PoolinGH: Fast, Efficient, and Robust GitHub Repository Mining

Maxime André
maxime.andre@unamur.be
Namur Digital Institute
Namur, Belgium

Marco Raglianti
marco.raglianti@usi.ch
REVEAL @ Software Institute — USI
Lugano, Switzerland

Souhaila Serbout
souhaila.serbout@uzh.ch
University of Zurich
Zurich, Switzerland

Anthony Cleve
anthony.cleve@unamur.be
Namur Digital Institute
Namur, Belgium

Michele Lanza
michele.lanza@usi.ch
REVEAL @ Software Institute — USI
Lugano, Switzerland

Abstract

Researchers in Mining (open-source) Software Repositories (MSR) often create datasets that should survive the single paper and support long-term investigation of specific phenomena. Although popular, these studies recurrently deal with similar technical limitations. For instance, public collaborative development platforms, such as GitHub, impose hourly rate limits on their API requests. Furthermore, depending on network and API conditions, queries can fail and disrupt the process. These unexpected events can slow down or even invalidate the mining. Nevertheless, there are ways to minimize the undesirable effects in a reusable way while still complying with such limitations. However, best practices are often (re-)implemented on an *ad hoc* basis. Whatever works.

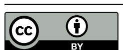
We propose PoolinGH, a lightweight, open-source, easy-to-use library, aimed at supporting researchers. It is designed to accelerate and ensure efficient and robust mining on the GitHub REST API while taking full advantage of its capabilities. PoolinGH enables automatic pooling of multiple access tokens and parallelizes queries. It optimizes queues and regulates network and API usage for respecting GitHub's limits and best practices. Error management and recovery or pruning in case of deadlocks are ensured. Search coverage maximization and progress monitoring are among the most useful features to avoid reinventing the wheel. We also provide solution templates that meet common needs for specific extensions of PoolinGH. A preliminary evaluation of these examples, involving tens of thousands of requests, demonstrates tangible gains.

CCS Concepts

• **Software and its engineering** → *Software configuration management and version control systems*; **Software libraries and repositories**; • **Information systems** → **Data mining**.

Keywords

mining software repositories, github rest api, library, poolingh



This work is licensed under a Creative Commons Attribution 4.0 International License.
MSR '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793321

ACM Reference Format:
Maxime André, Marco Raglianti, Souhaila Serbout, Anthony Cleve, and Michele Lanza. 2026. PoolinGH: Fast, Efficient, and Robust GitHub Repository Mining. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793302.3793321

1 Introduction

MSR, Mining (open-source) Software Repositories has become increasingly popular in the last decade, as demonstrated by numerous studies [5, 7, 29, 30] and tools [4]. MSR researchers query, *automatically* and *at scale*, public collaborative development platforms, such as *GitHub*, to create datasets that hopefully will make specific phenomena emerge from data, while facing several technical limitations [5, 7, 30]. For instance, GitHub imposes hourly rate limits (*i.e.,* quotas) on their *GitHub REST API* [14] to guarantee the quality of service and prevent potential abuse [5, 7]. Repository miners are encouraged to authenticate [15] to obtain 5,000 requests per hour instead of the default 60 [19] (previously 30 requests per minute [7]). Besides hourly rates, there is a maximum of 1,000 results per request [20], and a limit on query length (256 characters, a maximum of 5 *AND*, *OR*, or *NOT*) [18]. Platform-imposed limitations are complemented by less controllable ones. For example, depending on network and API load [5], some requests may unpredictably fail and need to be re-executed. Finally, there are requests that inevitably fail "by design" (*e.g.,* retrieving the list of *Linux kernel* contributors [24]).

