram removidos projetos com o mesmo nome e proprietário. Como resultado dos 527 projetos base, foram filtrados 401 que não haviam sido utilizados pelo estudo de Gruber et al. (2021).

3.3 Filtragem dos projetos que são aplicáveis ao FlaPy

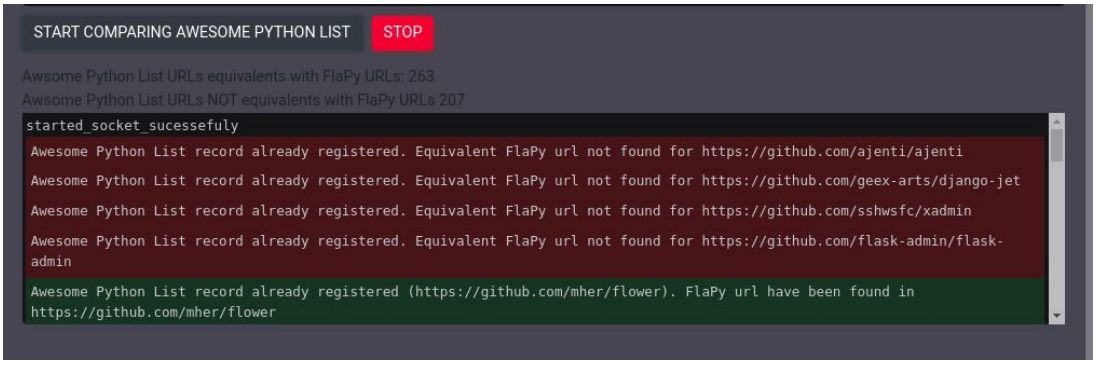
Como próximo passo, foi realizada a verificação de quais projetos poderiam ser executados utilizando a ferramenta FlaPy. Para isso, foi criado um arquivo bash para cada projeto, com a função de executar a ferramenta FlaPy armazenando a saída em um arquivo de texto, podendo ser feito de forma paralela para executar diversos projetos simultanea-

Figura 5: URLs Utilizadas no estudo de Gruber et al. (2021) cadastradas no sistema, visualizadas no painel de admin do Django.



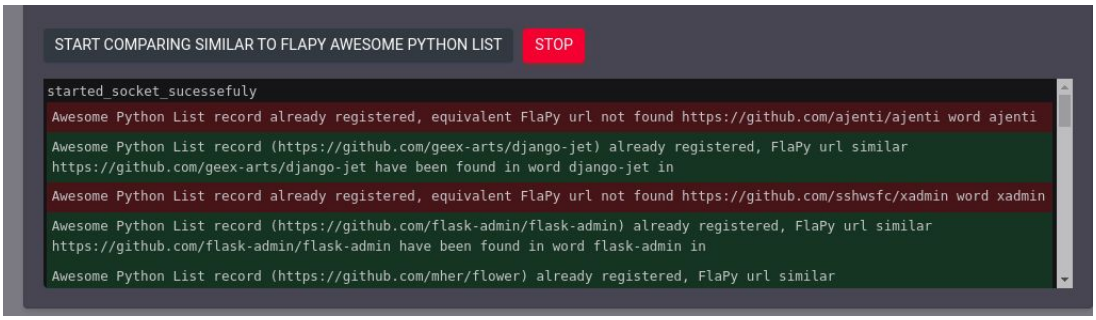
Fonte: Autoria própria

Figura 6: Interface da ferramenta e log de comparação das URLs.



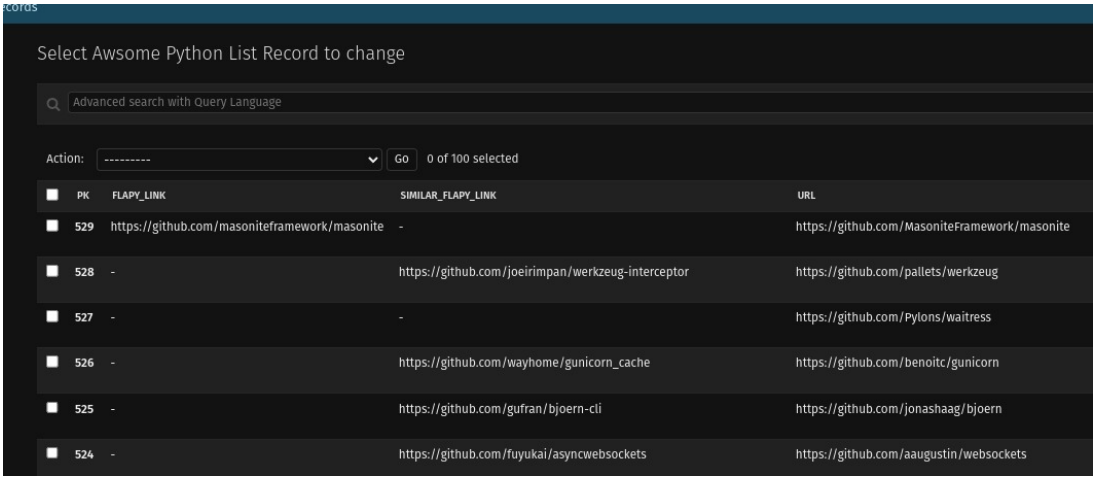
Fonte: Autoria própria

Figura 7: Interface da ferramenta e log de comparação de URLs similares.



Fonte: Autoria própria

Figura 8: Adim do Django exibindo os dados já cadastrado das URLs similares e equivalentes.



Fonte: Autoria própria

mente. Assim obteve-se 22 projetos, que possuíam testes automatizados executáveis com PyTest, com pelo menos um teste aprovado e que o FlaPy conseguia interpretar seus resultados.

Com o intuito de melhorar os resultados, foi analisado o código da ferramenta FlaPy. A primeira observação foi que uma imagem Docker com o FlaPy já dentro dela era instanciada, e o projeto a ser testado era clonado e executado internamente dentro dessa imagem. Com isso, não era possível somente criar um *fork* e melhorar a ferramenta, era necessário alterar a imagem Docker. A imagem foi salva e as alterações foram feitas diretamente nela. Além disso, foi criado um *fork* da ferramenta FlaPy, no qual a imagem local passou a ser utilizada em vez da original As principais alterações realizadas na ferramenta, para ser possível executar mais projetos, foram relacionadas a instalação dos projetos. Na Figura 9 pode ser observado o método `find_setup_files` em que na linha 7 se percorre a lista de possíveis nomes para o arquivo de *setup*, verificando se algum desses nomes é um arquivo presente no projeto analisado, se for, é adicionado a variável de retorno setups_list. Na Figura 10 pode-se observar outras alterações realizadas, na linha 5 foi implementada a lógica de ignorar pacotes comentados, e não tentar realizar sua instalação. Na linha 9 é realizada a *build* e instalação dos arquivos de *setup*. Por fim na linha 14 foi feita a inclusão dos parâmetros de instalação *test*, *dev* e *docs* para evitar que pacotes necessários para execução de teste fossem ignorados. As alterações podem ser listadas nos seguintes itens:

1. Identificação de arquivos de setup do projeto.
2. Ignorar pacotes comentados nos arquivos de *requirements*, que geravam erros na instalação.
3. Instalar arquivos sem *requirements*, mas com `setup.py`.
4. Instalar arquivos sem *requirements*, mas com `pyproject.toml`.
5. Instalar projetos com diversas configurações de `pyproject`, com divisões de `test`, `dev` e `docs`.

