PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Campus Coração Eucarístico

Curso de Engenharia de Software - 6º Período - Noite

Disciplina: Laboratório de Experimentação de Software
Professor: Danilo de Quadros Maia Filho

Análise de Características de Repositórios Populares no Github

Luiz Filipe Nery Costa
Wilken Henrique Moreira

Sumário

1 Introdução

1.1 Contextualização

Repositórios open-source têm ganhado grande visibilidade nos últimos anos, não apenas como ferramentas de apoio ao desenvolvimento de software, mas também como referência de boas práticas colaborativas e de inovação tecnológica. Plataformas como o GitHub concentram milhares de projetos de diferentes áreas, que variam desde bibliotecas pequenas até sistemas amplamente utilizados por empresas e desenvolvedores em todo o mundo.

1.2 Objetivo do Estudo

Neste laboratório, buscamos compreender as principais características dos repositórios mais populares do GitHub, analisando aspectos relacionados à sua maturidade, atividade e colaboração. Entre os pontos de interesse estão a idade dos projetos, a frequência de contribuições externas, a regularidade de lançamentos de novas versões, o intervalo entre atualizações, as linguagens predominantes e a taxa de resolução de problemas (issues).

1.3 Hipóteses Iniciais

De maneira preliminar, levantamos as seguintes hipóteses:

1. Repositórios com maior número de estrelas tendem a ser mais maduros, por já terem conquistado estabilidade e confiança da comunidade;
2. Tais repositórios atraem alta contribuição externa, uma vez que sua visibilidade aumenta o engajamento;
3. Projetos populares mantêm ciclos regulares de releases e atualizações, refletindo preocupação com evolução contínua;
4. São desenvolvidos majoritariamente em linguagens consolidadas no mercado, como JavaScript, Python e Java;
5. Apresentam boa taxa de fechamento de issues, indicando gestão ativa da comunidade.

2 Objetivos

O presente trabalho tem como objetivo geral analisar as características dos 1.000 repositórios open-source mais populares do GitHub, a partir de métricas de maturidade, colaboração e manutenção. Para tanto, a investigação será guiada pelas seguintes questões de pesquisa.

2.1 Questões de Pesquisa Principais

Busca-se responder às seguintes questões centrais sobre as características de sistemas populares:

- **RQ01.** Os sistemas populares são maduros/antigos?
  Métrica: idade do repositório (a partir da data de criação).
- **RQ02.** Os sistemas populares recebem muita contribuição externa?
  Métrica: total de pull requests aceitas.
- **RQ03.** Os sistemas populares lançam releases com frequência?
  Métrica: total de releases.
- **RQ04.** Os sistemas populares são atualizados com frequência?
  Métrica: tempo até a última atualização.
- **RQ05.** Os sistemas populares são escritos nas linguagens mais utilizadas?
  Métrica: linguagem primária do repositório.
- **RQ06.** Os sistemas populares possuem um alto percentual de issues fechadas?
  Métrica: razão entre número de issues fechadas e total de issues.

2.2 Questão de Pesquisa Bônus

Adicionalmente, como objetivo complementar, investiga-se como essas características variam conforme a linguagem de programação:

- **RQ07.** Sistemas escritos em linguagens mais populares recebem mais contribuição externa, lançam mais releases e são atualizados com mais frequência?

3 Referencial Teórico

3.1 Mineração de Repositórios de Software

A área de *Mining Software Repositories* (MSR) explora dados de sistemas de controle de versão, trackers de issues e comunicações entre desenvolvedores para compreender aspectos como evolução, qualidade e defeitos de software. Revisões sistemáticas destacam processos estruturados para a condução de pesquisas nessa área, fornecendo uma base sólida para a análise empírica de software (FELIPE; KÁSSIO et al., 2018). No contexto deste trabalho, a MSR fornece a principal base metodológica para analisar os 1.000 repositórios mais populares do GitHub, possibilitando a coleta e interpretação de métricas de popularidade, maturidade e engajamento comunitário.

3.2 Software Open Source

O movimento *Open Source Software* (OSS) tem impacto significativo na engenharia de software, sendo responsável por grande parte da inovação tecnológica. A popularidade e o valor gerado por esses projetos reforçam a relevância de investigar as características dos repositórios mais bem-sucedidos do GitHub, já que o OSS se consolidou como base para o desenvolvimento moderno de software.

3.3 Git e GitHub

O Git é um sistema de controle de versão distribuído amplamente adotado, enquanto o GitHub atua como uma plataforma social que amplia as capacidades de colaboração, rastreabilidade e transparência dos projetos. Relatos práticos destacam como o GitHub facilita a coordenação entre múltiplos desenvolvedores, ajudando a superar desafios comuns em projetos de pesquisa e engenharia (MOZILLA SCIENCE GROUP, 2017). No escopo deste estudo, o GitHub é a fonte primária de dados, oferecendo métricas relevantes como número de estrelas, issues, *pull requests* e releases.

3.4 API do GitHub

O GitHub disponibiliza duas APIs principais para integração: REST e GraphQL. A documentação oficial apresenta as diferenças entre ambas, destacando limitações, estrutura de payload e casos de uso (GITHUB DOCS, 2024b). Desde 2016, a API GraphQL passou a oferecer maior flexibilidade e tipagem forte para consultas complexas (GITHUB DOCS, 2024a). Este estudo adota a API GraphQL para a coleta de dados devido à sua maior eficiência e capacidade de customização das consultas.

3.5 REST e GraphQL

REST e GraphQL representam dois paradigmas distintos de design de APIs. Enquanto REST é amplamente difundido por sua simplicidade, o GraphQL busca eficiência ao reduzir problemas de *over-fetching* e permitir consultas específicas. Comparações técnicas ressaltam os trade-offs entre as abordagens (ISAIAH, 2021), e guias de migração do próprio GitHub ilustram como reescrever chamadas REST em GraphQL para otimizar o desempenho (GITHUB DOCS, 2024c). No contexto deste trabalho, o uso do GraphQL é essencial para consultar grandes volumes de dados de maneira eficiente.