Despite the many challenges to obtain quality data for MSR studies, there are clever ways to overcome them while complying with data providers' requirements [16]. A common practice is to "pool" access tokens among researchers and to parallelize the mining with rotating quotas. It is also possible to queue requests in a specific order to optimize uptime and minimize waiting. Unless complex computations and analyses are integrated in the early stages of the pipeline, knowing that tokens will inevitably be paused at some point, alternating working periods can accommodate short time scale limitations. In case of errors, best practices consist in either retrying or, when impossible or unsuccessful, pruning the wrong entries and storing the cause of error for rigorous *post hoc* quality assurance. Finally, oftentimes, the number of repository-specific queries can be reduced by optimizing the search space upfront, through pre-filtering and by identifying input ranges according to the expected results [7]. Regrettably, these practices are often adopted and (re-)implemented by researchers on an *ad hoc* basis [16] and to the minimum extent necessary due to time constraints.

2 PoolinGH

PoolinGH [1] is a lightweight, open-source, easy-to-use library designed to accelerate and ensure efficient and robust mining of GitHub through its APIs [14]. It enables automatic pooling of multiple access tokens, parallelizes queries, optimizes the search space and individual queues, monitors mining progress, and regulates network and API usage while respecting GitHub's terms of service. It also maximizes search coverage using a *binary-tree*-based range decomposition and performs error management and recovery, pruning results in case of deadlocks, without imposing any constraints on the type of query or the format of the results. The main contributions are the library with its optimization wrapper and the solution templates and examples illustrating specific implementations for common use cases. The library relies on three basic concepts:

- **Request**: GitHub REST API HTTP call. It specifies a URL, options, and a callback function to asynchronously process the results.
- **Client**: GitHub REST API client associated to one GitHub access token to perform *requests* to the API as an authenticated user.
- **Queue**: Process that sends *requests* to *clients*, dynamically parallelizes and distributes work, complies with rate limits (based on API response headers) with safety thresholds, and facilitates mining monitoring and error management.

2.1 Features

The PoolinGH package can be installed from npm [1] to then import the classes corresponding to the main concepts:

```
1 import {GitHubApiRequest,GitHubApiClient,GitHubApiQueue}
2 from 'poolingh';
```

Pooling: The user can create as many clients as there are tokens. The GitHub REST API Terms of Service prohibit sharing tokens solely to exceed API limitations [21]. We discourage creating fake accounts or collecting tokens abusively. Users remain solely responsible of potential misconduct. We encourage using legitimate and trustworthy users credentials in a secure way (*e.g.,* read-only token). The token pooling feature should only centralize a process that would otherwise need to be run separately on different researchers' devices, thus reducing decentralization friction and redundancy.

```
1 let client1 = new GitHubApiClient(YOUR_TOKEN_1);
2 let client2 = new GitHubApiClient(YOUR_TOKEN_2);
3 let client3 = new GitHubApiClient(YOUR_TOKEN_3);
4 // ...
```

Queuing: With the clients, the user can create a queue:

```
1 let queue = new GitHubApiQueue([client1, client2, ...]);
```

Querying: For each query, the user creates a request:

```
1 let request = new GitHubApiRequest(
2     'https://api.github.com/
3     search/repositories?q=stars:>=10000', // YOUR URL
4     {}, // YOUR OPTIONS
5     (result) => console.log(result) // YOUR CALLBACK
6 );
```

The callback function is customizable (*e.g.,* write to CSV, insert into a database, enqueue follow-up requests). The request can be appended at the front (unshift) or the end (push) of the queue:

The user can start and stop the queue at any time and requests are processed as long as the queue is running. When a request fails, it is sent back to the end of the queue waiting for a future retry.

Regulation: By default, all clients start together and work in parallel. Depending on the network and API load, distributing the work to minimize network spikes might be desirable, resulting in smoother requests traffic. The client's pause(resetAt) method pauses or delays clients (*e.g.,* to interleave requests on startup):

```
1 for (let i = 0; i < queue.getClients().length; i++) {
2     queue.getClients()[i].pause(Date.now() + 1000*60*i);}
```