Figura 9: Alterações na ferramenta FlaPy para buscar arquivos de setup.py nos projetos, visando ampliar a gama de projetos executáveis.

```
def find_setup_files(self) -> List[str]:
    """Search for setup in common files"""
    setups_list: List[str] = []
    possible_setup_filenames = [
        "setup.py",
    ]
    for file_name in possible_setup_filenames:
        current_file = self._path / file_name
        self._logger.info(
            f"\n #current_file {current_file} "
        )
        if current_file.is_file():
            self._logger.info(
                f"\n ## is setup {current_file} "
            )
            setups_list.append(str(current_file))
    return setups_list
```

Fonte: Autoria própria

Figura 10: Alterações na ferramenta FlaPy para ampliar a possibilidade de projetos executáveis.

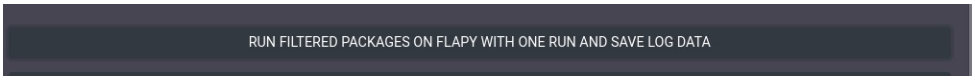
```
for package in self._packages:
    package_striped = package.strip()
    is_a_comment = False
    # ignore commented packages
    if package_striped.startswith('#') or package_striped.startswith('
    """') or "#" in package_striped:
        is_a_comment = True
    if not is_a_comment:
        command_list.append(f'echo "pip install: $(pip install {package
    })"')
for setup_file in self._setup_files:
    command_list.append(f'echo "python setup build:python {setup_file} $
    (python {setup_file} build)"')
    command_list.append(f'echo "python setup install:python {setup_file}
    $(python {setup_file} install)"')

command_list.append('echo "pip install -e .: $(pip install -e .)"')
command_list.append('echo "pip install .: $(pip install .[test,dev,docs
    ])"')
command_list.append('echo "pip freeze: $(pip freeze)"')
```

Fonte: Autoria própria

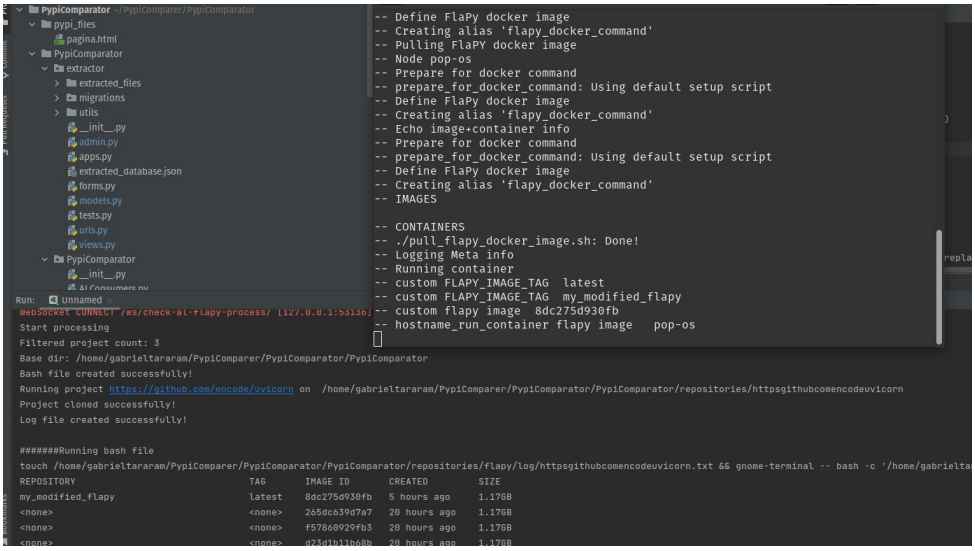
Com essas alterações na ferramenta, foi possível obter 36 projetos executáveis na ferramenta FlaPy, para uma execução. O método de verificar se o projeto continha pelo menos um teste aprovado, e o seu resultado ser possível de analisar pelo FlaPy foi realizado

Figura 11: Interface para executar a ferramenta FlaPy com uma run e armazenar o log.



Fonte: Autoria própria

Figura 12: Execução do primeiro teste com uma run.

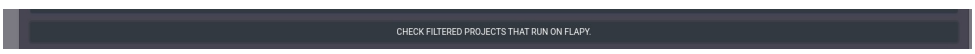


Fonte: Autoria própria

através de outra função feita simultaneamente com a função de executar e processar os pacotes, como pode ser observado nas Figuras 13 e 14. Nelas, são verificados o log e o arquivo CSV gerado.

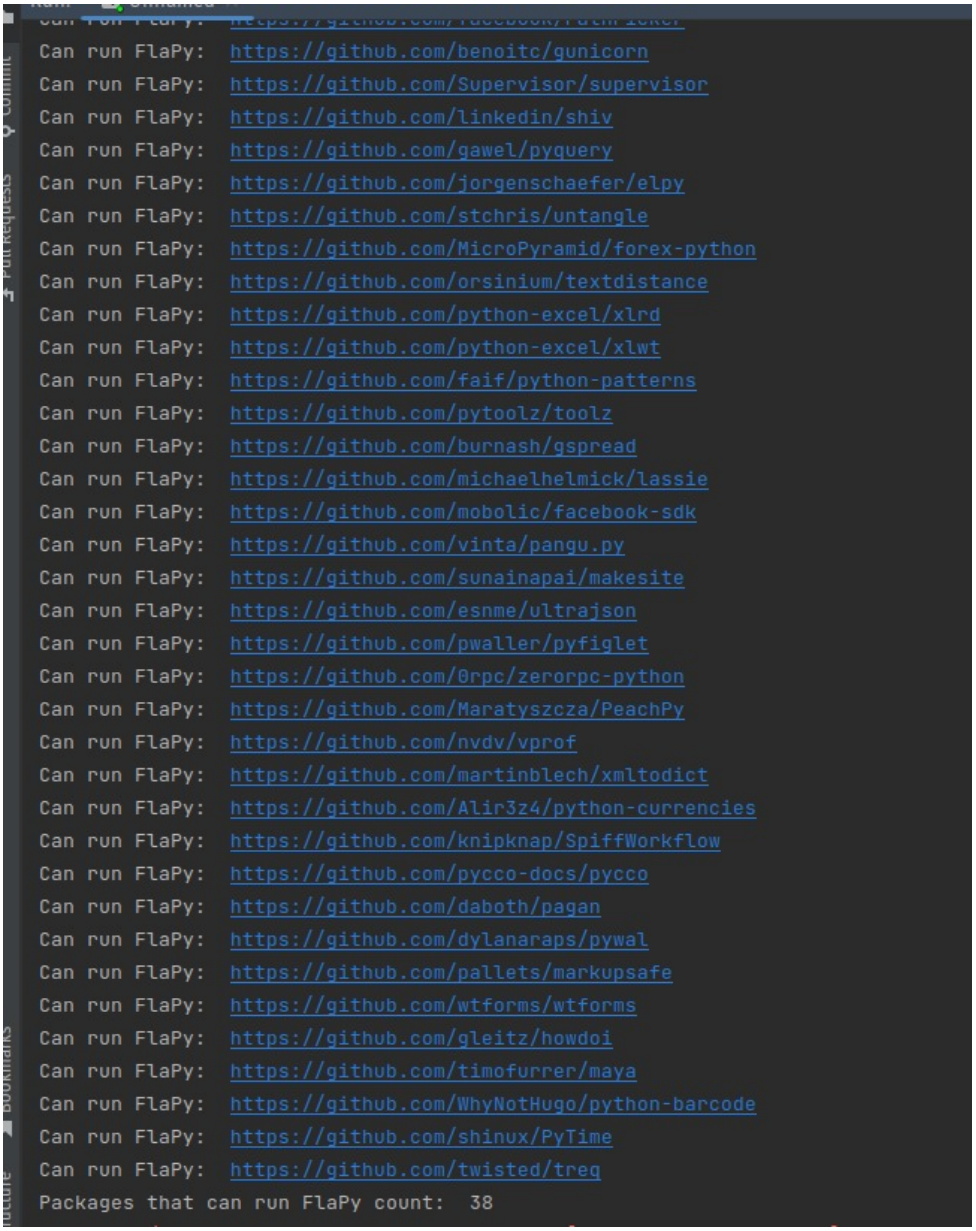
Na próxima etapa, as funcionalidades anteriores de execução e extração dos resultados foram replicadas, porém com parâmetros distintos. Inicialmente executando 20 *runs* e em seguida 400 *runs*, seguindo o padrão do número de *runs* do estudo de Gruber et al. (2021) para identificação de *flakiness*. Em seguida, foi implementado um novo recurso que permite extrair os resultados dos logs gerados pela ferramenta FlaPy e salvá-los em arquivos CSV, possibilitando a análise qualitativa e quantitativa dos resultados. tendo como interface final a Figura 15.