3.6 Ambientes Virtuais em Python (venv)

A reprodutibilidade é um princípio central em pesquisas empíricas. Ambientes virtuais em Python, como o `venv`, permitem isolar dependências e garantir a consistência dos

resultados. Estudos mostram que o uso de ambientes controlados é fundamental para que experimentos sejam totalmente replicáveis (MONDELLI et al., 2018). Neste estudo, a utilização de `venv` assegura a reprodutibilidade dos scripts de coleta e análise de dados dos repositórios GitHub.

3.7 Python para Análise de Dados em Engenharia de Software

O Python se consolidou como linguagem de apoio científico devido à sua simplicidade e ao vasto ecossistema de bibliotecas para análise de dados. Obras de referência como (MCKINNEY, 2022) descrevem técnicas de manipulação e visualização com `pandas` e `NumPy`, enquanto conteúdos práticos detalham módulos para exploração e visualização com `matplotlib` (MICROSOFT LEARN, 2023). Frameworks como Anaconda também reforçam o papel do Python em processos de limpeza, análise estatística e visualização de dados em larga escala (ANACONDA INC., 2023). Para este estudo, o Python será utilizado como linguagem principal na coleta, processamento e análise dos dados.

4 Definição das Métricas

Para responder às questões de pesquisa propostas, foram definidas as seguintes métricas, extraídas diretamente da API do GitHub para cada um dos 1.000 repositórios analisados.

4.1 M1: Idade do Repositório (RQ01)

A maturidade de um projeto foi medida por sua idade, calculada como a diferença entre a data da coleta dos dados (Agosto de 2025) e a data de criação do repositório. O valor final é expresso em anos.

4.2 M2: Total de Pull Requests Aceitas (RQ02)

Para medir a contribuição externa, foi contabilizado o número total de *pull requests* que foram mescladas (*merged*) ao ramo principal do projeto ao longo de toda a sua existência.

4.3 M3: Total de Releases (RQ03)

A frequência de lançamentos foi quantificada pelo número total de *releases* oficiais publicadas na página do repositório.

4.4 M4: Tempo desde a Última Atualização (RQ04)

Para avaliar se o projeto está ativo, calculou-se o tempo decorrido entre a data da coleta e a data do último *commit* ou atualização no repositório. A métrica é expressa em dias.

4.5 M5: Linguagem Primária (RQ05)

A popularidade da tecnologia foi determinada pela linguagem de programação primária associada a cada repositório, conforme a classificação automática realizada pelo GitHub.

4.6 M6: Razão de Issues Fechadas (RQ06)

A eficiência na gestão de problemas foi calculada pela razão entre o número de *issues* fechadas e o número total de *issues* (abertas e fechadas). A fórmula utilizada foi:

$$\text{Razão} = \frac{\text{Issues Fechadas}}{\text{Issues Abertas} + \text{Issues Fechadas}}$$

O resultado é um valor no intervalo [0, 1], onde valores mais próximos de 1 indicam maior eficiência na resolução de issues.

5 Metodologia

5.1 Ambiente de Desenvolvimento e Arquitetura

O projeto de software para a realização deste estudo foi desenvolvido com foco em modularidade, reprodutibilidade e testabilidade. A seguir, detalha-se a arquitetura do repositório e as tecnologias empregadas.

5.1.1 Linguagem e Ambiente

A linguagem de programação utilizada foi o **Python 3.10**, escolhida devido ao seu vasto ecossistema de bibliotecas para análise de dados e automação de tarefas web. Para garantir a **reprodutibilidade** do experimento, o projeto foi desenvolvido em um ambiente virtual (`venv`), com todas as dependências externas documentadas no arquivo `requirements.txt`.

5.1.2 Bibliotecas Utilizadas

Para a implementação do projeto, foram utilizadas as seguintes bibliotecas:

- **requests:** Utilizada para a comunicação HTTP com as APIs do GitHub (REST e GraphQL).
- **python-dotenv:** Empregada para gerenciar variáveis de ambiente, como o token de autenticação da API, de forma segura e separada do código-fonte.
- **pandas** e **numpy:** Essenciais para a etapa de análise, sendo o Pandas utilizado para a manipulação e estruturação dos dados em DataFrames e o NumPy para operações numéricas subjacentes.

- **matplotlib** e **seaborn:** Usadas em conjunto para a geração das visualizações e gráficos que compõem a análise dos resultados, salvos no diretório `output/plots/`.
- **pytest:** Adotado como framework de testes para validar componentes críticos do sistema, como a conectividade com a API e a lógica de consultas, garantindo a robustez do coletor de dados.

5.1.3 Arquitetura do Repositório de Análise

O projeto foi organizado em uma estrutura de diretórios que separa as responsabilidades, facilitando a manutenção e a clareza do código:

- `main.py`: Ponto de entrada do sistema, responsável por orquestrar a execução dos módulos de coleta, análise, visualização e geração do relatório final.
- `src/`: Diretório principal do código-fonte, organizado em módulos com responsabilidades bem definidas:
  - `collectors/`: Contém a lógica de extração de dados. Foram implementados dois coletores (`graphql_collector.py` e `rest_collector.py`), oferecendo dois caminhos possíveis para a coleta. Para este trabalho, conforme exigido, foi utilizado exclusivamente o `graphql_collector.py`.
  - `modules/`: Abriga os módulos que processam os dados coletados, como `data_analyzer.py` (cálculos estatísticos), `data_visualizer.py` (geração de gráficos) e `report_generator.py` (criação de relatórios).
  - `config.py`: Centraliza a configuração do projeto, como a leitura de variáveis de ambiente.
- `test/`: Contém os testes automatizados que garantem a qualidade e o funcionamento esperado dos componentes do sistema.
- `output/`: Diretório de destino para todos os artefatos gerados pela execução do script, incluindo os dados brutos (`top_10_repos.csv`), os gráficos (`.png`) e os relatórios de análise (`.md`).

subsectionProcesso de Coleta de Dados

A coleta de dados dos 1.000 repositórios mais populares do GitHub foi realizada de forma automatizada, seguindo um processo estruturado em quatro etapas principais, detalhadas a seguir.