Robustness: To address issues encountered when using the network or API, and to avoid being flagged as malicious by GitHub [16], each client is equipped with configurable safety settings. For example, each token has a safety margin for the number of requests remaining before switching to pause mode, and a safety margin for recovery time after a pause. Increasing or decreasing these settings can help align clients if PoolinGH is faster than the GitHub REST API at updating quotas. The following example increases the remaining requests threshold to 10 and the recovery buffer time to 5,000 ms (from the default 5 and 2,000, respectively).

```
1 let client1 = new GitHubApiClient(YOUR_TOKEN_1,10,5000);
```

Similar safety margins exist for queues. We distinguish limits between errors per request and per queue. The first limit aims to prevent wasting requests with repeated failures on the same query. Requests that reach the threshold are immediately pruned so that the process can continue safely. The second prevents massive error rates that could be flagged as malicious [16]. When the limit is reached, the process is immediately stopped. By default, the threshold per request is 5, and the threshold for the queue is equal to 1,000 times the request threshold. The queue can be initialized with different settings as follows:

```
1 let queue = new GitHubApiQueue([client1,...], 10, 20000);
```

Monitoring: PoolinGH integrates a logger. Each log line includes the timestamp and the ID of the token. It logs queue starts, stops, pauses, and resumptions, the current remaining requests count, and the performed queries. The logger also keeps track of 4xx and 5xx HTTP errors, such as attempting to exceed safety thresholds for failures, retries, and dropouts. It helps users conduct *post-hoc* investigations into unexpected events.

2.2 Templates

Besides the core implementation of PoolinGH, we provide[1] complementary example templates to illustrate the features in the most common cases. Among them, there are two templates of particular interest: Dynamic queue and search coverage maximization.

[1] In a separate repository at: https://github.com/PoolinGH/poolingh-examples

Dynamic queue: Some requests may need to be followed up by additional ones. For instance, listing repositories may be followed up by a request for each repository. In this case, the queue is dynamic, as it can receive new requests during execution as soon as each repository is retrieved from the initial request.
Search coverage optimization: To overcome the API restriction of 1,000 results per query [20], the search space is quickly pre-analyzed. Using a *binary-tree bisection* logic, a query evaluating a range exceeding 1,000 results is subdivided recursively until each child query represents contiguous ranges (successively further optimized by pruning empty ones), each within the limit. While the primary criteria of the request remain unchanged, a secondary "proxy" criterion is introduced to efficiently partition the space without altering the main criterion. This logic ensures a complete yet efficient coverage of all repositories of interest.

2.3 Implementation

The library is developed in Node.js. Figure 1 depicts the architecture, which reflects the three main concepts introduced earlier and the requests flows between the classes and external providers. Once the library installed, requests are created and appended to the queue. It distributes them to the available clients that each invokes the GitHub REST API. Once the response obtained, the callback is run.

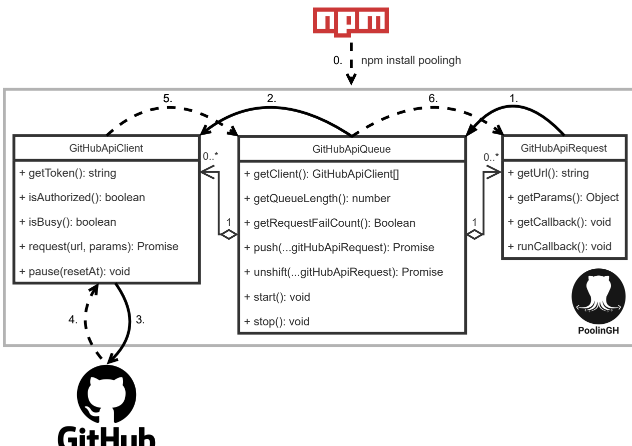


Figure 1: Architecture diagram

3 Evaluation

We evaluated PoolinGH through three illustrative experiments. In Figure 2 we show client usage, while in Figure 3 the number of requests over time. Each experiment targeted the same 47,1K repositories with at least 1,000 stars, as pre-filtered (in less than 2 minutes) through our *binary-tree bisection* approach, converted to 47,8K requests (1 per repository and about 700 for the range identification and repository listing) to the GitHub REST API, and then appended into our *dynamic queue*.