Figura 13: Botão para verificar projetos que rodam FlaPy, através dos logs gerados.



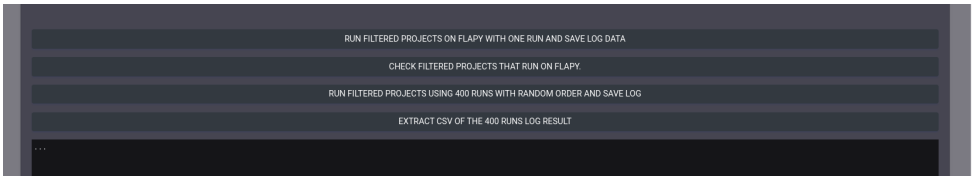
Fonte: Autoria própria

Figura 14: Resultado da execução no terminal da verificação de pacotes que conseguem executar o FlaPy.



Fonte: Autoria própria

Figura 15: Opções da ferramenta desenvolvida para executar e analisar com uma ou 400 *runs*.



Fonte: Autoria própria

3.4 Análise dos testes *flaky* identificados

Como etapa final, foi feita a análise dos resultados, tanto de forma comparativa com o estudo de Gruber et al. (2021) quanto qualitativa dos testes *flaky*.

Para a análise qualitativa, foi selecionado apenas um teste *flaky* dependente de ordem, e feita a sua depuração, de forma a identificar como este tipo de *flakiness* ocorre. Essa decisão foi tomada pois todos os testes *flaky* dependentes de ordem já têm sua causa raiz identificada, sendo um apenas selecionado para exemplificação.

Para os testes *flaky* não dependentes de ordem, também foi feita a depuração, porém analisando cada caso individualmente, verificando as mensagens de erro e percorrendo o código-fonte do projeto com o teste *flaky* identificado, de forma a encontrar os fatores que levaram a *flakiness* do teste de forma mais específica.

3.5 Ameaças a validade

Uma possível ameaça à validade do estudo é o fato de que o código-fonte da ferramenta FlaPy, utilizado no estudo, não foi analisado de forma integral, somente trechos de código, com foco na ampliação da gama de projetos executáveis. Por exemplo, uma falha identificada foi na geração dos resultados CSV, em que a ferramenta incluía vírgulas como separador, porém também nos textos de resultado como "passed,failed", ou não lidava com saídas de log com vírgula, o que atrapalhava a divisão de colunas dos resultados CSV. Isto demonstra que a ferramenta não é isenta de falhas. Apesar disso, como foi feita a análise qualitativa dos testes *flaky* de não dependência de ordem, a veracidade dos resultados pode ser verificada, não ameaçando esta identificação.

Outra ameaça relevante é o número de projetos analisados, em comparação com o estudo de Gruber et al. (2021). Neste estudo foram filtrados e analisados 36 projetos. Além disso, é importante mencionar que apenas projetos de código aberto foram analisados, o que pode reduzir a representatividade dos resultados, já que projetos privados, que podem ter características diferentes, não foram incluídos.

Outra possível ameaça é a análise qualitativa manual, que, por ser realizada por uma pessoa, está sujeita a erros. Para mitigar essa possível ameaça, o orientador revisou manualmente as análises realizadas.

3.6 Artefatos gerados e informações adicionais:

A ferramenta desenvolvida ao longo deste trabalho, assim como os logs, os arquivos CSVs gerados e a imagem Docker modificada estão disponíveis em https://github.com/gabrielmtararam/FlaPy-tools.

No total, conforme pode ser observado no Capítulo 4, os testes foram executados em 282,2 horas, executando entre 4 e 7 projetos simultaneamente, dependendo da velocidade

com que a ferramenta FlaPy finalizava os resultados.

Os experimentos foram executados utilizando as seguintes configurações de hardware: processador AMD Ryzen 5 5500U com Radeon Graphics 2.10 GHz, memória RAM instalada de 20,0 GB e sistema operacional Linux Pop!_OS 20.04 LTS.

4 ANÁLISE DE RESULTADOS

Neste capítulo, serão detalhados os resultados obtidos ao longo da execução do estudo mencionado no capítulo anterior, acompanhados de suas respectivas discussões e análises. A estrutura deste capítulo está organizada da seguinte forma: a Seção 4.1 examina os projetos selecionados; as Seções 4.2 e 4.3 exploram os dados coletados em maior profundidade, com o objetivo de responder às questões de pesquisa propostas neste trabalho.

4.1 Caracterização dos projetos selecionados

Com base na seleção dos projetos descrita na Seção 3.3, os 36 projetos são apresentados na Tabela 1. Para avaliar a popularidade dos projetos, foram considerados os seguintes indicadores do GitHub: número de estrelas, *forks*, *open issues* e *closed issues*. A média de estrelas é 4.259, a mediana 1.927, também foi identificado que o projeto com mais estrelas é o faif/python-patterns com 39.982 estrelas, enquanto o projeto com o menor número de estrelas é o Alir3z4/python-currencies com 72 estrelas. Os dados sobre *open issues* e *closed issues* também podem ser observados.

Tabela 1: Dados detalhados dos projetos

Projeto	**Stars**	**Forks**	**Open Issues**	**Closed Issues**
google/yapf	13704	888	377	463
frewsxcv/python-geojson	904	120	24	74
facebook/PathPicker	5093	281	21	153
benoitc/gunicorn	9666	1735	302	1792
stchris/untangle	611	83	16	41
Supervisor/supervisor	8379	1240	136	1026
linkedin/shiv	1714	93	43	78
gawel/pyquery	2283	181	54	134
jorgenschaefer/elpy	1885	260	370	1240
MicroPyramid/forex-python	651	192	33	59
python-excel/xlwt	1042	285	0	80
orsinium/textdistance	3339	247	0	54
python-excel/xlrd	2148	441	0	164
pytoolz/toolz	4614	258	84	143
faif/python-patterns	39982	6919	11	76
mobolic/facebook-sdk	2733	953	21	218
burnash/gspread	6987	938	40	868
michaelhelmick/lassie	609	49	10	29
vinta/pangu.py	248	26	4	5
sunainapai/makesite	1795	295	7	9
esnme/ultrajson	4290	364	25	320
pwaller/pyfiglet	1347	138	2	60
pycco-docs/pycco	843	146	31	28
Maratyszcza/PeachPy	1968	159	24	65
nvdv/vprof	3953	155	29	57
martinblech/xmltodict	5433	466	85	164
Alir3z4/python-currencies	72	12	4	3
knipknap/SpiffWorkflow	1649	311	8	136
dylanaraps/pywal	8264	318	144	402
daboth/pagan	302	44	4	4
twisted/treq	585	137	55	108
pallets/markupsafe	620	157	0	115
wtforms/wtforms	1495	395	60	396
gleitz/howdoi	10539	869	15	248
timofurrer/maya	3409	199	19	75
shinux/PyTime	159	25	0	4