1. **Autenticação:** Para interagir com a API do GitHub de forma eficiente, foi utilizado um token de acesso pessoal (*Personal Access Token*). Este método de autenticação permite realizar um número maior de requisições por hora, evitando os limites de taxa mais restritos aplicados a chamadas não autenticadas.

2. **Construção da Query GraphQL:** Foi elaborada uma consulta GraphQL específica para este estudo. A query foi projetada para buscar, em uma única chamada por repositório, todos os campos necessários para o cálculo das métricas, incluindo a data de criação, contagem total de *pull requests* mescladas, total de *releases*, e as contagens de *issues* abertas e fechadas.

3. **Execução e Paginação:** A busca dos 1.000 repositórios foi implementada através de um laço de repetição. A cada iteração, o script utilizou o mecanismo de paginação da API GraphQL, passando o cursor `endCursor` obtido na resposta anterior para solicitar a próxima "página"de resultados, até que o número total de repositórios desejado fosse atingido.

4. **Armazenamento:** Ao final do processo de coleta, os dados brutos foram processados, estruturados e salvos em um arquivo no formato CSV (*Comma-Separated Values*), nomeado `repositories.csv`. A escolha pelo formato CSV visa facilitar a importação e manipulação dos dados na fase de análise com a biblioteca Pandas.

5.2 Processo de Análise dos Dados

Após a coleta, os dados brutos foram sistematicamente processados e analisados para extrair as informações necessárias para responder às questões de pesquisa. Este processo seguiu três etapas principais.

1. **Carregamento e Limpeza dos Dados:** O arquivo `repositories.csv`, gerado na etapa anterior, foi carregado em um DataFrame utilizando a biblioteca Pandas. Em seguida, foi realizada uma etapa de pré-processamento e limpeza, que incluiu o tratamento de valores nulos (como repositórios sem uma linguagem primária definida) e a conversão de colunas textuais de data para o tipo `datetime`, formato adequado para cálculos temporais.

2. **Extração e Cálculo das Métricas:** Com os dados limpos e devidamente formatados, as métricas definidas para o estudo foram calculadas. Colunas existentes no DataFrame foram utilizadas para criar novas colunas correspondentes a cada métrica. A `Idade do Repositório`, por exemplo, foi calculada subtraindo a data de criação de cada projeto da data de execução da análise (27 de agosto de 2025). De forma análoga, outras métricas, como a razão de *issues* fechadas, foram calculadas a partir das colunas de dados brutos.

3. **Análise Estatística e Geração de Gráficos:** Para responder a cada Questão de Pesquisa (RQ), foram aplicadas as seguintes análises estatísticas:

    - **Mediana:** Calculada para as métricas quantitativas (RQs 01, 02, 03, 04 e 06), por ser uma medida de tendência central robusta a valores extremos (*outliers*).

- **Contagem de Frequência:** Utilizada para a métrica categórica (RQ05 - Linguagem Primária), a fim de identificar as tecnologias mais recorrentes.

Adicionalmente, optou-se por uma abordagem automatizada para a visualização dos dados. O mesmo script de análise foi responsável por gerar, com o auxílio das bibliotecas Matplotlib e Seaborn, um conjunto de gráficos (histogramas, gráficos de barras, etc.) para cada métrica. Essa automação permitiu não apenas complementar a análise estatística com uma interpretação visual, mas também garantir a consistência e reprodutibilidade das visualizações geradas.

5.3 Desafios Metodológicos e Contramedidas

Durante a execução do projeto, diversos desafios técnicos e conceituais foram encontrados. A seguir, são detalhados os principais obstáculos e as estratégias adotadas para superá-los.

- **Complexidade do GraphQL e Prazos do Projeto:** A API GraphQL do GitHub, apesar de poderosa, apresenta uma curva de aprendizado mais acentuada em comparação com a API REST, especialmente na montagem de consultas aninhadas e na compreensão do seu esquema de tipos. Considerando os prazos do projeto, essa complexidade representava um risco. Para mitigar esse risco, foi desenvolvida uma abordagem secundária utilizando a API REST (`rest_collector.py`). Embora o coletor GraphQL tenha sido o principal método utilizado para cumprir os requisitos do laboratório, a implementação REST serviu como um *backup* funcional e um ponto de referência para a validação dos dados.

- **Gerenciamento de Limites de Taxa (Rate Limiting):** A API do GitHub impõe um limite de requisições para evitar abusos. A API GraphQL, em particular, utiliza um sistema de "pontos"onde consultas mais complexas consomem mais pontos. O principal desafio foi duplo:

  1. *Eficiência da Query:* A consulta GraphQL foi cuidadosamente otimizada para buscar um grande volume de informações heterogêneas (dados do repositório, contagens de PRs, issues, etc.) em uma única chamada, o que consumiria dezenas de requisições na API REST. Isso tornou o processo extremamente eficiente em termos de consumo de pontos do *rate limit*.
  2. *Limite de 1000 Itens por Busca:* A API do GitHub não retorna mais de 1000 resultados para uma única busca. Para coletar os 1000 repositórios, foi essencial implementar corretamente a lógica de paginação, fazendo múltiplas chamadas sequenciais (ex: 10 chamadas buscando 100 repositórios cada) e utilizando o cursor `endCursor` para navegar entre as "páginas"de resultados.

