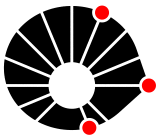

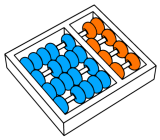

Um Dataset de Aplicações de Microserviços em Produção

Pedro Henrique Rodrigues de Araújo
Breno Bernard Nicolau de França

Relatório Técnico - IC-PFG-25-46
Projeto Final de Graduação
2025 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Um Dataset de Aplicações de Microserviços em Produção

Pedro Henrique Rodrigues de Araújo Breno Bernard Nicolau de França

Resumo

Atualmente, a arquitetura de microserviços está amplamente presente em aplicações do mercado, consolidando-se como um dos principais padrões para sistemas distribuídos em nuvem, devido à modularidade, escalabilidade e independência de seus componentes. Em razão dessa popularidade crescente, diversas pesquisas acadêmicas dependem de aplicações baseadas em microserviços para servir de benchmarks em seus estudos.

Segundo (D'ARAGONA et al., 2024), grande parte desses trabalhos fundamenta suas análises em aplicações desenvolvidas especificamente para fins experimentais, o que pode introduzir vieses relevantes nos resultados. Com o objetivo de mitigar esse problema, os autores propuseram a criação e a disponibilização de um dataset composto por aplicações open source desenvolvidas segundo a arquitetura de microserviços.

Este estudo também tem como objetivo disponibilizar um dataset público de aplicações baseadas em microserviços. Diferentemente de (D'ARAGONA et al., 2024), contudo, este trabalho se restringe a aplicações reais, excluindo provas de conceito e repositórios de caráter experimental ou simplificado. O intuito é reduzir vieses e representar de forma mais fidedigna o ecossistema real de aplicações que adotam essa arquitetura.

Para atingir esse objetivo, realizou-se um estudo de Mineração de Repositórios de Software (MSR), no qual foi desenvolvido um notebook em Python que consultou a API do GitHub. Inicialmente, foram identificados 1.969 repositórios; após uma seleção criteriosa, 126 foram selecionados para análise manual. Ao final desse processo, obtiveram-se 9 repositórios que atendiam aos critérios estabelecidos, compondo assim o dataset final de aplicações reais baseadas em microserviços.

1 Introdução

A arquitetura de microserviços tem se consolidado como uma das abordagens mais relevantes para o desenvolvimento de sistemas distribuídos e aplicações em nuvem, impulsionada pela necessidade de maior escalabilidade, modularidade e agilidade na entrega de software (LEWIS; FOWLER, 2014). Segundo Fowler e Lewis, essa arquitetura consiste na decomposição de uma aplicação em pequenos serviços independentes, cada um responsável por uma funcionalidade específica, executando em seu próprio processo e comunicando-se por mecanismos leves, o que possibilita ciclos de desenvolvimento autônomos, a adoção de diferentes tecnologias e a escalabilidade seletiva (LEWIS; FOWLER, 2014).

Apesar de seus benefícios, a adoção de microserviços introduz desafios significativos, como a coordenação entre serviços e o maior investimento em automação e processos de DevOps, elementos também destacados por (LEWIS; FOWLER, 2014). Tais desafios têm motivado a intensificação de estudos empíricos sobre microserviços, porém, grande parte

dessas pesquisas ainda se baseia em aplicações feitas sob medida, como provas de conceito e repositórios simplificados, o que pode introduzir vieses e limitar a validade externa dos resultados (D'ARAGONA et al., 2024).

Diante dessa limitação, (D'ARAGONA et al., 2024) argumenta que pesquisas sobre microserviços frequentemente se baseiam em repositórios desenvolvidos exclusivamente para o estudo. A partir disso, (D'ARAGONA et al., 2024) realizou um estudo em larga escala para identificar e catalogar projetos open-source baseados na arquitetura de microserviços. Os pesquisadores partiram de uma base com 389.559 repositórios que, após a aplicação de filtros, foi reduzida para um conjunto de 3.804 repositórios. Após uma etapa de rotulagem manual, o dataset resultante apresentou 378 aplicações com mais de 100 commits e com pelo menos três microserviços, que incluem aplicações acadêmicas, industriais e exemplos. Para cada repositório, foi documentado o tamanho do projeto, número de contribuidores, objetivo e fundação de apoio. Esse dataset proposto por (D'ARAGONA et al., 2024) permite que a comunidade escolha um repositório mais alinhado com o objetivo de cada pesquisa. Apesar de (D'ARAGONA et al., 2024) identificar corretamente o problema dos vieses decorrentes do uso de aplicações não representativas, o dataset proposto ainda inclui uma diversidade de aplicações que abrangem exemplos, provas de conceito e projetos didáticos, o que pode não eliminar completamente o viés em estudos que buscam representar a realidade de sistemas em produção. Em contraste, este estudo tem como objetivo construir e disponibilizar um dataset público formado exclusivamente por aplicações reais desenvolvidas segundo a arquitetura de microserviços, excluindo projetos experimentais ou de caráter didático. Para isso, realizou-se um estudo de Mineração de Repositórios de Software (MSR) utilizando a API do GitHub, combinando filtragem automatizada e análise manual na seleção de repositórios. O resultado obtido é um conjunto de aplicações reais que pode servir de base para estudos futuros, contribuindo para a redução de vieses nas pesquisas sobre a arquitetura de microserviços. O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta o contexto para microserviços, Docker em microserviços e o processo de Mineração de Repositórios de Software. A Seção 3 descreve o processo de seleção dos repositórios. A Seção 4 apresenta o dataset resultante do processo de seleção. A Seção 5 discute as ameaças à validade do dataset construído e, por fim, a Seção 6 apresenta as considerações finais deste trabalho.

2 Background

A arquitetura de microserviços tem sido fortemente discutida tanto na academia como na indústria por se tratar de uma alternativa à arquitetura monolítica (LEWIS; FOWLER, 2014). Nesse contexto, a forma como o código é organizado em repositórios, bem como a estratégia de conteinerização adotada, influencia diretamente a forma como os microserviços são desenvolvidos, implantados e estudados empiricamente.

2.1 Arquitetura de Microserviços

A arquitetura de microserviços, descrita por Lewis e Fowler em (LEWIS; FOWLER, 2014), propõe o desenvolvimento de sistemas como um conjunto de pequenos serviços de respon-

sabilidade bem definida, que executam em processos isolados e se comunicam por mecanismos leves, como APIs HTTP ou soluções de mensageria. Cada serviço pode ser desenvolvido, implantado e escalado de forma independente, permitindo ciclos de desenvolvimento autônomos e a adoção de diferentes tecnologias por equipe ou por serviço. Entre as características recorrentes destacam-se a organização do sistema em torno de capacidades de negócio, a automatização de pipelines de implantação, a observabilidade distribuída e o forte alinhamento com práticas de DevOps (FRANCESCO; LAGO; MALAVOLTA, 2019). Em contrapartida, o aumento do número de serviços introduz complexidade operacional, exigindo mecanismos apropriados de orquestração, descoberta de serviços, monitoramento e tratamento de falhas.