Fonte: Autoria própria

4.2 QP 1 - Os resultados identificados por Gruber et al. (2021) são similares em projetos distintos?

Analisando todos os CSVs gerados como resultado da ferramenta FlaPy, foi organizada a Tabela 2 que apresenta os proprietários e nome dos projetos em sua primeira coluna, na segunda apresenta o *hash code* do *commit* em que foi executada a analise, em seguida a quantidade de testes presentes no projeto, o número de *runs* executadas para cada projeto, a quantidade de testes *flaky* identificados como dependentes de ordem, a quantidade de testes *flaky* identificados como não dependentes de ordem, a soma dos teste *flaky* identificados, e por fim o tempo total gasto na execução de cada projeto. Dos 36 projetos analisados, em 33 foi possível executar com 420 *runs*, 3 somente com 20 *runs*, não sendo retornada mensagem de erro ao tentar executar com 400 runs, somente o processo executando internamente na imagem Docker não apresentando saída depois de determinado tempo.

De 6859 testes executados, em 74 foram identificados testes *flaky* o que equivale a 1,079% dos testes, valor próximo ao estudo de Gruber et al. (2021)(0.86% dos testes).

Analisando a porcentagem de projetos que apresentaram *flakiness*, dos 36 projetos selecionados, 7 apresentaram algum teste *flaky* (19,44%). Esse valor é superior ao apresentado por Gruber et al. (2021) (4,5%). Dos testes *flaky* identificados por Gruber et al. (2021), 59% foram caracterizados como testes dependentes de ordem, 13% não dependentes de ordem, e 28% relacionados a problemas de infraestrutura. Nos testes realizados para esta pesquisa, dos 74 testes *flaky*, 63 foram identificados como dependentes de ordem (85,1%), e 11 como não dependentes de ordem (14,9%). Essa análise revela uma prevalência maior de testes dependentes de ordem em comparação com o estudo de Gruber et al. (2021).

Tabela 2: Resultados obtidos através da ferramenta FlaPy

Repositorio	**Commit Hash Code**	**#Testes**	**#Execuções**	**#Testes *flaky* Order Dep**	**#Testes *flaky* Non Order Dep**	**#Testes *flaky***	**Tempo Total de Teste**
google/yapf	1afbf71	611	420	59	0	59	9h 10m, 26s
frewsxcv/python-geojson	7afead1	68	420	0	0	0	2h 42m, 36s
facebook/PathPicker	9d5f611	9	420	1	0	1	3h 43m, 55s
benoitc/gunicorn	9802e21	343	420	0	0	0	10h 58m, 31s
stchris/untangle	7eec044	34	420	0	0	0	4h 47m, 41s
Supervisor/supervisor	0e0d3ff	1376	420	0	0	0	8h 10m, 59s
linkedin/shiv	e475914	57	420	0	0	0	12h 13m, 35s
gawel/pyquery	f2533d1	150	420	0	1	1	6h 33m, 28s
jorgenschaefer/elpy	777e990	317	420	0	0	0	19h 49m, 15s
MicroPyramid/forex-python	e1fa196	39	400	2	7	9	5h 26m, 26s
python-excel/xlwt	5a222d0	24	420	0	0	0	3h 51m, 34s
orsinium/textdistance	bcd2174	430	420	0	1	1	41h 19m, 11s
python-excel/xlrd	0c4e80b	83	420	0	0	0	5h 28m, 20s
pytoolz/toolz	7a0382e	204	420	0	0	0	9h 35m, 27s
faif/python-patterns	f401747	65	420	0	0	0	4h 54m, 11s
mobolic/facebook-sdk	3fa89fe	26	420	0	0	0	3h 54m, 28s
burnash/gspread	5e726a2	140	420	0	0	0	10h 11m, 13s
michaelhelmick/lassie	1122c71	44	420	0	0	0	7h 0m, 4s
vinta/pangu.py	ac8b504	56	420	0	0	0	3h 4m, 16s
sunainapai/makesite	40bb66a	50	420	0	0	0	2h 49m, 13s
esnme/ultrajson	e994c12	291	420	0	0	0	11h 36m, 0s
pwaller/pyfiglet	b5bb371	3	420	0	0	0	8h 10m, 34s
pycco-docs/pycco	27b1bf0	17	420	0	2	2	12h 31m, 27s
Maratyszcza/PeachPy	349e8f8	1310	20	0	0	0	4h 53m, 5s
nvdv/vprof	99bb5cd	27	420	0	0	0	5h 58m, 39s
martinblech/xmltodict	9c3ec3c	60	420	0	0	0	4h 30m, 26s
Alir3z4/python-currencies	2545a4b	7	420	0	0	0	4h 25m, 38s
knipknap/SpiffWorkflow	ed6c428	1	420	0	0	0	8h 23m, 3s
dylanaraps/pywal	236aa48	28	420	0	0	0	4h 4m, 34s
daboth/pagan	3fa030c	8	420	0	0	0	7h 48m, 6s
twisted/treq	22a776d	242	20	1	0	1	0h 41m, 59s
pallets/markupsafe	d120573	74	420	0	0	0	5h 48m, 9s
wtforms/wtforms	f6e1e22	352	420	0	0	0	6h 34m, 23s
gleitz/howdoi	6f8813d	25	420	0	0	0	8h 20m, 25s
timofurrer/maya	bcc4885	282	420	0	0	0	9h 2m, 40s
shinux/PyTime	3cf2176	6	20	0	0	0	3h 38m, 18s
Total	–	**6859**	–	**63**	**11**	**74**	**282,2 horas**

Fonte: Autoria própria

4.3 QP 2 - Para os testes *flaky* identificados, quais foram as causas?

Como etapa final, foi realizada uma análise qualitativa dos testes identificados como *flaky*. Os testes dependentes da ordem de execução apresentam um caso mais simples, pois todos exigem que os testes sejam executados na mesma sequência, impossibilitando sua execução em paralelo.

Por exemplo, no teste do projeto google/yapf, o log de erro observado na Figura 17, indica na linha 13 que a diferença entre o conteúdo de determinado arquivo e sua formatação não é o esperado. Analisando o código dos testes, disponível na Figura 18, é possível observar que aparentemente não existe uma aleatoriedade no teste, na linha 2, é definido o código com erro de indentação em uma string; na linha 6, o arquivo é aberto e o código é inserido nele; em seguida, na linha 7, o arquivo é formatado e a diferença entre o conteúdo original e o formatado é retornada; por fim, na linha 8, ocorre a comparação, que é onde o erro se manifesta. Isso indica que o mesmo código é sempre executado com os mesmos parâmetros, porém, ao analisar outros métodos da classe, como

o método `testFormatFileInPlace` presente na linha 10, é possível observar que na linha 13 tenta acessar o mesmo arquivo, `self.test_tmpdir`, e realizar alterações nele, como isso não é realizado utilizando, por exemplo, um `FileLock`, múltiplos métodos podem tentar modificar o mesmo arquivo em paralelo, gerando problemas.