- **Gerenciamento de Credenciais:** O acesso à API exigiu o uso de um token pessoal, uma credencial sensível. Para evitar expor o token no código-fonte, foi utilizada a biblioteca `python-dotenv`, que carrega a credencial a partir de um arquivo local (`.env`) não versionado, uma prática recomendada para a segurança de projetos de software.

- **Consistência e Limpeza dos Dados:** Os dados retornados pela API nem sempre eram consistentes. Foram encontrados casos de repositórios sem linguagem primária definida, com o sistema de *issues* desabilitado ou sem nenhuma *release* publicada. Esses casos exigiram a implementação de rotinas de tratamento de dados nulos e de validação na etapa de limpeza para garantir que a análise estatística não fosse distorcida ou interrompida por erros.

6 Análise Experimental

Nesta seção, são apresentados os resultados obtidos a partir da análise dos dados coletados dos 1.000 repositórios com maior número de estrelas no GitHub. Cada subseção a seguir corresponde a uma das questões de pesquisa (RQs) definidas, exibindo os valores estatísticos e as visualizações de dados pertinentes.

6.1 RQ01: Maturidade dos Repositórios Populares

Para investigar se sistemas populares são maduros (antigos), analisou-se a idade de cada repositório. A análise estatística revelou que a **mediana da idade dos repositórios é de 8,4 anos**.

A Figura 1 ilustra a distribuição da idade dos projetos. O histograma demonstra uma ampla variedade na maturidade dos repositórios, enquanto o box plot evidencia que 50% dos projetos analisados possuem entre aproximadamente 5 e 11 anos de existência, reforçando a tendência de que a popularidade está associada a projetos bem estabelecidos.

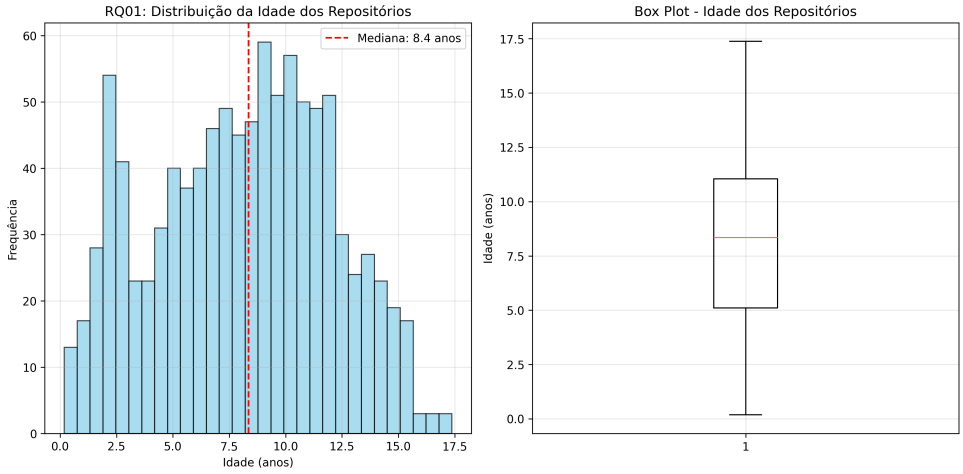


Figura 1: Distribuição da idade dos repositórios analisados, com histograma e box plot. A linha tracejada no histograma indica a mediana de 8,4 anos.

6.2 RQ02: Contribuição Externa em Sistemas Populares

A questão sobre o volume de contribuição externa foi respondida analisando o número total de *pull requests* aceitas. O valor encontrado para a **mediana de pull requests aceitas foi de 710**. Este valor indica que um projeto popular típico já recebeu centenas de contribuições da comunidade.

A Figura 2 oferece uma visão mais detalhada. O histograma à esquerda, com sua escala de frequência logarítmica, revela uma distribuição de dados extremamente assimétrica: a grande maioria dos repositórios possui um número de contribuições próximo à mediana, enquanto um pequeno grupo de projetos atinge dezenas de milhares de PRs. O gráfico de barras à direita destaca esses outliers, como o repositório `first-contributions`, que sozinho acumula mais de 80.000 contribuições aceitas.

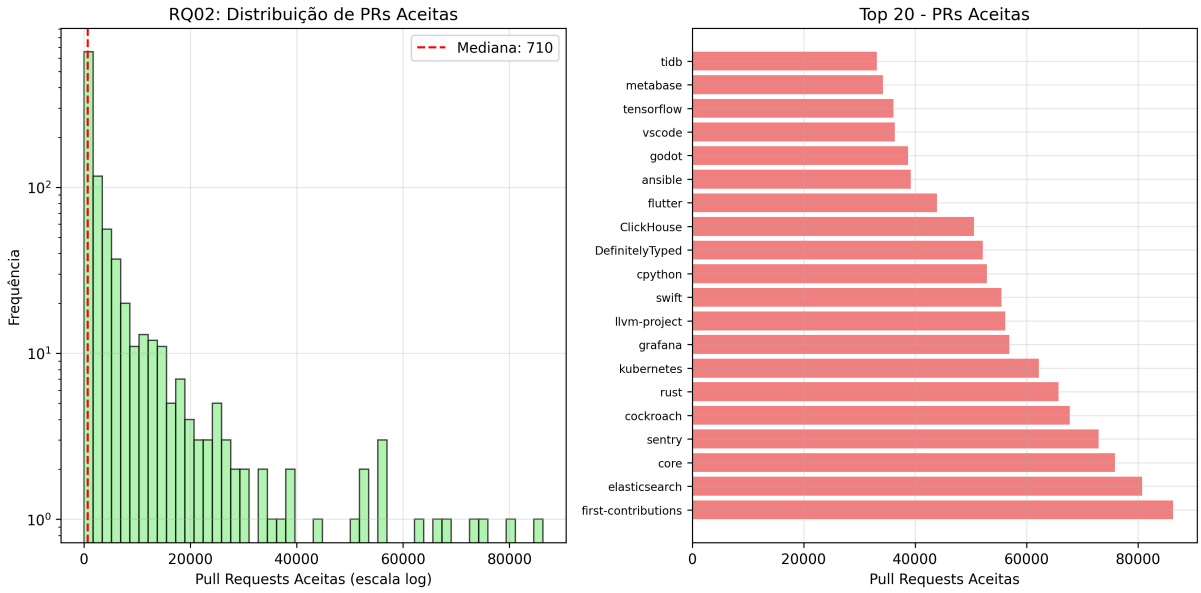


Figura 2: Distribuição de pull requests (PRs) aceitas. À esquerda, o histograma com a mediana indicada. À direita, o ranking dos 20 repositórios com maior número de PRs.