As illustrated in Figure 2, the first experiment ❶ was performed with a single client, alternating sequentially between periods of execution and pause. This experiment lasted approximately 9 hours, with frequent long pauses to wait for hourly limits to reset.

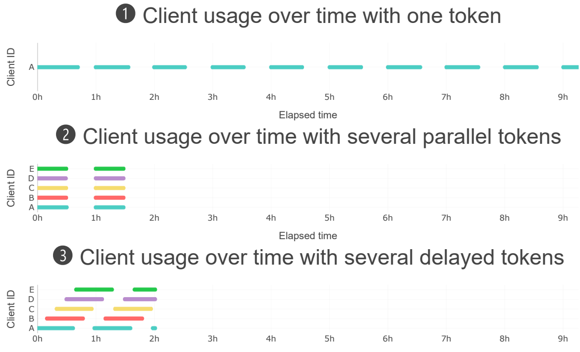


Figure 2: Client usage over time.

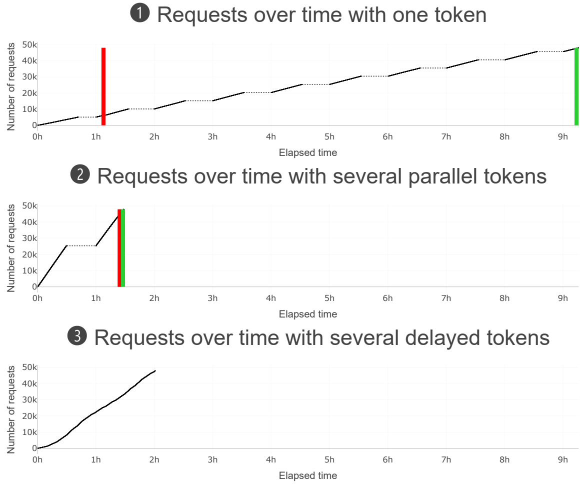


Figure 3: Requests over time.

The second experiment ❷ leveraged five clients in parallel. The work was distributed equally while respecting rate limits.

Figure 3 highlights the reduction of the total duration to about 1.5 hours, including a single 30 minutes pause, for all clients at the same time (flat line before the end of the first hour). The parallel execution resulted in a 6 times faster mining.

The third experiment ❸ used again five clients, but delayed them ten minutes apart in order to smooth network traffic. As shown in Figure 2, the experiment lasted about 30 minutes longer than the second one. The less steep slope drawn in Figure 3 indicates lower network congestion. As expected, no pause period was experienced.

During ❶ and ❷, we can also see how PoolinGH is robust with respect to failures. A 502 (Bad Gateway) and a connection error (ECONNRESET) happened in response to a request (red bar), successfully retried (green bar) before the end of the mining. PoolinGH's error handling and dynamic queue ensured a final error rate of 0%. Interestingly, when congestion was reduced by the delayed client starts, no such error was detected.