Fazendo outra análise do log de erro, é possível observar que, na linha 11, é feita uma chamada para a função `FormatFile`, e na linha 13 da Figura 17 a mensagem de erro indica que a diferença esperada é uma indentação de 2 espaços, e a obtida é de 4 espaços.

Ao analisar trechos do código-fonte do projeto, disponíveis na Figura 16,, é possível fazer a seguinte análise: na linha 42 é definida indentação com que será realiza a formatação, temos que a largura é proveniente de uma variável global `style` e seu atributo `'INDENT_WIDTH'`, porem essa variável é definida de forma global, como por exemplo é o caso da classe `BasicReformatterTest` presente na linha 45, e do seu método de `setup` na linha 49 em que é feita a chamada da função `SetGlobalStyle` passando como parâmetro o retorno da chamada da função `CreateYapfStyle` em que é definida a largura daindentação para 2 espaços. Portanto, se vários códigos forem executados em paralelo, diferentes configurações globais serão definidas, o que poderá gerar resultados diferentes nos testes.

Figura 16: Trechos do código fonte do projeto yapf

```
style.SetGlobalStyle(style.CreateYapfStyle())

def FormatFile(...):
  ...
  reformatted_source, changed = FormatCode(....)

def FormatCode(...):
  ...
  reformatted_source = FormatTree(tree, style_config=style_config, lines
   =lines)

def FormatTree(...):
  ...
  return reformatter.Reformat(...)

def Reformat(llines, lines=None):
  ...
  indent_width = style.Get('INDENT_WIDTH')

class BasicReformatterTest(yapf_test_helper.YAPFTest):

  @classmethod
  def setUpClass(cls):
    style.SetGlobalStyle(style.CreateYapfStyle())


def CreateYapfStyle():
  """Create the YAPF formatting style."""
  ...
  style['INDENT_WIDTH'] = 2
  ...
  return style
```

Fonte: https://github.com/google/yapf

Figura 17: Mensagem de erro na execução do teste `testFormatFileDiff` do projeto yapf.

```
1  ______________________ FormatFileTest.testFormatFileDiff
       ______________________
2
3  self = <yapftests.yapf_test.FormatFileTest testMethod=testFormatFileDiff
       >
4
5  def testFormatFileDiff(self):
6      unformatted_code = textwrap.dedent("""\
7          if True:
8           pass
9      """)
10     with utils.TempFileContents(self.test_tmpdir, unformatted_code) as
       filepath:
11         diff, _, _ = yapf_api.FormatFile(filepath, print_diff=True)
12 >       self.assertIn('+  pass', diff)
13 E       AssertionError: '+  pass' not found in '--- /tmp/tmpv9a0_3jo/
       tmp2ttegdlw\t(original)\n+++ /tmp/tmpv9a0_3jo/tmp2ttegdlw\t(
       reformatted)\n@@ -1,2 +1,2 @@\n if True:\n- pass\n+    pass\n'
14
15 yapftests/yapf_test.py:252: AssertionError
```

Fonte: Autoria própria

Figura 18: Código fonte da classe `FormatFileTest` do projeto yapf

```
1  def testFormatFileDiff(self):
2      unformatted_code = textwrap.dedent("""\
3          if True:
4           pass
5      """)
6      with utils.TempFileContents(self.test_tmpdir, unformatted_code) as
       filepath:
7          diff, _, _ = yapf_api.FormatFile(filepath, print_diff=True)
8          self.assertIn('+  pass', diff)
9
10 def testFormatFileInPlace(self):
11     unformatted_code = 'True==False\n'
12     formatted_code = 'True == False\n'
13     with utils.TempFileContents(self.test_tmpdir, unformatted_code) as
       filepath:
14         result, _, _ = yapf_api.FormatFile(filepath, in_place=True)
15         self.assertEqual(result, None)
16         with open(filepath) as fd:
17             self.assertCodeEqual(formatted_code, fd.read())
18
19         self.assertRaises(
20             ValueError,
21             yapf_api.FormatFile,
22             filepath,
23             in_place=True,
24             print_diff=True)
```

Fonte: https://github.com/google/yapf

Para os testes não dependentes de ordem, foram analisados os 11 casos identificados.

No primeiro caso, foi analisado o teste `TestWebScrappingEncoding test_get` do projeto PyQuery, das 420 *runs* executadas em ordem fixa, uma falhou e 419 foram aprovadas. Analisando a mensagem de erro de execução dos testes que falharam, ilustrados na Figura 19, pode-se observar que a falha ocorreu ao tentar acessar a URL indicada, como pode ser observado na linha 17, o que pode indicar uma URL incorreta, um problema de *DNS*, ou mesmo falha na conexão de Internet.

Figura 19: Mensagem de erro na execução do teste `TestWebScrappingEncoding` do projeto PyQuery.

```
1  self = <urllib3.connection.HTTPConnection object at 0x7a9ac7d8d4b0>
2
3  def _new_conn(self) -> socket.socket:
4      """Establish a socket connection and set nodelay settings on it.
5
6      :return: New socket connection.
7      """
8      try:
9          sock = connection.create_connection(
10             (self._dns_host, self.port),
11             self.timeout,
12             source_address=self.source_address,
13             socket_options=self.socket_options,
14         )
15     except socket.gaierror as e:
16         raise NameResolutionError(self.host, self, e) from e
17 E       urllib3.exceptions.NameResolutionError: <urllib3.connection.
       HTTPConnection object at 0x7a9ac7d8d4b0>: Failed to resolve 'ru.
       wikipedia.org' ([Errno -3] Temporary failure in name resolution)
18
19 ../flapy_virtual_env/lib/python3.10/site-packages/urllib3/connection.py
       :205: NameResolutionError
```

Fonte: Autoria própria

No código do teste, disponível na Figura 20, é possível observar que ele tenta acessar o site Wikipedia, na linha 3 é feita a tentativa de requisição através do método `get`, e na linha 7 verificado seu resultado. Como esse *endpoint* pode não estar acessível no momento do teste, pode ocorrer um problema de conexão, ou, dependendo do número de requisições, o acesso pode até ser bloqueado, o que gera problemas de consistência entre diferentes execuções.

Figura 20: Código fonte da classe `TestWebScrappingEncoding` do projeto PyQuery

```
class TestWebScrappingEncoding(TestCase):

    def test_get(self):
        d = pq(url='http://ru.wikipedia.org/wiki/....',
               method='get')
        print(d)
        self.assertEqual(d('#pt-login').text(), '...')
```

Fonte: https://github.com/gawel/pyquery

No projeto MicroPyramid/forex-python, a ferramenta FlaPy identificou 7 teste *flaky* não dependentes de ordem.

O primeiro caso de *flakiness* dentre os 7 identificados, é o teste `test_convert_to_btc_on_with_invalid_currency`. Das 400 *runs* executadas em ordem fixa, ocorreu falha em 399 e foi aprovado em somente 1.

Ao verificar o erro, observado na Figura 21, não é possível identificar a causa da instabilidade, porém temos alguns indicativos, como a mensagem de erro presente na linha 7 "AssertionError : RatesNotAvailableError not raised by `convert_to_btc_on`", ela indica que o teste esperava que fosse lançada uma exceção, ao tentar realizar determinada conversão, porém isso não ocorreu.