6.3 RQ03: Frequência de Lançamentos (Releases)

Para avaliar a frequência com que sistemas populares lançam novas versões, foi contabilizado o total de *releases* de cada repositório. A análise dos dados mostrou que a **mediana do total de releases é de 36**. Este valor sugere que um projeto popular típico possui um histórico considerável de lançamentos de versões estáveis.

A Figura 3 aprofunda a análise. O histograma à esquerda evidencia uma forte concentração de projetos com menos de 100 releases, indicando que, embora a maioria dos projetos populares publique versões, poucos o fazem com extrema frequência. O gráfico de dispersão à direita, que compara o número de releases com o de estrelas, não revela uma correlação clara. Observa-se que projetos com um número altíssimo de estrelas podem ter poucas releases e vice-versa, sugerindo que uma frequência de lançamento massiva não é um pré-requisito para se atingir alta popularidade.

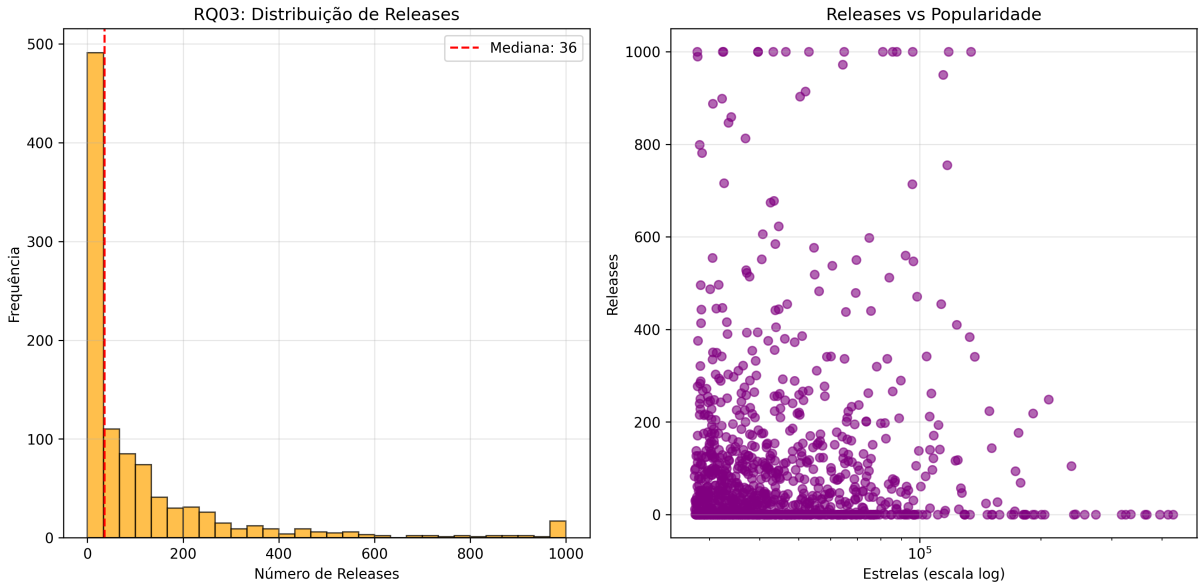


Figura 3: À esquerda, histograma da distribuição do número de releases, com mediana de 36 indicada. À direita, gráfico de dispersão comparando releases e estrelas.

6.4 RQ04: Frequência de Atualizações

A atividade recente dos repositórios foi medida pelo tempo decorrido desde a última atualização. A análise estatística apresentou uma **mediana de -1 dia**. Este valor, embora contraintuitivo, indica que mais da metade dos repositórios foi atualizada no mesmo dia da coleta de dados, sendo o resultado negativo um artefato do cálculo de diferença de datas com precisão de horas. Para fins práticos, a mediana pode ser considerada como 0 dias.

Este achado é reforçado de maneira conclusiva pela Figura 4. O gráfico de pizza à direita demonstra que **100% dos repositórios populares analisados foram atualizados no último mês**. Este resultado expressivo sugere que a manutenção ativa e constante é uma característica praticamente universal entre os projetos de maior destaque no GitHub.