4 Related Work

Tools supporting the mining in MSR are not new. In 2011, just three years after the launch of GitHub, when the platform had around 2 million repositories [38], *GH Archive* [26], now discontinued, provided public, hourly archiving of GitHub events. In the same vein, Gousios and Spinellis [25] introduced *GHTorrent* [13], another offline queryable mirror dataset of GitHub's event history, which is also no longer maintained. Dyer *et al.* [11] introduced *Boa*, a domain-specific language (DSL) and infrastructure designed to ease MSR on "ultra-large-scale" software repositories, particularly for hypothesis testing. In the realm of DSLs, GitHub launched in 2016 their GraphQL API [22, 35]. It facilitates the aggregation and formatting of results. However, it also comes with result truncations and a specific rate limit calculation, called "points", in parallel to the REST API's limits [23]. GraphQL has been complementary to the REST API, which remains supported and popular [17]. Leveraging graph-based data representation, Wagstrom *et al.* [36] provided *GitMiner* [37], a tool that produces a dataset as a queryable graph representing relationships between repositories. Focusing on the stages after mining, Sokol *et al.* [33] presented *MetricMiner* [12], a tool to assist researchers in analyzing mined data (*e.g.,* metric calculation and statistical inference). With implementations in modern programming languages, Spadini *et al.* introduced *RepoDriller* [2] (in Java) and *PyDriller* [34] (in Python), two frameworks that simplify mining Git repositories, especially targeting commits and developers' data. More recently, Dabic *et al.* [7] opted for a different approach with *SEART GHS* [6], a representative, curated, and queryable online large sample dataset of GitHub repositories of interest for MSR studies. To reduce the need for language-specific tools, Dalla Palma *et al.* [9] presented *RepoMiner* [8], a language-agnostic framework to help researchers automate dataset creation for defect prediction studies. Heseding *et al.* [28] proposed *pyrepositoryminer* [27], a Python command-line tool implementing optimization strategies on Infrastructure-as-Code platforms (e.g., Ansible [10], TOSCA [3, 31]) to accelerate MSR tasks. Similarly, Nguyen *et al.* [32] proposed *PANDORA*, a tool to automatically and continuously mine data from multiple repositories across different platforms. Finally, there are several general-purpose web scraping tools, such as Scrapy [39], but they are less relevant and reliable, due to the limited number of results returned by GitHub on its web interface to prevent scraping.

5 Discussion

PoolinGH is not intended to replace previously mentioned solutions. Most are designed differently (*e.g.,* sampling, DSLs, offline dumps, GitHub model) making our approach complementary, by standing out in the following ways.

Lightweight. PoolinGH aims to minimize operational overhead compared to hosted services, (offline) dataset mirrors, and complex frameworks. Unlike mirror-based solutions, demanding substantial storage and infrastructure, our library does not store the complete repository and has a small runtime and storage footprint.

Robust. The strength of PoolinGH lies in its robustness, which is configurable from end to end. It ensures careful error recovery to find a compromise between the final error rate and time spent in completing the mining. Failed queries are handled automatically. Pruning thresholds help to manage transient and repeating failures. Used as safety measure to prevent wasting queries, they are still counted towards rate limits. In addition, PoolinGH offers monitoring through detailed logs to understand problematic queries and iterate during exploration.

Unrestricted. PoolinGH acts more like an API manipulation layer than an intermediary. It maintains a bijective compliant contract exposing the entire GitHub REST API, without introducing an alternative data model that would limit request types and impose specific result formats.

Up-to-date. Unlike dumping and archive-based solutions, our library always targets the most recent data available on GitHub.

Lawful. Our implementation complies with GitHub policies on API overloading, rate limits, and error handling. Nevertheless, users remain solely responsible for using and sharing legitimate and secure (*e.g.,* read-only) credentials in an ethical manner.

Practical. PoolinGH offers practical features to cleverly cope with GitHub REST API restrictions while favoring best practices. It allows automatic personal token pooling to parallelize queries with continuous monitoring of the individual limit rates. As encouraged by GitHub [16], which does not provide an implementation, PoolinGH offers an optimized queuing system, supporting both FIFO (*first in, first out*) and LIFO (*last in, first out*) strategies. Our solution provides additional practical features, such as the *binary-tree bisection* and the *dynamic queue*. The former contributes to efficient selection of the search space, while the latter enables decomposing complex requests (with multiple logical operators) into simpler linked sub-requests, to deal with query length limitations. For parallelism, PoolinGH offers a user-controlled customization: from fully parallel pooling to sequential distribution, including staggered parallelism to smooth network congestion and API load. Our tool comes with various templates solutions.

Relevant. Most of the similar approaches are developed for Java [2] and Python [8, 27] environments. Prior to our library, the JavaScript ecosystem lacked a comparable solution. PoolinGH, available through `npm`, is relevant as it fills this gap.