Figura 21: Mensagem de erro na execução do teste `test_convert_to_btc_on_with_invalid_currency` do projeto forex-python.

```
self = <tests.test_bitcoin.TestConvertToBtcOn testMethod=
    test_convert_to_btc_on_with_invalid_currency>

def test_convert_to_btc_on_with_invalid_currency(self):
    date_obj = datetime.datetime.today() - datetime.timedelta(days=15)
    self.assertRaises(RatesNotAvailableError, convert_to_btc_on, 300, '
    XYZ', date_obj)
    # Expecting RatesNotAvailableError to be raised when converting to
    BTC with invalid currency
E   AssertionError: RatesNotAvailableError not raised by
    convert_to_btc_on

tests/test_bitcoin.py:91: AssertionError
______ TestConvertBtcToCurOn.
    test_convert_to_btc_on_with_invalid_currency ______
```

Fonte: Autoria própria

No entanto, percorrendo as funções utilizadas nos testes, no trecho de código da Figura 22 é possível verificar que na linha 18 é feita uma consulta em determinada API, o que provavelmente é a causa deste teste *flaky*, por diversos fatores, como indisponibilidade do serviço no momento do teste ou bloqueio por limite de requisições excedidas. Embora

a mensagem de erro não ofereça uma indicação clara, essa foi a única causa identificada para a inconsistência entre as repetições dos testes.

Figura 22: Código fonte do método `convert_to_btc_on` do projeto forex-python.

```
def convert_to_btc_on(self, amount, currency, date_obj):
    """
    Convert X amount to Bitcoin based on a given date's rate
    """
    if isinstance(amount, Decimal):
        use_decimal = True
    else:
        use_decimal = self._force_decimal

    start = date_obj.strftime('%Y-%m-%d')
    end = date_obj.strftime('%Y-%m-%d')
    url = (
        'https://api.coindesk.com/v1/bpi/historical/close.json'
        '?start={}&end={}&currency={}'.format(
            start, end, currency
        )
    )
    response = requests.get(url)
```

Fonte: https://github.com/MicroPyramid/forex-python

O segundo teste do projeto MicroPyramid/forex-python em que foi identificado *flakiness* não dependente de ordem foi o teste `test_previous_price_invalid_currency`. Assim como no primeiro teste, das 400 *runs* executadas em ordem fixa, 399 falharam e apenas 1 foi aprovada. Como pode ser observado na Figura 24, a linha 7 indica o erro "AssertionError : RatesNotAvailableError not raised by"assim como o teste anterior. Analisando o código do teste em que ocorreu o erro, é executada uma chamada da função `get_previous_price` conforme pode ser observado na Figura 23. Assim como o caso anterior, conforme mostra a Figura 25, na linha 7 é definida uma URL de determinada API, e em seguida na linha 13 é feita uma tentativa de requisição do tipo get para esta API, o que pode indicar a causa da inconsistência entre os testes, similar ao caso anterior, e novamente a solução indicada seria a aplicação de *Mock* para simular as respostas da API.

Figura 23: Código fonte do método `test_previous_price_invalid_currency` do projeto forex-python.

```
def test_previous_price_invalid_currency(self):
    date_obj = datetime.datetime.today() - datetime.timedelta(days=15)
    self.assertRaises(RatesNotAvailableError, get_previous_price, 'XYZ',
    date_obj)
```

Fonte: https://github.com/MicroPyramid/forex-python

Figura 24: Mensagem de erro na execução do teste `test_previous_price_invalid_currency` do projeto forex-python.

```
____________ TestPreviousPrice.test_previous_price_invalid_currency
    ____________

self = <tests.test_bitcoin.TestPreviousPrice testMethod=
    test_previous_price_invalid_currency>

def test_previous_price_invalid_currency(self):
    date_obj = datetime.datetime.today() - datetime.timedelta(days=15)
>   self.assertRaises(RatesNotAvailableError, get_previous_price, 'XYZ',
    date_obj)
E   AssertionError: RatesNotAvailableError not raised by
    get_previous_price

tests/test_bitcoin.py:34: AssertionError
```

Fonte: Autoria própria

Figura 25: Código fonte do método `get_previous_price` do projeto forex-python.

```
def get_previous_price(self, currency, date_obj):
    """
    Get price for one Bitcoin on given date
    """
    start = date_obj.strftime('%Y-%m-%d')
    end = date_obj.strftime('%Y-%m-%d')
    url = (
        'https://api.coindesk.com/v1/bpi/historical/close.json'
        '?start={}&end={}&currency={}'.format(
            start, end, currency
        )
    )
    response = requests.get(url)
```

Fonte: https://github.com/MicroPyramid/forex-python

O terceiro caso `test_previous_price_list_with_invalid_currency`, é muito similar aos casos anteriores. Assim como no primeira teste, das 400 *runs* executadas em ordem fixa, ocorreu falha em 399 e foi aprovado em somente 1. O log de erros, conforme pode ser observado na Figura 27, na linha 10 a mensagem afirma que o resultado da comparação do teste não foi o esperado, ao invés de retornar "False"foi retornada um dicionario contendo datas como chave e valores numéricos como conteúdo. Analisando o código da função de teste disponível na Figura 26 em que na linha 4 é executado o método na `get_previous_price_list`. Nesse método, disponível na Figura 28 é possível observar que na linha 7 é definida uma URL tendo como alvo determinada API, e em seguida, na linha 13 é feita uma tentativa de requisição do tipo get utilizando a URL definida anteriormente. O que assim como o caso anterior pode indicar a causa da instabilidade entre os

testes. Porém neste caso, não foi um caso de uma falha de tentar acessar a URL, a falha ocorreu no fato da API retornar valores, e não algo vazio gerando um valor equivalente a False. Uma possível explicação é que, no momento em que o teste foi implementado os parâmetro enviados para a API retornavam algo vazio, e agora retornam valores.

Figura 26: Código fonte do método `test_previous_price_list_with_invalid_currency` do projeto forex-python.

```
def test_previous_price_list_with_invalid_currency(self):
    start_date = datetime.datetime.today() - datetime.timedelta(days=15)
    end_date = datetime.datetime.today()
    price_list = get_previous_price_list('XYZ', start_date, end_date)
    self.assertFalse(price_list)
    self.assertEqual(type(price_list), dict)
```

Fonte: https://github.com/MicroPyramid/forex-python

Figura 27: Mensagem de erro na execução do teste `test_previous_price_list_invalid_currency` do projeto forex-python.

```
_____ TestPreviousPriceList.
   test_previous_price_list_with_invalid_currency _____

self = <tests.test_bitcoin.TestPreviousPriceList testMethod=
   test_previous_price_list_with_invalid_currency>

def test_previous_price_list_with_invalid_currency(self):
    start_date = datetime.datetime.today() - datetime.timedelta(days=15)
    end_date = datetime.datetime.today()
    price_list = get_previous_price_list('XYZ', start_date, end_date)
>   self.assertFalse(price_list)
E   AssertionError: {'2024-05-26': 68622.3849, '2024-05-27': 69594.3797,
    '2024-05-28': 68509.3062,
                    '2024-05-29': 67651.6413, '2024-05-30': 68381.619, '
   2024-05-31': 67673.9278,
                    '2024-06-01': 67789.7859, '2024-06-02': 67891.8579,
   '2024-06-03': 69109.3176,
                    '2024-06-04': 70565.2068, '2024-06-05': 71158.5455,
   '2024-06-06': 70830.2702,
                    '2024-06-07': 69458.6146, '2024-06-08': 69447.7584,
   '2024-06-09': 69683.6751} is not false

tests/test_bitcoin.py:52: AssertionError
```

Fonte: Autoria própria

Figura 28: Código fonte do método `get_previous_price_list` do projeto forex-python

```
1  def get_previous_price_list(self, currency, start_date, end_date):
2      """
3      Get list of Bitcoin prices between two dates
4      """
5      start = start_date.strftime('%Y-%m-%d')
6      end = end_date.strftime('%Y-%m-%d')
7      url = (
8          'https://api.coindesk.com/v1/bpi/historical/close.json'
9          '?start={}&end={}&currency={}'.format(
10                 start, end, currency
11             )
12     )
13     response = requests.get(url)
14     if response.status_code == 200:
15         data = self._decode_rates(response)
16         price_dict = data.get('bpi', {})
17         return price_dict
18     return {}
```

Fonte: https://github.com/MicroPyramid/forex-python

Os próximos três casos, repetem os códigos desses três primeiros, `test_convert_to_btc_on_with_invalid_currency`, `test_previous_price_invalid_currency`, `test_previous_price_list_with_invalid_currency`, no entanto, diferem por utilizarem um método de *setup* com parâmetros diferentes, especificamente com o parâmetro force_decimal=True, conforme pode ser observado na Figura 29 na linha 2.