2.2 Mono-repositório e Multi-repositórios

No contexto de microserviços, a organização de repositórios tende a seguir dois principais padrões: mono-repo e multi-repo. No primeiro padrão, todo o código da aplicação é concentrado em único repositório, enquanto no segundo ele é dividido em múltiplos repositórios independentes. Segundo (REISINGER et al., 2019), o modelo mono-repo facilita a visibilidade do sistema como um todo, permitindo maior padronização de práticas de desenvolvimento e facilitando refatorações que afetem diferentes partes da aplicação. Por outro lado, a organização em multi-repo oferece maior isolamento e modularidade entre os microserviços da aplicação, permitindo mais autonomia em todo o ciclo de desenvolvimento, incluindo versionamento e pipelines de DevOps separados.

2.3 Docker na arquitetura de microserviços

Segundo (JARAMILLO; NGUYEN; SMART, 2016), a ferramenta Docker tornou-se uma tecnologia disruptiva ao transformar a forma como aplicações são empacotadas, distribuídas e executadas, oferecendo contêineres leves, portáteis e facilmente replicáveis (JARAMILLO; NGUYEN; SMART, 2016). Cada contêiner fornece um ambiente isolado que encapsula exatamente as dependências necessárias para a execução de um serviço, o que se alinha diretamente ao princípio de independência e autonomia dos microserviços.

Como destacado por (JARAMILLO; NGUYEN; SMART, 2016), o uso de contêineres facilita a automação ao longo de todo o ciclo de vida do software, favorecendo pipelines de integração e entrega contínua, além de reduzir o acoplamento entre equipes, que passam a construir, testar e implantar seus serviços de forma mais isolada. Em projetos mono-repo, múltiplos serviços podem compartilhar o mesmo repositório, mas possuir Dockerfiles distintos e pipelines de CI/CD específicos. Em projetos multi-repo, é comum que cada repositório esteja diretamente associado a uma imagem de contêiner ou a um conjunto pequeno de serviços relacionados.

2.4 Mineração de Repositórios de Software

O método de Mineração de Repositórios de Software (MSR) dedica-se à extração e análise de dados provenientes de repositórios de software. Esses dados permitem investigar práticas

de desenvolvimento, evolução de sistemas e fenômenos relacionados à engenharia de software empírica (VIDONI, 2022)

Segundo (VIDONI, 2022), estudos MSR geralmente envolvem três etapas: i) seleção de repositórios, ii) extração de dados relevantes e iii) análise sistemática desses dados para responder às questões de pesquisa. O autor ressalta que a condução inadequada dessas etapas, especialmente a ausência de critérios de seleção transparentes e a falta de discussão de ameaças à validade, pode comprometer a confiabilidade dos resultados e dificultar sua replicação (VIDONI, 2022). O método de MSR pode ser aplicado em diversos contextos, como no estudo da evolução de software (SAHA et al., 2013) (RAY et al., 2012) e na predição de defeitos (KIM et al., 2014) (NAGAPPAN; BALL; ZELLER, 2014). Exemplos de estudos MSR incluem a investigação da evolução de clones de código em sistemas open source (SAHA et al., 2013), análises em larga escala relacionando linguagens de programação e qualidade de código em projetos do GitHub (RAY et al., 2012) e estudos de predição de defeitos baseados em histórico de mudanças e métricas extraídas de repositórios (KIM et al., 2014; NAGAPPAN; BALL; ZELLER, 2014). Esses trabalhos ilustram como diferentes tipos de dados provenientes de repositórios podem ser explorados para responder questões sobre desenvolvimento de software.

3 Métodos

A fim de responder à seguinte questão de pesquisa, descrevemos o método de pesquisa nessa seção.

RQ Quais são alguns dos repositórios open-source que representam aplicações reais em produção desenvolvidas na arquitetura de microserviços?

3.1 Critérios de Seleção

A seguir, detalham-se cada um dos critérios utilizados para determinar se um repositório poderia ser incluído neste estudo. Tais critérios foram definidos para identificar aplicações reais baseadas na arquitetura de microserviços, com documentação suficiente e atividade recente que permitisse sua avaliação.

- Disponibilidade na plataforma GitHub: Somente foram considerados repositórios hospedados no GitHub. Essa decisão se justifica pelo fato da plataforma ser amplamente adotada pela comunidade open-source, oferecendo uma API unificada que permite a extração estruturada de metadados de forma reprodutível. A utilização de uma única fonte reduz a variabilidade, padroniza a coleta e está em linha com estudos de mineração de repositórios de software.

- Mínimo de 100 estrelas: Repositórios incluídos deveriam possuir pelo menos 100 estrelas. Esse critério funciona como um indicador indireto de relevância, visibilidade e adoção do projeto, reduzindo a chance de que aplicações pouco utilizadas ou sem

tração comunitária fossem incorporadas ao dataset. Projetos amplamente reconhecidos tendem a refletir práticas mais próximas das adotadas em contextos reais de produção.

- Mínimo de três contribuidores: A exigência de ao menos três contribuidores distintos buscou evitar repositórios mantidos exclusivamente por uma única pessoa, cuja representatividade como aplicação real poderia ser limitada. Projetos com múltiplos contribuidores tendem a apresentar maior maturidade organizacional, diversidade de decisões arquiteturais e maior probabilidade de estarem inseridos em processos colaborativos típicos de sistemas distribuídos.

- Atividade recente: Apenas repositórios que apresentaram algum commit no período de até um ano anterior à mineração foram incluídos. Esse critério garante que as aplicações selecionadas estejam em uso ou manutenção ativa, alinhadas ao objetivo de identificar sistemas reais e contemporâneos. Projetos abandonados poderiam comprometer a representatividade do dataset e distorcer conclusões sobre ecossistemas atuais de microserviços.

- Aceitação de monorepos, microserviços isolados e plataformas baseadas em microserviços: Além dos requisitos estruturais e de atividade, este estudo também considerou diferentes formas de organização arquitetural que aparecem em aplicações reais baseadas em microserviços. Foram incluídos repositórios estruturados como monorepos, nos quais diversos serviços coexistem em um único código-fonte, bem como microserviços isolados pertencentes a aplicações multirepo. Essa flexibilidade busca refletir a variedade de práticas de versionamento adotadas na indústria, evitando que o dataset final fique restrito a um único modelo de organização.