6 Conclusion

We presented PoolinGH, a lightweight open-source library designed to accelerate and ensure efficient mining of the *GitHub REST API* while dealing with the limitations faced by researchers conducting MSR studies. With automatic pooling of multiple tokens, parallel queries, optimized queues, our tool regulates network and API usage while respecting GitHub's rate limits and best practices. It performs error recovery and pruning of unfulfillable queries, maximizes efficient search coverage, and monitors the progress of the mining process. Our evaluation demonstrated that it can reduce mining duration linearly with the number of tokens (from 9 to 1.5 hours for 50k requests), being even more efficient for short minings. When network congestion is an issue, PoolinGH offers easy mechanisms to smooth network traffic with robust error recovery.

We plan to extend the library to support other code collaboration platforms beyond GitHub, enriching and refining the supported optimization strategies. Our tool extends the MSR researcher's toolbox with distinct advantages, fostering faster and more efficient dataset creation for exploration and experiments alike.

Open Source

To ensure access, verifiability, and replicability of our work, we make the following resources available as open source (MIT license):

Source Code	https://github.com/PoolinGH/poolingh
Archive	https://zenodo.org/records/17574294
NPM Package	https://www.npmjs.com/package/poolingh
Templates & Examples	https://github.com/PoolinGH/poolingh-examples

Acknowledgments

Research supported by the SofinaBoël Fund for Education and Talent, the Federation Wallonie-Bruxelles (ARC project RAINDROP), and by the Swiss National Science Foundation through the project "FORCE" (SNF project 232141).