Figura 29: Código fonte da variação de *setup* dos testes do projeto forex-python.

```
1  def setUp(self):
2      self.b = BtcConverter(force_decimal=True)
```

Fonte: https://github.com/MicroPyramid/forex-python

O sétimo e último caso identificado para este projeto é o `test_previous_price_valid_currency`. Diferente do padrão apresentado nos testes anteriores, das 400 *runs* executadas em ordem fixa, foi aprovado em 399 e falhou em somente 1.

O log de erros, conforme pode ser observado na Figura 30, na linha 9 a mensagem afirma que o resultado da comparação do teste não foi o esperado, em vez de retornar um valor do tipo *float*, foi retornado um valor nulo, possivelmente indicando que não houve retorno de alguma chamada de função.

Analisando o código do teste, que pode ser observado na Figura 31, na linha 3 é feita uma chamada do método `get_previous_price`, disponível na Figura 25, onde, na linha 7, é definida uma URL de uma determinada API, e na linha 13, é realizada uma tentativa de requisição do tipo get, indicando a possível causa da inconsistência entre os testes.

Como o resultado da consulta depende do retorno da API, se em um momento a consulta retorna um valor e em outro momento não retorna nada, isso gera uma inconsistência entre os resultados dos testes.

Figura 30: Mensagem de erro na execução do teste `test_previous_price_valid_currency` do projeto forex-python.

```
_____ TestPreviousPrice.test_previous_price_valid_currency _____

self = <tests.test_bitcoin.TestPreviousPrice testMethod=
    test_previous_price_valid_currency>

def test_previous_price_valid_currency(self):
    date_obj = datetime.datetime.today() - datetime.timedelta(days=15)
    price = get_previous_price('USD', date_obj)
>   self.assertEqual(type(price), float)
E   AssertionError: <class 'NoneType'> != <class 'float'>

tests/test_bitcoin.py:30: AssertionError
```

Fonte: Autoria própria

Figura 31: Código fonte do método `test_previous_price_valid_currency` do projeto forex-python.

```
def test_previous_price_valid_currency(self):
    date_obj = datetime.datetime.today() - datetime.timedelta(days=15)
    price = self.b.get_previous_price('USD', date_obj)
    self.assertEqual(type(price), Decimal)
```

Fonte: https://github.com/MicroPyramid/forex-python

Outro projeto em que foram identificadas *flakiness* que não dependem da ordem dos testes, foi o pycco-docs/pycco; nele foram encontrados dois casos. O primeiro método foi o `test_get_language_bad_source` em que das 420 *runs* em ordem fixa, 143 passaram e 277 foram reprovadas.

Analisando o código do teste identificado como *flaky*, disponível na Figura 33, na linha 1 e 2 que é importada a biblioteca hypothesis[3] (utilizada para facilitar e gerar testes automatizados de forma parametrizada em Python), e na linha 4 que o teste `test_get_language_bad_source` utiliza ela para ser gerado.

Analisando o log de erros, disponível na Figura 32, na linha 18 a própria biblioteca hypothesis emite uma mensagem indicando que pode existir um problema de *flakiness*, por não ser aprovado na primeira execução e o padrão não se repetir. Na linha 25 a biblioteca também, indica a causa da falha, indicando que o tempo do teste excedeu o

[3]https://github.com/HypothesisWorks/hypothesis/tree/master/hypothesis-python

esperado pela biblioteca, sendo que ela mesma indica uma possível solução na mensagem de erro, como sendo definir como parâmetro do teste deadline=None.

Figura 32: Mensagem de erro na execução do teste `test_get_language_bad_source` do projeto pycco.

```
_________________________ test_get_language_bad_source
    _________________________

    @given(text() | none())
def test_get_language_bad_source(source):

tests/test_pycco.py:95:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
../flapy_virtual_env/lib/python3.10/site-packages/hypothesis/core.py
    :651: in execute_once
    self.__flaky(
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self = <hypothesis.core.StateForActualGivenExecution object at 0
    x7cf47c574760>
message = "Hypothesis test_get_language_bad_source(source='') produces
    unreliable results: Falsified on the first call but did not on a
    subsequent one"

    def __flaky(self, message):
        if len(self.falsifying_examples) <= 1:
>           raise Flaky(message)
E           hypothesis.errors.Flaky: Hypothesis
    test_get_language_bad_source(source='') produces unreliable results:
    Falsified on the first call but did not on a subsequent one

../flapy_virtual_env/lib/python3.10/site-packages/hypothesis/core.py
    :856: Flaky
---------------------------------- Hypothesis
    ----------------------------------
Falsifying example: test_get_language_bad_source(
    source='',
)
Unreliable test timings! On an initial run, this test took 266.86ms,
    which exceeded the deadline of 200.00ms, but on a subsequent run it
    took 3.28 ms, which did not. If you expect this sort of variability
    in your test timings, consider turning deadlines off for this test by
    setting deadline=None.
```

Fonte: Autoria própria

Figura 33: Código fonte do método `test_get_language_bad_source` do projeto pycco.

```
from hypothesis import assume, example, given
from hypothesis.strategies import booleans, lists, none, text,
    sampled_from, data
...
@given(text() | none())
def test_get_language_bad_source(source):
    code = "#!/usr/bin/python\n"
    code += FOO_FUNCTION
    assert p.get_language(source, code) == PYTHON
    with pytest.raises(ValueError) as e:
        assert p.get_language(source, "badlang")

    msg = "Can't figure out the language!"
    try:
        assert e.value.message == msg
    except AttributeError:
        assert e.value.args[0] == msg
```

Fonte: https://github.com/pycco-docs/pycco

Para o segundo caso `test_process`, das 420 *runs* em ordem fixa, 52 passaram e 368 foram reprovadas.

Analisando o código do teste, disponível na Figura 34, é possível observar que na linha 1 é utilizado os *decorators* da biblioteca hypothesis indicando que o teste é gerado através dela.

conforme pode ser visto na Figura 35. Na linha 5 possível verificar novamente pelo log que o tempo do teste excedeu o esperado pela biblioteca hypothesis em algum caso, gerando assim inconsistência entre os resultados.