- Documentação predominantemente em inglês: Somente foram incluídos repositórios cujo README e código estivessem predominantemente em inglês. Esse critério teve caráter prático: a análise manual e a interpretação arquitetural tornam-se mais consistentes quando realizadas em um idioma compartilhado pelos pesquisadores. Além disso, o inglês constitui o padrão de comunicação da maior parte dos projetos open-source.

Também foram incluídos repositórios que representam plataformas operacionais ou sistemas de suporte cujo funcionamento interno depende de múltiplos serviços independentes. Embora tais plataformas não se configurem como uma aplicação tradicional única, são compostas por componentes distribuídos que interagem entre si, implementando, de fato, princípios fundamentais da arquitetura de microserviços. Assim, mesmo quando sua finalidade é habilitar, gerenciar ou coordenar outras aplicações, a estrutura interna dessas plataformas atende aos critérios estabelecidos para este estudo.

3.2 Critérios de Exclusão

Após a aplicação dos critérios de inclusão, um segundo conjunto de critérios foi utilizado para identificar repositórios que, embora inicialmente elegíveis, não atendiam ao objetivo

de identificar aplicações reais estruturadas como microserviços.

- Ausência de README: Repositórios sem README foram excluídos imediatamente, pois a falta de documentação mínima impede a identificação do objetivo, escopo e arquitetura do projeto. A ausência desse arquivo compromete significativamente qualquer tentativa de análise e caracterização.
- Ausência de Dockerfile ou artefatos equivalentes: Foram excluídos repositórios que não apresentavam Dockerfile ou arquivos equivalentes de definição de contêineres. A conteinerização é hoje um elemento fundamental na implantação e operação de microserviços; portanto, a ausência desse artefato sugere que o repositório não representa um serviço executável de forma independente ou alinhado às práticas contemporâneas.
- Projetos experimentais ou didáticos: Repositórios classificados como experimentais — incluindo provas de conceito, exemplos de cursos, materiais didáticos e demonstrações — foram excluídos. Embora esses projetos frequentemente mencionem microserviços, sua finalidade pedagógica ou exploratória não reflete cenários reais de produção. A exclusão foi definida por meio de palavras-chave no README e análise manual.
- Ferramentas e bibliotecas que não implementam microserviços: Por fim, foram excluídos repositórios que, embora mencionassem microserviços em sua descrição, não implementavam essa arquitetura. Projetos como ferramentas, bibliotecas, frameworks ou soluções auxiliares — por exemplo, sistemas de tracing, gateways ou mecanismos de observabilidade — não configuram aplicações compostas por múltiplos serviços independentes e, portanto, não atendem ao foco deste estudo.

3.3 Fonte da Busca

O GitHub foi selecionado como fonte primária devido à sua ampla adoção na comunidade open source e ao grande volume de projetos ativos. Além disso, a plataforma fornece uma API que permite realizar buscas complexas, facilitando a aplicação de técnicas de MSR. Como controle para validação dos critérios, utilizou-se o Spinnaker, um sistema reconhecido na literatura como exemplo de aplicação real composta por diversos microserviços.

A busca pelos repositórios foi conduzida por meio do endpoint de Search Repositories da API oficial do GitHub, que permite criar consultas estruturadas utilizando operadores específicos. Segundo a documentação da GitHub REST API, uma query pode ser construída combinando palavras-chave, filtros por campos, operadores relacionais e parâmetros de ordenação.

3.4 Procedimento de Busca

A consulta final utilizada é composta pelos seguintes elementos, conforme descrito na documentação oficial:

- **Search keywords**: termos livres que descrevem o conteúdo desejado no repositório.

```
1    query = "microservices distributed"
```

- **in**: especifica onde os termos devem ser procurados (título, descrição ou README).

```
query = "microservices in:description"
```

- **sort**: define o atributo usado para ordenar resultados (e.g., stars, updated).

```
query = "microservices sort:stars"
```

- **stars**: filtra repositórios pelo número de estrelas utilizando operadores relacionais (=, ≥, ≤, >, <).

```
query = "microservices stars:>100"
```

- **pushed**: filtra por data de último commit, permitindo garantir que apenas projetos ativos sejam retornados.

```
query = "microservices pushed:>=2020-01-01"
```

Com isso, temos a seguinte query final:

```
query = "microservice OR service OR micro-service in:description sort:
stars stars:>=100 pushed:>={QUERY_DATE_THRESHOLD}"
```

Onde **QUERY_DATE_THRESHOLD** é definido como a data de 12 meses atrás, no formato YYYY-MM-DD:

```
QUERY_DATE_THRESHOLD = (datetime.now() - relativedelta(months=12)).
strftime("%Y-%m-%d")
```

3.4.1 Limitações da API do GitHub

A API restringe os resultados a, no máximo, 1.000 repositórios por consulta. Mesmo que a busca retorne mais resultados, apenas os primeiros 1.000 podem ser paginados e consultados pela API. Essa restrição influenciou a quantidade de repositórios encontrados, visto que, para aumentar o número de repositórios alcançados pela pesquisa, seria necessário ajustar os parâmetros da query de busca.

Ainda, o Github restringe a quantidade de requisições que podem ser realizadas por minuto e, portanto, foram necessárias duas estratégias para contornar esse problema:

- **Salvamento local de resultado de requisições**: Uma estratégia adotada para reduzir o número de requisições foi salvar localmente os resultados das fases de processamento que necessitavam que chamadas fossem feitas para a API. Com isso, foi possível reutilizar os dados de chamadas anteriores, assim como consultar a evolução dos repositórios durante o processamento.

```
def save_json(data, filename=FILENAME, folder=INITIAL_DATA_FOLDER,
timestamp=False, total_count=None):
  """
  Save a Python object (list/dict) as JSON .
  If timestamp=True, appends a date/time string to the filename.
  Includes total_count if provided.
```

```
    """
    if timestamp:
        ts = time.strftime("%Y-%m-%d_%H-%M-%S")
        name, ext = filename.rsplit('.', 1)
        filename = f"{name}_{ts}.{ext}"
    path = f"{folder.rstrip('/')}/{filename}"

    # Ensure the directory exists before saving
    os.makedirs(os.path.dirname(path), exist_ok=True)

    data_to_save = {
        "repos": data,
        "total_count": total_count
    }

    with open(path, 'w') as f:
        json.dump(data_to_save, f, indent=2)

    print(f"Saved JSON to: {path}")
    return path
```