References

[1] Maxime André, Souhaila Serbout, and Marco Raglianti. 2025. *PoolinGH.* doi:10.5281/zenodo.17574293
[2] Maurício Aniche et al. 2018. *RepoDriller.* https://github.com/mauricioaniche/repodriller/releases/tag/repodriller-2.0.1
[3] Matija Cankar, Anže Luzar, and Damian A. Tamburri. 2020. Auto-scaling Using TOSCA Infrastructure as Code. In *Proceedings of the 14th European Conference on Software Architecture (ECSA).* Springer, 260–268. doi:10.1007/978-3-030-59155-7_20
[4] K.K. Chaturvedi, V.B. Sing, and Prashast Singh. 2013. Tools in Mining Software Repositories. In *Proceedings of the 13th International Conference on Computational Science and Its Applications (ICCSA).* IEEE, 89–98. doi:10.1109/ICCSA.2013.22
[5] Valerio Cosentino, Javier Luis, and Jordi Cabot. 2016. Findings from GitHub: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR).* ACM, 137–141. doi:10.1145/2901739.2901776
[6] Ozren Dabic et al. 2024. *SEART GitHub Search (GHS).* https://github.com/seart-group/ghs/releases/tag/v1.17.1
[7] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *Proceedings of the 18th International Conference on Mining Software Repositories (MSR).* IEEE, 560–564. doi:10.1109/MSR52588.2021.00074
[8] Stefano Dalla Palma et al. 2023. *radon-repository-miner.* https://github.com/radon-h2020/radon-repository-miner/releases/tag/v1.0.4
[9] Stefano Dalla Palma, Dario Di Nucci, and Damian Tamburri. 2021. RepoMiner: a Language-agnostic Python Framework to Mine Software Repositories for Defect Prediction. arXiv:2111.11807. doi:10.48550/arXiv.2111.11807
[10] Michael DeHaan et al. 2025. Ansible. https://github.com/ansible/ansible
[11] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE).* IEEE, 422–431. doi:10.1109/ICSE.2013.6606588
[12] Fred Hutch Data Science Lab. 2025. *metricminer.* https://github.com/fhdsl/metricminer/tree/4ca1d7e6035ece5a1c3e92d644504939eb4541db
[13] ghtorrent. 2020. *The GHTorrent project.* https://github.com/ghtorrent/ghtorrent.org/tree/51890965af72da85bdd0954a50ef1a71603fb4e7
[14] GitHub. 2022. About the REST API. https://docs.github.com/en/rest/about-the-rest-api/about-the-rest-api?apiVersion=2022-11-28
[15] GitHub. 2022. Authenticating to the REST API. https://docs.github.com/en/rest/authentication/authenticating-to-the-rest-api?apiVersion=2022-11-28
[16] GitHub. 2022. Best practices for using the REST API. https://docs.github.com/en/rest/using-the-rest-api/best-practices-for-using-the-rest-api?apiVersion=2022-11-28
[17] GitHub. 2022. Comparing GitHub's REST API and GraphQL API. https://docs.github.com/en/rest/about-the-rest-api/comparing-githubs-rest-api-and-graphql-api?apiVersion=2022-11-28
[18] GitHub. 2022. Limitations on query length. https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28#limitations-on-query-length
[19] GitHub. 2022. Rate limits for the REST API. https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28
[20] GitHub. 2022. Using pagination in the REST API. https://docs.github.com/en/rest/using-the-rest-api/using-pagination-in-the-rest-api?apiVersion=2022-11-28
[21] GitHub. 2025. GitHub Terms of Service. https://docs.github.com/en/site-policy/github-terms/github-terms-of-service#h-api-terms
[22] GitHub. 2026. GitHub GraphQL API documentation. https://docs.github.com/en/graphql/
[23] GitHub. 2026. Rate limits and query limits for the GraphQL API. https://docs.github.com/en/graphql/overview/rate-limits-and-query-limits-for-the-graphql-api
[24] GitHub REST API. 2026. List of Linux kernel contributors. https://api.github.com/repos/torvalds/linux/contributors
[25] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: Github's data from a firehose. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR).* IEEE, 12–21. doi:10.1109/MSR.2012.6224294
[26] Ilya Grigorik. 2025. *GH Archive.* https://github.com/igrigorik/gharchive.org/tree/f1f4200e4a14da266081697adda7f1119cc54c03
[27] Fabian Heseding and Willy Scheibel. 2022. *fabianhe/pyrepositoryminer: 0.9.1.* doi:10.5281/zenodo.5918480
[28] Fabian Heseding, Willy Scheibel, and Jürgen Döllner. 2022. Tooling for time- and space-efficient git repository mining. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR).* ACM, 413–417. doi:10.1145/3524842.3528503
[29] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the Working Conference on Mining Software Repositories (MSR).* ACM, 92–101. doi:10.1145/2597073.2597074
[30] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071. doi:10.1007/s10664-015-9393-5
[31] Anže Luzar, Sašo Stanovnik, and Matija Cankar. 2020. Examination and Comparison of TOSCA Orchestration Tools. In *Proceedings of the 14th European Conference on Software Architecture (ECSA).* Springer, 247–259. doi:10.1007/978-3-030-59155-7_19
[32] Hung Nguyen, Francesco Lomio, Fabiano Pecorelli, and Valentina Lenarduzzi. 2022. PANDORA: Continuous Mining Software Repository and Dataset Generation. In *Proceedings of the 29th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 263–267. doi:10.1109/SANER53432.2022.00041
[33] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. 2013. MetricMiner: Supporting researchers in mining software repositories. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 142–146. doi:10.1109/SCAM.2013.6648195
[34] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2025. *PyDriller.* https://github.com/ishepard/pydriller/releases/tag/2.9
[35] The GraphQL Foundation. 2026. GraphQL | A query language for your API. https://graphql.org/
[36] Patrick Wagstrom, Corey Jergensen, and Anita Sarma. 2013. A network of Rails a graph dataset of Ruby on Rails and associated projects. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR).* IEEE, 229–232. doi:10.1109/MSR.2013.6624033
[37] Patrick Wagstrom and Corey Jergenson. 2012. *GitMiner.* https://github.com/pridkett/gitminer/releases/tag/fse2012
[38] Wikipedia. 2025. GitHub. https://en.wikipedia.org/w/index.php?title=GitHub&oldid=1312962171
[39] Zyte. 2026. *scrapy.* https://github.com/scrapy/scrapy/tree/9bae1ee21f51813f0fcc5869284acc4fb22ff649