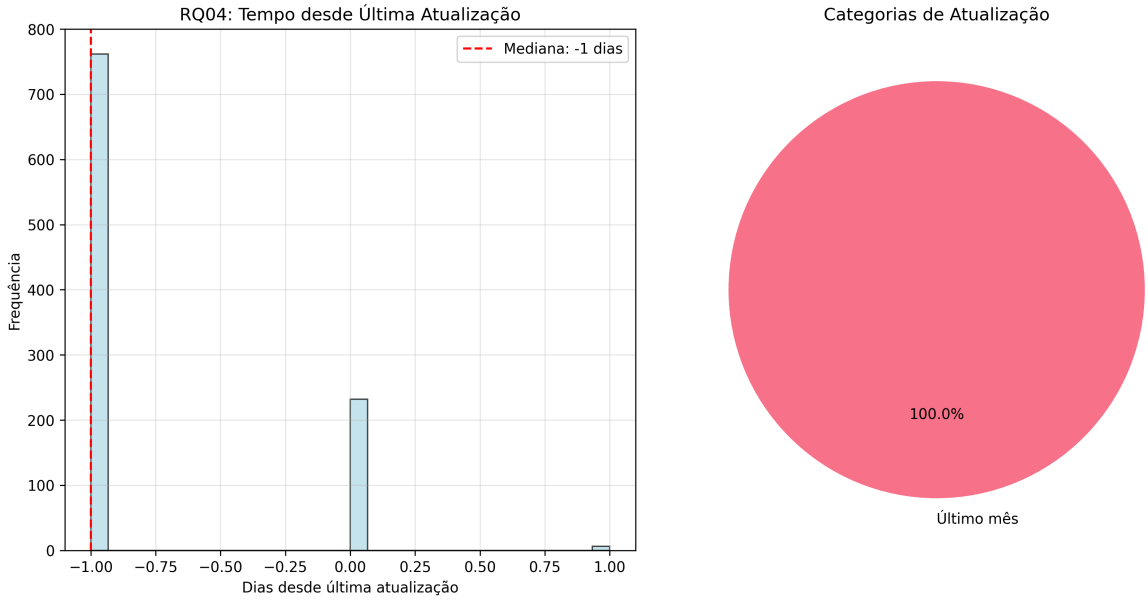
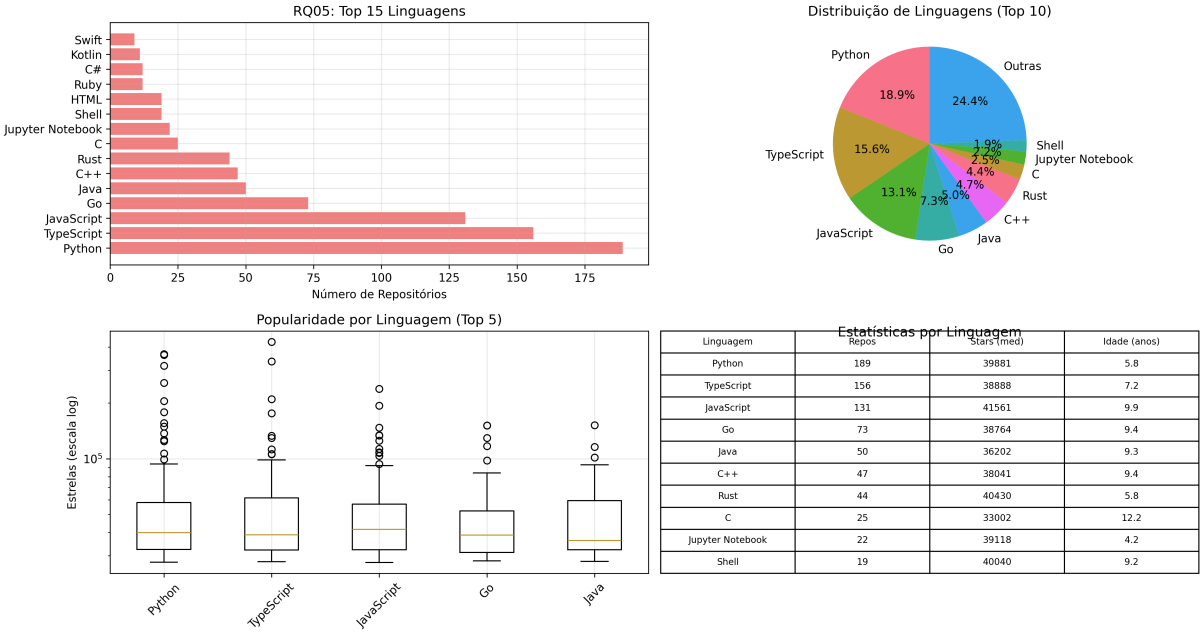


Figura 4: À esquerda, histograma do tempo desde a última atualização, com mediana de -1 dia. À direita, gráfico de pizza mostrando que 100% dos projetos foram atualizados no último mês.

6.5 RQ05: Linguagens de Programação Predominantes

A análise da linguagem primária dos repositórios buscou identificar as tecnologias mais utilizadas em projetos populares. A contagem de frequência, detalhada na Figura 5, revelou um predomínio de **Python** (189 repositórios), **TypeScript** (156 repositórios) e **JavaScript** (131 repositórios). Juntas, estas três linguagens representam quase metade (47,6

A Figura 5 apresenta um painel completo da análise. O gráfico de barras (superior esquerdo) ranqueia as 15 linguagens mais comuns. O gráfico de pizza (superior direito) ilustra a distribuição percentual, destacando que um grupo diversificado de tecnologias compõe a categoria "Outras"(24,4%). O box plot (inferior esquerdo) compara a popularidade (número de estrelas) entre as cinco principais linguagens, mostrando que todas possuem medianas de estrelas muito altas e distribuições similares. Por fim, a tabela (inferior direito) resume estatísticas importantes, confirmando os totais de repositórios e mostrando a mediana de idade e de estrelas para cada tecnologia principal.



Linguagem	Repos	Stars (med)	Idade (anos)
Python	189	39881	5.8
TypeScript	156	38888	7.2
JavaScript	131	41561	9.9
Go	73	38764	9.4
Java	50	36202	9.3
C++	47	38041	9.4
Rust	44	40430	5.8
C	25	33002	12.2
Jupyter Notebook	22	39118	4.2
Shell	19	40040	9.2

Figura 5: Painel de análise das linguagens de programação. No sentido horário, a partir do canto superior esquerdo: ranking das 15 linguagens mais frequentes; distribuição percentual das 10 principais; tabela de estatísticas por linguagem; e box plot da popularidade (estrelas) para as 5 principais.

6.6 RQ06: Percentual de Issues Fechadas

Para avaliar a eficiência na gestão de problemas, foi calculada a razão de *issues* fechadas em relação ao total de *issues* abertas e fechadas. A análise revelou que a **mediana para a taxa de fechamento de issues foi de 86,76%**. Este valor elevado indica que, tipicamente, a grande maioria das issues submetidas em projetos populares é eventualmente resolvida e fechada.