Listing 1: Função para salvar o resultado das requisições

```
def load_json(filename=FILENAME, folder=INITIAL_DATA_FOLDER, newest=True):
    """
    Load a JSON file.
    If newest=True, automatically loads the most recently modified file
    matching the base filename (e.g., repos_*.json).
    Returns the data and total_count if available.
    """
    folder = folder.rstrip('/')
    base, ext = os.path.splitext(filename)

    # Ensure the folder exists before trying to list its contents
    if not os.path.exists(folder):
        raise FileNotFoundError(f"Folder '{folder}' not found. Please ensure data is mined or placed in the correct location.")

    if newest:
        candidates = [
            os.path.join(folder, f)
            for f in os.listdir(folder)
            if f.startswith(base) and f.endswith(ext)
        ]

        if not candidates:
            raise FileNotFoundError(f"No files matching '{base}*{ext}' found in {folder}")
        path = max(candidates, key=os.path.getmtime)
        print(f"Newest file detected: {os.path.basename(path)}")
    else:
        path = f"{folder}/{filename}"
```

```
28
29          with open(path, 'r') as f:
30              data_loaded = json.load(f)
31
32          print(f"Loaded JSON from: {path}")
33
34          if type(data_loaded) is dict and "repos" in data_loaded:
35            return data_loaded.get("repos"), data_loaded.get("total_count
    ")
36
37          return data_loaded, None
```

Listing 2: Função para carregar um arquivo salvo anteriormente

- **Função de sleep**: A outra estratégia adotada foi adicionar uma função para detectar quando a resposta da API indicava que o limite de requisições havia sido atingido e, a partir disso, aguardar para que o restante das requisições pudesse ser completado.

```
1     # A shared lock to ensure only one thread prints the rate limit
    warning and runs the tqdm progress bar
2     RATE_LIMIT_LOCK = threading.Lock()
3
4     def sleep_if_needed(response, fallback_delay=5):
5         """
6         Waits if the response indicates rate limit exceeded.
7         """
8         remaining = response.headers.get("X-RateLimit-Remaining")
9         reset_time = response.headers.get("X-RateLimit-Reset")
10
11        wait_seconds = 0
12        desc = ""
13
14        if remaining is not None and remaining == "0":
15            reset_timestamp = int(reset_time) if reset_time else None
16            wait_seconds = max(0, reset_timestamp - int(time.time()))
    if reset_timestamp else fallback_delay
17            desc = f"Rate limit hit. Waiting {wait_seconds} seconds..."
18        elif response.status_code == 403:
19            wait_seconds = fallback_delay
20            desc = f"Received 403 (possible rate limit). Waiting {
    fallback_delay} seconds..."
21
22        if wait_seconds > 0:
23            # Try to acquire the lock without blocking. Only the thread
     that gets the lock proceeds to print.
24            if RATE_LIMIT_LOCK.acquire(blocking=False):
25                try:
26                    # The first thread prints the message and runs the
    visual countdown.
27                    print(desc, flush=True)
28                    for _ in trange(wait_seconds, desc="Waiting", unit=
    "sec"):
29                        time.sleep(1)
30                finally:
```

```
31                      # Release the lock after waiting.
32                      RATE_LIMIT_LOCK.release()
33              else:
34                  # Other threads that couldn't acquire the lock will
    wait silently to respect the rate limit.
35                  time.sleep(wait_seconds)
```

Listing 3: Função para aguardar em caso de limite de requisições da API

3.4.2 Ordenação por Estrelas

Nessa primeira versão da string de busca (Seção 3.4), está ausente o parâmetro **sort** e, por isso, os resultados retornados estarão ordenados por **best_match**. A primeira execução completa do notebook foi realizada com essa query, que apresentou resultados satisfatórios, com repositórios presentes nos resultados finais. No entanto, essa string de busca não retorna nenhum dos repositórios da aplicação de controle **Spinnaker**, o que torna necessária a sua atualização com o parâmetro **sort**, presente na string final.

3.4.3 Processamento Paralelo

Para melhorar o desempenho do notebook Python, foram implementadas estratégias de processamento paralelo utilizando múltiplos workers, reduzindo o tempo de execução entre as iterações do processo de mineração.

3.5 Seleção Automatizada dos Repositórios

A seleção automatizada de repositórios foi dividida em cinco etapas distintas, cada uma com o objetivo de selecionar repositórios que atendam a algum dos requisitos propostos.

A primeira etapa ocorre logo após a coleta dos 1.000 repositórios retornados pela API do Github e consiste na extração das informações necessárias de cada repositório, além de solicitar à API o número de contribuidores de cada um. Na segunda etapa, os repositórios são filtrados pelo número mínimo de contribuidores. Em seguida, na terceira etapa, todos os repositórios que contiverem alguma palavra da lista de filtros de About e Topics são eliminados. De forma semelhante, a quarta etapa também filtra repositórios por meio de uma lista de palavras, mas agora esse filtro é aplicado no README do repositório. Essas duas etapas foram separadas porque a lista de palavras a serem buscadas no README precisa ser mais restrita, devido à maior probabilidade de encontrar falsos negativos ao aplicar o filtro no README. Por fim, a última etapa automática consiste em verificar a existência de arquivos Docker nos repositórios analisados.

3.5.1 Extração de Informações Necessárias

Nesta etapa, nenhum repositório é excluído, visto que o objetivo é apenas selecionar as informações necessárias para o estudo, além de requisitar à API a informação sobre o número de contribuidores.

```python
    def get_contributor_count(session: requests.Session, owner: str, repo: str, anon: bool = True, max_retries: int = 3):
        url = f"{GITHUB_BASE_URL}/repos/{owner}/{repo}/contributors?per_page=1&anon={'true' if anon else 'false'}"

        for _ in range(max_retries):
            r = session.get(url)
            if r.status_code == 200:
                if 'Link' in r.headers:
                    for part in r.headers['Link'].split(','):
                        if 'rel="last"' in part:
                            last_url = part[part.find('<') + 1:part.find('>')]
                            if 'page=' in last_url:
                                page_num = last_url.split('page=')[-1].split('&')[0]
                                try:
                                    return int(page_num)
                                except ValueError:
                                    pass
                try:
                    return len(r.json())
                except Exception:
                    return 0

            if r.status_code in (403, 502, 503, 504):
                sleep_if_needed(r)
                continue
            else:
                return 0
        return 0
```

Listing 4: Função para recuperar o número de contribuidores de um repositório

```python
    def worker(item):
        owner = item['owner']['login']
        repo = item['name']
        contribs = get_contributor_count(session, owner, repo)
        pushed_at = item.get('pushed_at')

        return {
            'name': repo,
            'full_name': item['full_name'],
            'repo_url': item['html_url'],
            'owner_name': owner,
            'owner_url': item['owner']['html_url'],
            'description': item.get('description'),
            'fork': item.get('fork'),
            'stars': item.get('stargazers_count', 0),
            'language': item.get('language'),
            'license_name': item['license']['name'] if item.get('license') els  'No license',
            'topics': item.get('topics', []),
            'forks': item.get('forks', 0),
```

```
            'open_issues': item.get('open_issues', 0),
            'default_branch': item.get('default_branch'),
            'score': item.get('score'),
            'contributors': contribs,
            'pushed_at': pushed_at,
        }
```