Figura 34: Código fonte do método `test_process` do projeto pycco.

```
@given(booleans(), booleans(), data())
def test_process(preserve_paths, index, data):
    lang_name = data.draw(sampled_from([l["name"] for l in
    supported_languages.values()]))
    p.process([PYCCO_SOURCE], preserve_paths=preserve_paths,
              index=index,
              outdir=tempfile.gettempdir(),
              language=lang_name)
```

Fonte: https://github.com/pycco-docs/pycco

Figura 35: Mensagem de erro na execução do teste `test_process` do projeto pycco.

```
Falsifying example: test_process(
    preserve_paths=False, index=True, data=data(...),
)
Draw 1: 'ruby'
Unreliable test timings! On an initial run, this test took 269.31ms,
   which exceeded the deadline of 200.00ms, but on a subsequent run it
   took 198.22 ms, which did not. If you expect this sort of variability
    in your test timings, consider turning deadlines off for this test
   by setting deadline=None.
- generated xml file: /workdir/sameOrder/tmp/
   httpsgithubcompyccodocspycco_output0.xml -

---------- coverage: platform linux, python 3.10.1-final-0 -----------
Coverage XML written to file /workdir/sameOrder/tmp/
   httpsgithubcompyccodocspycco_coverage0.xml
```

Fonte: Autoria própria

O ultimo projeto em que foram identificados testes *flaky* sem dependência de ordem, foi o projeto life4/textdistance. O teste `test_normalization_range [alg1]` em ordem fixa de execução, falhou em 1 e foi aprovado em 419 execuções.

Novamente, analisando o código do teste, disponível na Figura 36, é possível observar que na linha 2 são utilizados os *decorators* da biblioteca hypothesis indicando que o teste é gerado através dela.

De forma similar aos testes *flaky* do projeto pycco, conforme pode ser observado na Figura 37, na linha 7 é possível verificar novamente pelo log que o tempo do teste excedeu o esperado pela biblioteca hypothesis, gerando assim inconsistência entre os resultados.

Figura 36: Código fonte do método `test_normalization_range` do projeto textdistance.

```
@pytest.mark.parametrize('alg', ALGS)
@hypothesis.given(
    left=hypothesis.strategies.text(),
    right=hypothesis.strategies.text(),
)
def test_normalization_range(left, right, alg):
    assert 0 <= alg.normalized_distance(left, right) <= 1
    assert 0 <= alg.normalized_similarity(left, right) <= 1
```

Fonte: https://github.com/life4/textdistance

Figura 37: Mensagem de erro na execução do teste `test_normalization_range` do projeto textdistance.

```
False, 'external': True})) produces unreliable results: Falsified on the
    first call but did not on a subsequent one
E           Falsifying example: test_normalization_range(
E               alg=Hamming({'qval': 1, 'test_func': <function Base.
    _ident at 0x745a5df87c70>, 'truncate': False, 'external': True}),
E               left='3',
E               right='B',
E           )
E           Unreliable test timings! On an initial run, this test took
    336.82ms, which exceeded the deadline of 200.00ms, but on a
    subsequent run it took 0.08 ms, which did not. If you expect this
    sort of variability in your test timings, consider turning deadlines
    off for this test by setting deadline=None.
```

Fonte: Autoria própria

4.4 Lições Aprendidas

Além das análises para identificação das causas de *flakiness*, é possível propor soluções para os padrões observados.

No caso dos testes *flaky* causados por acesso a recursos externos como API, sites, etc. É possível a utilização de *Mock* para simular uma resposta do recurso, evitando problemas de conexão. Porém, apesar de isso evitar erros no teste, permite que uma alteração na API passe despercebida pelos testes automatizados.

Para o padrão de múltiplos acessos a variáveis globais, uma possível alternativa para corrigir esse caso de *flakiness* é não utilizar uma variável global, mas sim implementar uma classe baseada na variável, e cada teste instanciar um objeto dessa classe, alterando conforme necessário.

No caso da biblioteca Hypothesis, ela mesma indica uma possível solução na mensagem de erro, sugerindo definir como parâmetro do teste deadline=None.

4.5 Considerações Finais

As análises mostraram uma incidência de *flakiness* nos projetos Python selecionados para esta pesquisa, 19,44% dos projetos apresentaram testes *flaky*, significativamente acima dos 4,5% reportados por Gruber et al. (2021). A maioria (85,1%) dos testes*flaky* eram dependentes da ordem de execução. Testes não dependentes de ordem mostraram inconsistências, principalmente devido à dependência de APIs externas. Esses resultados sublinham a necessidade de mitigar o *flakiness* para garantir a confiabilidade dos testes.

As conclusões obtidas neste projeto serão detalhadas no próximo capítulo.

5 Conclusão

Este estudo teve como objetivo examinar a ocorrência de testes *flaky* em projetos Python distintos dos analisados por Gruber et al. (2021), utilizando a ferramenta FlaPy para responder às questões de pesquisa propostas. Os resultados mostraram que dos 36 projetos selecionados, 7(19,44%) apresentaram testes *flaky*, uma taxa maior que os apresentados por Gruber et al. (2021), em que de 22352 projetos 1006(4,5%) apresentaram testes *flaky*. Esse resultado evidencia a alta prevalência de instabilidade nos testes dos projetos analisados. De 6859 testes executados, 74(1,079%) foram identificados como testes *flaky*, valor próximo ao identificado por Gruber et al. (2021)(0.86% dos testes).

A análise dos tipos de testes *flaky* revelou que a 63(85,1%) testes *flaky* eram dependentes da ordem de execução, o que ressalta a necessidade de um cuidado com a independência entre os testes para minimizar a chance de ocorrer *flakiness*. Os testes não dependentes de ordem, embora menos comuns, mostraram inconsistências principalmente devido à dependência de APIs, acesso a recursos externos ao projeto testado e limite de tempo de execução de bibliotecas externas aos projetos.

Para pesquisas futuras, sugere-se a expansão da ferramenta FlaPy para permitir a execução de ainda mais projetos com formas de instalação distintas, por exemplo de forma automática buscar pacotes que são necessários para instalação, e configuração de ambiente de teste. Em resumo, este trabalho contribui para a compreensão da *flakiness* em testes de software em projetos implementados utilizando Python e oferece informações valiosas para a melhoria da confiabilidade dos testes automatizados.

Referências

1 GRUBER, M. et al. An empirical study of flaky tests in python. In: *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 2021. p. 148–158. Disponível em: <https://doi.org/10.1109/ICST49551.2021.00026>.

2 KUMAR, D.; MISHRA, K. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, v. 79, p. 8–15, 12 2016.

3 GAROUSI, V.; FELDERER, M. Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Softw.*, v. 33, n. 3, p. 68–75, 2016. Disponível em: <https://doi.org/10.1109/MS.2016.30>.

4 HABCHI, S. et al. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 2022. p. 244–255. Disponível em: <https://doi.org/10.1109/ICST53961.2022.00034>.

5 LISTFIELD, J. *Google testing blog: Where do our flaky tests come from?* 2017. <https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come->. Accessed: 2024-1-9.

6 ECK, M. et al. 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>.

7 GRUBER, M. et al. Practical flaky test prediction using common code evolution and test history data. In: *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 2023. p. 210–221. Disponível em: <https://doi.org/10.1109/ICST57152.2023.00028>.

8 ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:General concepts. *ISO/IEC/IEEE 29119-1:2022(E)*, p. 1–60, 2022.

9 PRESSMAN, R.; MAXIM, B. *Software Engineering: A Practitioner's Approach*. Seventh. [S.l.]: Mc GRAW-HILL, 2010. ISBN 978–0–07–337597–7.

10 REITZ, K. *Requests: HTTP for Humans*. 2024. <https://github.com/psf/requests/blob/0e322af87745eff34caffe4df68456ebc20d9068/src/requests/models.py#L768>. Accessed: 2024-07-30.

11 PARRY, O. et al. 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>.