A Figura 6 detalha este comportamento. O histograma à esquerda mostra uma distribuição fortemente concentrada no lado direito do gráfico, com a maioria dos repositórios apresentando taxas de fechamento superiores a 80%. O gráfico de dispersão à direita, por sua vez, não demonstra uma correlação clara entre o volume total de issues e a taxa de fechamento. Isso sugere que mesmo projetos com um número massivo de issues (acima de 10.000) são capazes de manter uma alta eficiência na sua gestão.

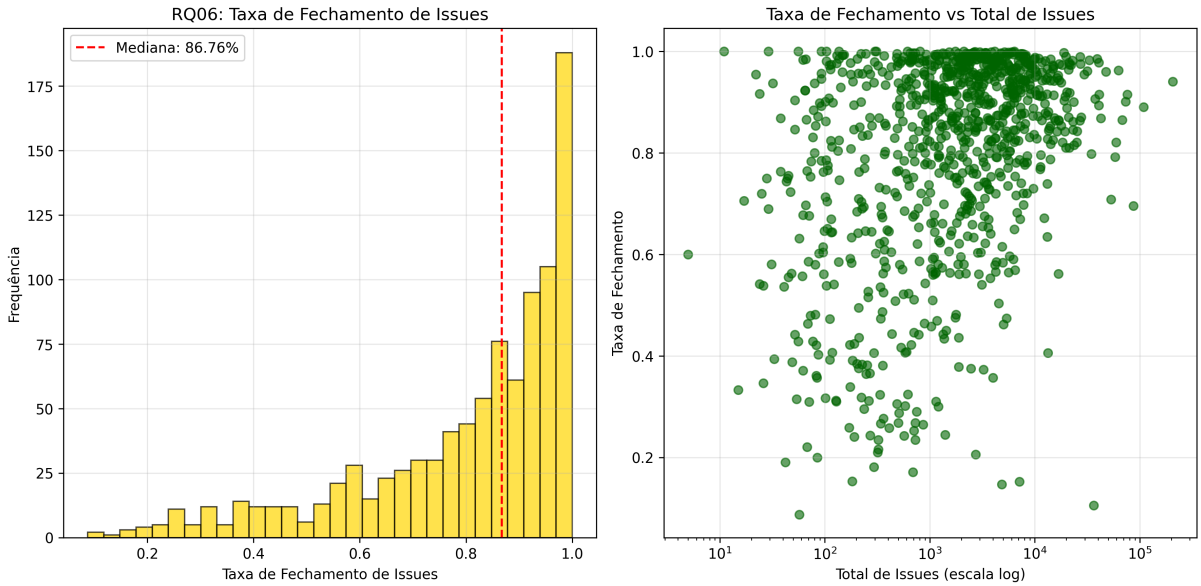


Figura 6: À esquerda, histograma da taxa de fechamento de issues, com mediana de 86,76% indicada. À direita, gráfico de dispersão comparando a taxa de fechamento com o total de issues.

7 Conclusão

Nesta seção, os resultados apresentados anteriormente são interpretados e contextualizados à luz das hipóteses iniciais. O objetivo é discutir se as expectativas sobre as características de repositórios populares foram confirmadas, refutadas ou se os dados revelaram um cenário mais complexo.

7.1 Hipótese 1: Repositórios Populares são Maduros

A primeira hipótese sugeria que a popularidade estaria associada a projetos mais antigos e estáveis. Os dados confirmam essa expectativa: a mediana de idade dos repositórios analisados foi de **8,4 anos**. Este valor indica que um projeto popular típico não é um fenômeno recente, mas sim um software que teve tempo para construir uma base de usuários, ganhar confiança da comunidade e atingir um estado de maturidade e estabilidade.

7.2 Hipótese 2: Alto Volume de Contribuição Externa

A expectativa era de que a alta visibilidade de projetos populares atrairia um grande volume de colaboração externa. Esta hipótese foi fortemente confirmada, com uma **mediana de 710 pull requests aceitas**. A distribuição dos dados, no entanto, revelou ser extremamente assimétrica, com projetos como o `first-contributions` atingindo dezenas de milhares de contribuições. Isso demonstra que, embora a contribuição externa

seja uma característica comum, o nível de engajamento em alguns repositórios atinge uma escala massiva, muito acima do típico.

7.3 Hipótese 3: Ciclos Regulares de Releases e Atualizações

Esta hipótese foi validada de forma mista e revelou nuances interessantes.

- **Atualizações:** A hipótese foi confirmada de maneira conclusiva. A análise mostrou que **100% dos repositórios foram atualizados no último mês**, com a mediana de tempo desde a última atualização sendo efetivamente zero dias. Isso indica que a manutenção constante é uma característica universal e indispensável para projetos populares.
- **Releases:** A hipótese foi confirmada parcialmente. Com uma **mediana de 36 releases**, fica claro que projetos populares publicam versões formais. Contudo, a análise de dispersão não mostrou uma correlação forte entre o número de releases e a popularidade (estrelas). Isso sugere que, embora a prática de versionamento seja importante, uma frequência de lançamento extremamente alta não é um pré-requisito para o sucesso.

7.4 Hipótese 4: Predomínio de Linguagens Consolidadas

Conforme esperado, a análise das linguagens primárias confirmou que os projetos mais populares são predominantemente desenvolvidos em tecnologias consolidadas e de alta demanda no mercado. O ranking foi liderado por **Python, TypeScript e JavaScript**, refletindo as tendências atuais no desenvolvimento web, de sistemas e de ciência de dados.

7.5 Hipótese 5: Gestão Ativa da Comunidade

A hipótese de que projetos populares teriam uma boa gestão de issues foi fortemente corroborada. A **mediana da taxa de fechamento de issues foi de 86,76%**. Um valor tão alto demonstra que esses projetos possuem comunidades ou equipes de mantenedores ativas e responsivas, que efetivamente processam e resolvem os problemas reportados pelos usuários, um fator crucial para a sustentabilidade e a boa reputação de um projeto.