Listing 5: Função para selecionar apenas as informações necessárias dos repositórios

3.5.2 Filtrar Número de Contribuidores

Neste passo, são selecionados apenas repositórios cujo número de contribuidores, retornados pelo endpoint **GITHUB_BASE_URL/repos/{owner}/{repo}/contributors**, seja maior do que 2, em que **owner** e **repo** são variáveis correspondentes ao repositório desejado. A seguir, descrevemos a função que seleciona as informações necessárias e implementa o filtro.

```
    def process_repositories(session: requests.Session, items: list,
min_contributors: int = 3):
        print(40 * "=","Processing repositories...", 40 * "=", "\n")
        if PROCESS_INITIAL_REPOS:
          results = []
          total_items = len(items)
          max_workers = min(32, max(1, total_items))
          print(f"Using {max_workers} threads to process {total_items}
repositories")

          # Increase connection pool size to avoid warnings
          adapter = requests.adapters.HTTPAdapter(pool_connections=
max_workers, pool_maxsize=max_workers)
          session.mount('http://', adapter)
          session.mount('https://', adapter)

          def worker(item):
              owner = item['owner']['login']
              repo = item['name']
              contribs = get_contributor_count(session, owner, repo)
              pushed_at = item.get('pushed_at')

              return {
                  'name': repo,
                  'full_name': item['full_name'],
                  'repo_url': item['html_url'],
                  'owner_name': owner,
                  'owner_url': item['owner']['html_url'],
                  'description': item.get('description'),
                  'fork': item.get('fork'),
                  'stars': item.get('stargazers_count', 0),
                  'language': item.get('language'),
                  'license_name': item['license']['name'] if item.get('
license') else 'No license',
                  'topics': item.get('topics', []),
                  'forks': item.get('forks', 0),
```

```
                    'open_issues': item.get('open_issues', 0),
                    'default_branch': item.get('default_branch'),
                    'score': item.get('score'),
                    'contributors': contribs,
                    'pushed_at': pushed_at,
                }

        # Executor with manual counter
        with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers
) as executor:
            futures = [executor.submit(worker, it) for it in items]

            for fut in tqdm(concurrent.futures.as_completed(futures),
total=total_items, desc="Processing repos"):
              try:
                  res = fut.result()
                  if res['contributors'] >= min_contributors:
                      results.append(res)
              except Exception as e:
                  print(f"Error processing repository: {e}", flush=True)

        print(f"Processing complete! {len(results)} repositories added.",
flush=True) # Clear the counter line

        save_json(results, timestamp=USE_TIMESTAMP, folder=
PROCESSED_INITIAL_DATA_FOLDER, total_count=len(results))

        print(f"Repositories that passed the filter (>=3 contributors): {
len(results)}", flush=True)
        print("Data saved and ready for analysis!", flush=True)

        return results
      else:
        results, total = load_json(newest=True, folder=
PROCESSED_INITIAL_DATA_FOLDER)
        count = total if total is not None else len(results)

        print(f"Repositories that passed the filter (>=3 contributors): {
len(results)}", flush=True)
        print("Data saved and ready for analysis!\n", flush=True)


        return results
```

Listing 6: Função que seleciona informações necessárias e filtra por número de contribuidores

3.5.3 Filtrar About e Topics

Para encontrar a lista de palavras que seria utilizada neste passo e no seguinte, foi realizado um processo iterativo no qual, a cada palavra adicionada, eram analisados quantos repositórios eram removidos por essa palavra. Caso fosse um número relevante, a palavra era mantida no filtro. A lista de palavras não poderia remover todos os repositórios da

aplicação de controle Spinnaker, e ror isso, essa verificação também foi realizada. As palavras adicionadas visam identificar repositórios que não se encaixam na definição proposta de aplicação real em microserviços.

A lista final de palavras para o About e Topics foi:

```
DESC_AND_TOPICS_FILTER_WORDS = ["sample", "demo", "example", "sample-app",
                                "demo-app", "example-app", "workshop",
                                "hackathon", "course", "guide", "papers",
                                "paper", "CRUD", "template", "masterclass",
                                "setup", "toolkit", "framework", "library",
                                "boilerplate", "starter"]
```

```
def filter_repos_by_description_and_topics(repos):
    """
    Filters a list of repositories based on keywords in their description and topics.

    Args:
        repos: List of repositories (dictionaries).

    Returns:
        A tuple containing:
            - final_repositories: List of repositories that do not contain filter words.
            - wrong_repositories: List of repositories that contain filter words.
    """
    wrong_repositories = []
    final_repositories = []

    print(40 * "=","Applying description and topics filter...", 40 * "=", "\n")

    for repo in tqdm(repos, desc="Filtering descriptions"):
        is_wrong = False
        description = repo.get("description", "").lower()
        topics = repo.get("topics", [])

        for word in DESC_AND_TOPICS_FILTER_WORDS:
            if word in description or any(word in topic for topic in topics):
                wrong_repositories.append(repo)
                is_wrong = True
                break
        if not is_wrong:
            final_repositories.append(repo)

    print(f"Description and topics filter applied.")
    print(f"Repositories that passed the filter: {len(final_repositories)}")
    print(f"Repositories that were filtered out: {len(wrong_repositories)}\n")

```

```
35          return final_repositories, wrong_repositories
```

Listing 7: Função para filtrar repositórios por palavras no About e nos Topics

3.5.4 Filtrar pelo README

Semelhante ao passo anterior, o mesmo processo iterativo para a escolha de palavras foi aplicado neste passo. No entanto, a lista de palavras acabou sendo menor, pois muitas das palavras utilizadas anteriormente removeriam repositórios da aplicação de controle, como, por exemplo, a palavra **"example"**, presente em READMEs que apresentam trechos de código de exemplo. Além disso, devido a os arquivos README serem escritos em Markdown, muitas palavras acabam ocorrendo dentro de hyperlinks. Para resolver esse problema, foi necessário remover todos os hiperlinks dos arquivos README antes de aplicar a função de filtro.

```
def _readme_worker(session: requests.Session, repo: dict):
    full_name = repo['full_name']
    url = f"https://api.github.com/repos/{full_name}/readme"

    try:
      response = session.get(url, headers=GITHUB_HEADERS)
      sleep_if_needed(response)

      readme_content = ""
      if response.status_code == 200:
        content = response.json().get('content')
        if content:
          readme_content = base64.b64decode(content).decode('utf-8').lower()
          readme_without_links = re.sub(r'\([^()]*\)', '()', readme_content)
        elif response.status_code == 404:
          pass
        else:
          print(f"\nWarning: Could not decode README content for {full_name}.  Status code: {response.status_code}")
        is_wrong = any(w in readme_without_links for w in README_FILTER_WORDS)
        return repo, is_wrong
    except Exception as e:
      print(f"\nError processing README for {full_name}: {e}")
      return repo, False
```

Listing 8: Função auxiliar para carregar e limpar o conteúdo do README

```
def filter_repos_by_readme(session: requests.Session, repos, folder_final=README_DATA_FOLDER, folder_wrong=README_REMOVED_DATA_FOLDER, timestamp=USE_TIMESTAMP):
    """
    Filters a list of repositories based on keywords in their README content.

    Args:
        session: The requests session to use for API calls.
```

```
        repos: List of repositories (dictionaries).
        folder_final: The folder to save the final results.
        folder_wrong: The folder to save the filtered out results.
        timestamp: Boolean to append a timestamp to the filename.

    Returns:
        A tuple containing:
            - final_repositories: List of repositories that do not contain filter words.
            - wrong_repositories: List of repositories that contain filter words.
    """
    wrong_repositories = []
    final_repositories = []
    total_repos = len(repos)

    print(40 * "=","Applying README filter...", 40 * "=", "\n")

    if not FILTER_README: # If FILTER_README is False, load data
        try:
            # Load both final and wrong repositories
            final_repositories, _ = load_json(newest=True, folder=folder_final)
            wrong_repositories, _ = load_json(newest=True, folder=folder_wrong)

            print("Loaded filtered data.")
            print(f"Repositories that passed the filter: {len(final_repositories)}")
            print(f"Repositories that were filtered out: {len(wrong_repositories)}\n")
            return final_repositories, wrong_repositories
        except FileNotFoundError:
            print("Filtered data not found. Please set FILTER_README to True to run the filter.")
            return [], [] # Return empty lists if file not found and not filtering
    print("Applying README filter...")
    max_workers = min(32, max(1, total_repos))

    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [executor.submit(_readme_worker, session, repo) for repo in repos]

        for fut in tqdm(concurrent.futures.as_completed(futures), total=total_repos, desc="Filtering READMEs"):
            try:
                repo, is_wrong = fut.result()
                if is_wrong:
                    wrong_repositories.append(repo)
                else:
                    final_repositories.append(repo)
            except Exception as e:
```

```
                print(f"Error processing repository: {e}", flush=True)

    print(f"README filter applied.")
    print(f"Repositories that passed the filter: {len(final_repositories)}")
    print(f"Repositories that were filtered out: {len(wrong_repositories)}\n")

    if SAVE_JSON:
        save_json(final_repositories, folder=folder_final, filename="repos_readme_filter.json", timestamp=timestamp)
        save_json(wrong_repositories, folder=folder_wrong, filename="repos_readme_removed_filter.json", timestamp=timestamp)

    return final_repositories, wrong_repositories
```

Listing 9: Função para filtrar repositórios com base no conteúdo do README

3.5.5 Filtrar por Dockerfile

Por fim, o último passo da seleção automatizada foi a filtragem de repositórios que não possuíam Dockerfile. Para isso, foram feitas novas requisições à API do GitHub, com o intuito de obter todas as linguagens presentes em cada repositório. Como descrito na Seção 2.3, Docker se alinha diretamente com princípios da arquitetura de microserviços e, por conta disso, decidiu-se selecionar apenas repositórios que apresentem algum Dockerfile, de forma semelhante ao que foi feito em (D'ARAGONA et al., 2024). É muito pouco provável que um repositório que implemente um microsserviço real não tenha sua definição em contêineres.

```
def _language_worker(session: requests.Session, repo: dict):
    full_name = repo['full_name']
    url = f"{GITHUB_BASE_URL}/repos/{full_name}/languages"

    try:
      response = session.get(url, headers=GITHUB_HEADERS)
      sleep_if_needed(response)

      languages = {}
      if response.status_code == 200:
        languages = response.json()
      else:
        print(f"\nWarning: Could not fetch language data for {full_name}. Status code: {response.status_code}")

      if "Dockerfile" in languages:
        return repo
      return None

    except Exception as e:
      print(f"\nError processing language data for {full_name}: {e}")
      return None
```

Listing 10: Função para recuperar linguagens do repositório e filtrar aqueles sem Dockerfile

```python
def filter_repos_by_language(session: requests.Session, repos,folder=LANGUAGES_DATA_FOLDER, timestamp=USE_TIMESTAMP):
    """
    Filters a list of repositories to keep only those that have Dockerfile as a language.

    Args:
        session: The requests session to use for API calls.
        repos: List of repositories (dictionaries).
        folder: The folder to save the results.
        timestamp: Boolean to append a timestamp to the filename.

    Returns:
        A list of repositories that have Dockerfile as a language.
    """
    filtered_repos = []
    total_repos = len(repos)

    print(40 * "=","Applying language filter...", 40 * "=", "\n")

    if not FILTER_LANG: # If FILTER_LANG is False, load data
        try:
            filtered_repos, _ = load_json(newest=True, folder=folder, filename="repos_lang_filter.json")
            print(Loaded filtered data.")
            print(f"Repositories that passed the filter: {len(filtered_repos)}")
            return filtered_repos
        except FileNotFoundError:
            print("Filtered data not found. Please set FILTER_LANG to True to run the filter.")
            return [] # Return empty list if file not found and not filtering


    # If FILTER_LANG is True, proceed with filtering
    print("Applying language filter...")
    max_workers = min(32, max(1, total_repos))

    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [executor.submit(_language_worker, session, repo) for repo in repos]

        for fut in tqdm(concurrent.futures.as_completed(futures), total=total_repos, desc="Filtering languages"):
          try:
            repo = fut.result()
            if repo:
              filtered_repos.append(repo)
          except Exception as e:
            print(f"Error processing repository: {e}", flush=True)

    print(f"Language filter applied.")
```

```
46      print(f"Repositories that passed the filter: {len(filtered_repos)}")
47
48      if SAVE_JSON:
49          save_json(filtered_repos, folder=folder, filename="repos_lang_filter
        .json", timestamp=timestamp)
50
51      return filtered_repos
```

Listing 11: Função para filtrar repositórios que possuam Dockerfile entre suas linguagens

3.6 Seleção Manual dos Repositórios

Após a etapa de seleção automatizada, os repositórios restantes passaram por três novas etapas de seleção, agora manuais. A primeira etapa consistiu em uma filtragem por idioma. A segunda etapa consistiu em uma checagem manual do objetivo da aplicação. Por fim, a terceira etapa consistiu em uma análise arquitetural da aplicação. A seguir, resumimos essas etapas:

1. **Remover repositórios com idiomas diferentes do inglês** Nesta etapa, foram removidos todos os repositórios que apresentavam algum idioma diferente do inglês no README. Neste passo, não foi feita nenhuma investigação sobre o objetivo da aplicação ou sua arquitetura.