7.6 Análise de Correlação entre Métricas

Para aprofundar a compreensão das relações entre as variáveis estudadas, foi gerada uma matriz de correlação, apresentada na Figura 7. A análise da matriz revela diversos pontos de interesse:

- **Forte Correlação Positiva:** Observa-se uma forte correlação de **0.63** entre o número de forks e de estrelas, o que era esperado, já que ambas são métricas primárias de popularidade e engajamento. Uma correlação igualmente forte (0.63) foi encontrada entre o total de *issues* e o número de *pull requests* aceitas. Isso sugere que projetos com alta atividade de desenvolvimento (PRs) também possuem um ecossistema vibrante de discussão e reporte de problemas (issues).

- **Correlações Moderadas:** A atividade de desenvolvimento (*pull requests*) apresenta correlação moderada com o número de releases (0.3) e com as métricas de popularidade (0.28 com forks). Isso indica que, como esperado, mais desenvolvimento tende a levar a mais lançamentos e maior visibilidade.

- **Correlações Fracas ou Negativas:** É notável a fraca correlação entre a idade do projeto (`age_years`) e a maioria das outras métricas, como estrelas (0.065). Isso reforça a ideia de que, embora os projetos populares *sejam* maduros, a idade por si só não é um fator que garante o aumento contínuo da popularidade. Adicionalmente, o tempo desde a última atualização (`days_since_update`) possui uma correlação fraca e negativa com as métricas de atividade, como estrelas (-0.18). Isso é coerente: quanto mais ativo e popular um projeto, menor tende a ser o tempo desde sua última atualização.

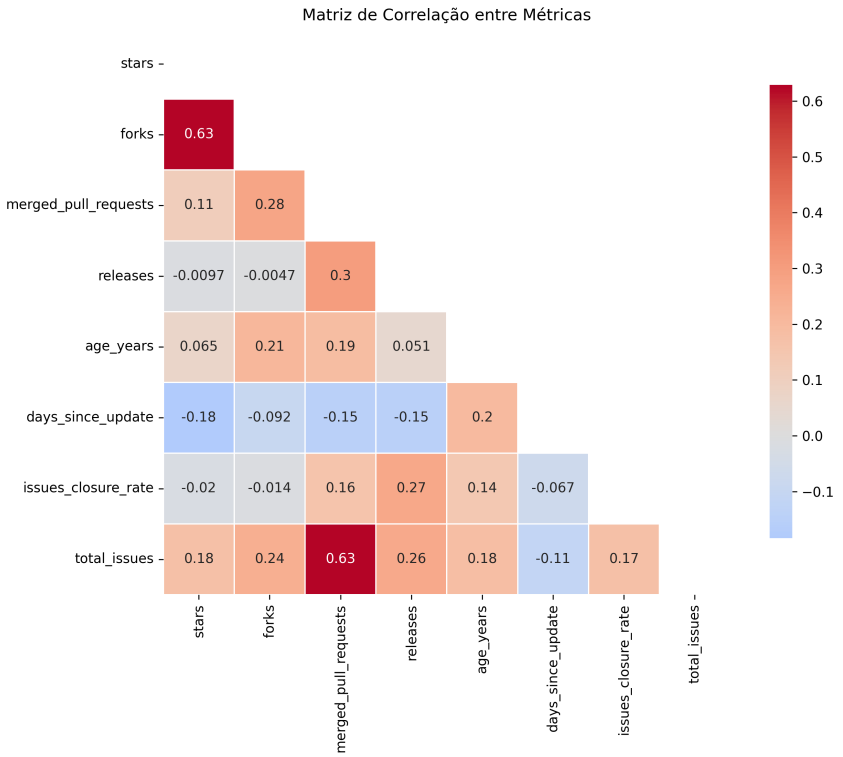


Figura 7: Matriz de correlação de Pearson entre as principais métricas quantitativas do estudo.

Referências

ANACONDA INC. **Python for Data Science: The Ultimate Guide**. [S.l.: s.n.], 2023. https://www.anaconda.com/learn/python-for-data-science.

FELIPE, K.; KÁSSIO, F. et al. A systematic process for Mining Software Repositories: Results from a systematic literature review. In: INFORMATION and Software Technology. [S.l.: s.n.], 2018.

GITHUB DOCS. **About the GitHub GraphQL API**. [S.l.: s.n.], 2024. https://docs.github.com/en/graphql.

_____. **Comparing GitHub's REST API and GraphQL API**. [S.l.: s.n.], 2024. https://docs.github.com/en/rest/overview/comparing-githubs-rest-api-and-graphql-api.

_____. **Migrating from REST to GraphQL**. [S.l.: s.n.], 2024. https://docs.github.com/en/graphql/guides/migrating-from-rest-to-graphql.

ISAIAH, Ayooluwa. **REST vs. GraphQL: A practical comparison**. [S.l.: s.n.], 2021. https://blog.logrocket.com/rest-vs-graphql-a-practical-comparison/.

MCKINNEY, Wes. **Python for Data Analysis**. 3rd. [S.l.]: O'Reilly Media, 2022.

MICROSOFT LEARN. **Explore and analyze data with Python**. [S.l.: s.n.], 2023. https://learn.microsoft.com/en-us/training/modules/explore-analyze-data-with-python/.

MONDELLI, A. et al. A framework for reproducibility using VMs and provenance capture. In: WORKSHOP on Reproducibility in Parallel Computing. [S.l.: s.n.], 2018.

MOZILLA SCIENCE GROUP. **Overcoming Challenges in Collaborative Projects using Version Control**. [S.l.: s.n.], 2017. https://mozillascience.github.io/collaborative-github-training/.