2. **Seleção de aplicações reais** Esta etapa serve como uma validação adicional dos filtros automatizados aplicados no README, nos Topics e no About. Nela, foram procuradas evidências sobre o objetivo da aplicação e sobre se o repositório poderia ser considerado uma aplicação real.

3. **Análise arquitetural do repositório** Esta etapa constitui uma análise mais aprofundada do repositório, buscando entender como sua arquitetura está estruturada e se ela pode ser considerada uma aplicação de microserviços. Para reduzir vieses, todos os repositórios que chegaram a essa etapa foram verificados por outros dois pesquisadores e, apenas se houvesse concordância entre os três, o repositório seria considerado para o dataset.

3.7 Extração de Metadados

Com os repositórios finais selecionados, foi necessária a extração de alguns metadados, tais como: existência de CI/CD, número de releases, data da última atualização, número de contribuidores, quantidade de microserviços, linguagem principal e se o projeto utiliza mono-repo ou multi-repo.

4 Resultados

A Tabela 1 apresenta o dataset de aplicações baseadas na arquitetura de microserviços, acompanhado de metadados extraídos para auxiliar pesquisadores na seleção das aplicações

mais adequadas para seus estudos. A primeira coluna mostra o repositório analisado, que corresponde ao repositório encontrado e selecionado pela mineração e não necessariamente a toda a aplicação.

É importante destacar que aplicações compostas por múltiplos repositórios, como é o caso do OpenStack, tiveram mais de um repositório identificado durante a etapa de MSR. Contudo, para evitar duplicidade e garantir que cada aplicação seja representada apenas uma vez no dataset, selecionou-se um único repositório por aplicação. Nesse sentido, a segunda coluna inclui a organização ou o projeto agregador dos demais repositórios.

Além do nome do repositório, a tabela apresenta informações sobre CI/CD, indicando se o repositório possui pipelines configurados para integração e entrega contínuas. A coluna Releases apresenta o número de versões publicadas, estimando a evolução do software ao longo do tempo. Já a coluna Última Atualização registra a data do commit mais recente, permitindo avaliar o nível de atividade atual do projeto.

O número de Contribuidores fornece uma métrica aproximada do engajamento da comunidade, enquanto a coluna microserviços representa a quantidade de serviços identificados na arquitetura observada, o que auxilia na seleção de projetos com graus variados de complexidade arquitetural. A coluna Linguagem informa a linguagem predominante no repositório, possibilitando que pesquisadores filtrem projetos de acordo com seus interesses. Por fim, a coluna Mono-repo/Multi-repo descreve a estratégia de organização adotada pela aplicação.

Tabela 1: Repositórios de microserviços

Repositório	Org or Core	CI/CD	Releases	Última Atualização	Contribuidores	Microserviços	Linguagem	Mono-repo Multi-repo
blockscout-rs	blockscout	Sim	157	03/12/2025	37	11	Rust	Mono-repo
bbox	bbox	Sim	16	11/10/2025	6	6	Rust	Mono-repo
convox	convox	Sim	169	08/12/2025	38	3	Go	Multi-repo
linkerd2-proxy	linkerd2	Sim	293	09/12/2025	58	5	Rust	Multi-repo
metaflow-service	metaflow	Sim	47	04/11/2025	23	3	Python	Mono-repo
trove	openstack	Sim	32	05/12/2025	247	6	Python	Multi-repo
paasta	Yelp	Sim	379	08/12/2025	189	N/A	Python	Mono-repo
corrosion	superfly	Sim	4	09/12/2025	18	8	Rust	Multi-repo
kayenta	spinnaker	Sim	140	08/04/2025	74	11	Java	Multi-repo

4.1 Visão Geral dos Repositórios

A seguir, apresentamos uma breve descrição de cada repositório que compõe o dataset final, destacando sua finalidade e papel no contexto das aplicações analisadas.

blockscout-rs: Repositório escrito em Rust que reúne os principais serviços do Blockscout, um explorador de blockchains compatíveis com EVM. Inclui componentes responsáveis pela indexação, consulta e exposição de dados on-chain.

bbox: Aplicação modular desenvolvida em Rust para processamento, análise e disponibilização de dados geoespaciais. O repositório agrupa diversos módulos que, em conjunto, formam o BBOX Server.

convox: Plataforma PaaS de código aberto voltada ao deploy e gerenciamento de aplicações conteinerizadas. O repositório concentra o runtime, a CLI e automações de infraestrutura necessárias ao funcionamento do Convox.

linkerd2-proxy: Proxy de alto desempenho escrito em Rust e utilizado como plano de dados do service mesh Linkerd. Atua como sidecar responsável por roteamento de tráfego, observabilidade e aplicação de políticas.

metaflow-service: Serviço de backend do Metaflow utilizado para rastreamento de metadata, controle de execuções e exposição de informações sobre pipelines de machine learning. Complementa a ferramenta principal mantida pela Netflix.

trove: Serviço Database-as-a-Service (DBaaS) do ecossistema OpenStack. Permite o provisionamento, gerenciamento e operação automatizada de bancos de dados em ambientes OpenStack.

paasta: Plataforma de deploy e operação de serviços conteinerizados desenvolvida pelo Yelp. Centraliza orquestração, CI/CD e definições de infraestrutura utilizadas em larga escala pela organização.

corrosion: Sistema distribuído escrito em Rust e desenvolvido pela Fly.io, oferecendo primitivas para construção de aplicações globais, incluindo replicação, coordenação e roteamento inteligente.

kayenta: Serviço integrante do Spinnaker dedicado à análise automatizada de canary releases. Implementa métricas, comparações estatísticas e mecanismos de tomada de decisão para implantações progressivas mais seguras.

4.2 Repositórios Conhecidos Não Retornados

Durante a execução deste estudo, observou-se que diversos repositórios amplamente reconhecidos na literatura sobre microserviços não foram recuperados pelo processo de mineração, apesar de sua importância histórica e de sua recorrência em trabalhos acadêmicos. Essa ausência está diretamente relacionada aos critérios definidos para a busca, em especial a exigência de atividade recente e a presença de artefatos compatíveis com práticas modernas de conteinerização.

Entre os repositórios não capturados destacam-se projetos da Netflix que influenciaram significativamente a adoção de microserviços na indústria, tais como **Conductor**, **Eureka** e **Hystrix**. Esses projetos, embora extremamente relevantes do ponto de vista histórico, apresentam baixa atividade recente, com contribuições majoritariamente antigas ou desen-

volvimento descontinuado. Como a query aplicada restringia os resultados a repositórios com atualizações dentro de um intervalo temporal específico, tais projetos não foram retornados pela API do GitHub.

Um caso semelhante ocorre com o **Choerodon**, uma plataforma empresarial que integra diversas ferramentas relacionadas à arquitetura de microserviços. Apesar de sua relevância e adoção em ambientes corporativos, o repositório principal não se enquadrou nos critérios de atualização recente exigidos pela consulta e, portanto, também não foi capturado.

Além disso, a filtragem pelo uso de conteinerização introduziu outras exclusões importantes. O **Zuul**, gateway desenvolvido pela Netflix e frequentemente citado em trabalhos sobre arquiteturas distribuídas, não foi selecionado por não apresentar um `Dockerfile` explícito em seu repositório. Considerando que a definição de contêineres é hoje um artefato amplamente consolidado em aplicações reais de microserviços, a ausência desse arquivo levou à remoção automática do Zuul durante a etapa de filtragem.

5 Ameaças à Validade

Tendo como base o estudo de (PETERSEN; GENCEL, 2013), foram elencadas possíveis ameaças à validade considerando uma visão pragmática.

Validade Externa. A capacidade de generalização do estudo está limitada ao contexto de repositórios de software livre. Nesse cenário, buscamos o tanto quanto possível, dadas as limitações da API do GitHub, recuperar o máximo de repositórios para um conjunto de análises automatizadas e manuais. O dataset de microserviços não abrange todas as possíveis implementações dessa arquitetura. No entanto, na tentativa de mitigar essa ameaça, não foi aplicado nenhum filtro relacionado a plataformas tecnológicas, exceto o Dockerfile, que é, de fato, o padrão para a conteinerização de microsserviços.

Validade de Constructo. Parte da análise dos repositórios possui caráter subjetivo, uma vez que envolve a análise manual da composição arquitetural desses repositórios. Como forma de mitigar essa ameaça, a escolha dos repositórios que compõem o dataset foi revisada por três pesquisadores, o que aumenta a confiabilidade do processo de seleção e acurácia na aplicação dos critérios pré-estabelecidos.

Validade Interna. Ao filtrar os repositórios a partir de parâmetros como número de estrelas, número de contribuidores e presença de arquivos Dockerfile no projeto, introduzimos uma ameaça à validade interna do estudo, pois passamos a estabelecer uma relação entre esses parâmetros e a relevância dos repositórios selecionados. Para mitigar essa ameaça, os parâmetros da query e das etapas de filtragem foram definidos de forma incremental, utilizando o Spinnaker como aplicação de controle e verificando se alterações nos filtros mantinham, até o final da execução do estudo, repositórios já reconhecidos como aplicações reais em microserviços.

Confiabilidade. A capacidade de reproduzir o estudo está limitada tanto às alterações naturais dos repositórios ao longo do tempo quanto ao caráter subjetivo da análise manual realizada por cada pesquisador. Para mitigar esse problema e aumentar a confiabilidade, a maior parte dos filtros aplicados aos repositórios retornados foi automatizada, garantindo que diferentes execuções do notebook em Python produzam resultados consistentes entre

si, ainda que haja variabilidade inerente aos repositórios analisados.

6 Considerações Finais

A partir do método de Mineração de Repositórios de Software, foi possível construir um dataset composto exclusivamente por aplicações reais baseadas na arquitetura de microserviços. Utilizando requisições à API do GitHub, processos de filtragem automatizados e etapas de análise e seleção manual, partimos de um total de 1.969 repositórios para chegar a um conjunto final de 9 aplicações que atendem aos critérios definidos, incluindo atividade recente, presença de conteinerização e evidências de uso em contextos reais. O dataset resultante está acompanhado de metadados que descrevem características relevantes para outros estudos, como o número de contribuidores, a quantidade de microserviços, a presença de CI/CD, a linguagem predominante e a organização em mono-repo ou multi-repo. Com base nessas informações, pesquisadores podem selecionar as aplicações que mais se alinham aos objetivos de seus estudos, semelhantemente ao estudo de (D'ARAGONA et al., 2024), porém com foco específico em aplicações com evidências de uso em produção. Como trabalhos futuros, é possível ampliar o dataset variando a query de busca e a fonte dos repositórios analisados, bem como extrair novos metadados que representem aspectos ainda não explorados neste estudo, como métricas de evolução, características de observabilidade ou informações mais detalhadas sobre o ambiente de execução das aplicações.

Referências

D'ARAGONA, D. A. et al. A dataset of microservices-based open-source projects. In: *Proceedings of the 21st International Conference on Mining Software Repositories*. [S.l.: s.n.], 2024. p. 504–509.

FRANCESCO, P. D.; LAGO, P.; MALAVOLTA, I. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, v. 150, p. 77–97, 2019. Disponível em: ⟨https://doi.org/10.1016/j.jss.2019.01.001⟩.

JARAMILLO, D.; NGUYEN, D. V.; SMART, R. Leveraging microservices architecture by using docker technology. In: *IEEE SoutheastCon 2016*. Norfolk, VA, USA: IEEE, 2016. p. 1–5. Disponível em: ⟨https://ieeexplore.ieee.org/document/7506647⟩.

KIM, S. et al. An empirical study of just-in-time defect prediction in open source software. In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. [S.l.]: ACM, 2014. p. 182–191. ISBN 978-1-4503-2883-8.

LEWIS, J.; FOWLER, M. *Microservices*. 2014. Disponível em: ⟨https://martinfowler.com/articles/microservices.html⟩.

NAGAPPAN, N.; BALL, T.; ZELLER, A. Using history to improve software fault prediction models. In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. [S.l.]: ACM, 2014. p. 88–97. ISBN 978-1-4503-2883-8.

PETERSEN, K.; GENCEL, C. Worldviews, research methods, and their relationship to validity in empirical software engineering research. In: *2013 Joint Conference of the 23rd International Workshop on Software Measurement (IWSM) and the Eighth International Conference on Software Process and Product Measurement (Mensura)*. [S.l.: s.n.], 2013. Kai Petersen: School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden; Cigdem Gencel: Facult of Computer Science, Free University of Bolzano/Bozen, Italy.

RAY, B. et al. A large-scale study of programming languages and code quality in github. In: *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR 2012)*. [S.l.]: IEEE, 2012. p. 90–99. ISBN 978-1-4673-1761-0.

REISINGER, M. et al. The issue of monorepo and polyrepo in large enterprises. In: *International Conference on Software Engineering Advances*. [S.l.: s.n.], 2019.

SAHA, R. K. et al. Understanding the evolution of type-3 clones: An exploratory study. In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*. [S.l.]: IEEE Computer Society, 2013. p. 139–148. ISBN 978-1-4673-2936-1.

VIDONI, M. A systematic process for mining software repositories: Results from a systematic literature review. *Information and Software Technology*, Elsevier, v. 144, p. 106791, 2022. Disponível em: ⟨https://doi.org/10.1016/j.infsof.2021.106791⟩.