www.nature.com/scientificdata

scientific data

Check for updates

OPEN

DATA DESCRIPTOR

Evolving collaboration, dependencies, and use in the Rust Open Source Software ecosystem

William Schueller[1,7], Johannes Wachs[1,2,7], Vito D. P. Servedio[1], Stefan Thurner[1,3,4] ✉ & Vittorio Loreto[1,5,6]

Open Source Software (OSS) is widely spread in industry, research, and government. OSS represents an effective development model because it harnesses the decentralized efforts of many developers in a way that scales. As OSS developers work independently on interdependent modules, they create a larger cohesive whole in the form of an ecosystem, leaving traces of their contributions and collaborations. Data harvested from these traces enable the study of large-scale decentralized collaborative work. We present curated data on the activity of tens of thousands of developers in the Rust ecosystem and the evolving dependencies between their libraries. The data covers eight years of developer contributions to Rust libraries and can be used to reconstruct the ecosystem's development history, such as growing developer collaboration networks or dependency networks. These are complemented by data on downloads and popularity, tracking dynamics of use, visibility, and success over time. Altogether the data give a comprehensive view of several dimensions of the ecosystem.

Background & Summary

Open Source Software (OSS) has recently been described as the "infrastructure" of the digital society[1]. OSS is an excellent example of open collaboration among many individuals that has a significant impact on the economy[2–5]. Within specific OSS **ecosystems** - collections of software programs or libraries are in many cases, but not always delineated by the use of a particular programming language like Rust, Python, or PHP - developers contribute software that depends on software already in the ecosystem, often created by strangers. For instance, a library that generates data from probability distributions may use a random number generator from another library rather than writing a new one. The outsourcing of core functions leads to a *rich structure of technical dependencies*, often represented as a network[6]. These libraries are usually hosted on collaborative coding platforms like GitHub or GitLab.

The nature of OSS contributions is such that the *traces of activity of individuals* are observable, i.e., what they contributed to which libraries and when. The cumulative efforts of thousands of developers can reveal a great deal about the nature of collaborative projects and work[7]. Information on the *use*, *visibility*, and *popular success* of individual libraries can be tracked over time[8], along with the co-evolution of technical dependencies and social collaboration[9]. Such data can give insight into the dynamics of massive and decentralized collaborations[6] and how these digital ecosystems evolve.

Here, we present a comprehensive dataset on one such ecosystem built around the Rust programming language. Rust, a relatively young language, has recently seen a sharp increase in popularity. Besides its significant connections with Mozilla[10], it is, as of December 2021, the second approved language of the Linux kernel besides C. For several years now, it has been voted the "most loved" language in the Stack Overflow Developer Survey. We have collected and curated temporal data on the technical dependencies, developer contributions, and the use and success of individual libraries. Specifically, we can observe when a developer made an elemental contribution of code to a specific library, what other libraries that library depends on, and how widely used and popular the library is.

[1]Complexity Science Hub Vienna, A-1080, Vienna, Austria. [2]Vienna University of Economics and Business, A-1020, Vienna, Austria. [3]Medical University of Vienna, A-1090, Vienna, Austria. [4]Santa Fe Institute, Santa Fe, USA. [5]Sony Computer Science Laboratories, 75005, Paris, France. [6]Physics Department, Sapienza University of Rome, 00185, Rome, Italy. [7]These authors contributed equally: William Schueller, Johannes Wachs. ✉e-mail: stefan.thurner@meduniwien.ac.at

We record over five million distinct contributions of over 72 thousand developers, contributing to over 74 thousand libraries over eight years.

Our data processing pipeline, available as open-source software, combines data from Cargo (the Rust ecosystem library manager) and the code hosting platforms GitHub and GitLab. It considers and handles multiple issues common to the study of collaborative software development data[11]: contributor disambiguation[12,13], bot detection[14], and the identification of nested projects and merged work. The result is a database tracking the evolution of a large, interconnected software ecosystem at a fine scale.

In contrast to other data sources on collaborative software development, our dataset contains more accurate and complete data for the Rust software ecosystem. Focusing on Rust allows us to integrate developer contributions with data on software dependencies and usage. In this way, our data is richer and more focused than what can be found in more extensive databases such as GHTorrent[15], GHArchive, Software Heritage[16], or World of Code[17]. Moreover, as we highlight in the Technical Validation section, we achieve a broader coverage by focusing on the Rust ecosystem: 15% of the packages in our dataset are not in the GHTorrent database. Our dataset also requires significantly less storage space than the sources mentioned above and can be directly analysed by researchers with minimal computing infrastructure requirements. At the same time, Rust is a large ecosystem that has evolved in a decentralized manner with contributions from thousands of developers hosted on multiple platforms, differentiating it from data sourced from single projects like the Linux Kernel or Apache projects[18].

We plan to update the dataset annually provided that the primary upstream sources (crates.io, GitHub & GitLab) remain stable. Researchers can also use our pipeline to reconstruct the dataset, a process that requires some data storage space (around 300 GB, though this volume will likely increase over time as the dataset is updated) and several days to query the data sources and process the results. Our code can also be adapted to collect data from other ecosystems, such as the Julia programming language's ecosystem. However, we note that not all ecosystems offer the same scope of data as Rust. For example: Rust is relatively young compared to Python, Ruby, or Javascript - as a result a much larger share of the Rust ecosystem's history is accessible on GitHub. The Rust ecosystem is also relatively small: estimates of the NPM (Node) ecosystem's size suggest its metadata alone are greater than 100 GB, while the repos themselves would take up multiple terabytes as described here: https://socket.dev/blog/inside-node-modules.

We proceed as follows: first, we describe our data collection and wrangling process and the resulting database. We compare our data coverage against GHTorrent, a widely used database of OSS contributions, finding that our data is more complete. We then outline usage notes for researchers interested in topics such as online cooperation[7,19] and collaborative innovation[20], success[21], and supply chain networks[22–24] in software. Our data can easily be represented as, for example, dynamic networks of collaborating developers, time series of usage statistics, growing networks of interdependent libraries, or combinations thereof.

Methods

We describe the data sources, and how we combine and curate data from various sources to create a comprehensive overview of the Rust ecosystem. We provide a visual overview of the established data processing pipeline in Fig. 1.

Data sources and collection.

Cargo: Libraries and dependencies. Our first source of data is the Cargo package (which are called *crates* in the Rust community) registry. Registries, often called package managers, play an important role in nearly all OSS ecosystems. They allow users to download and update different libraries while resolving dependencies and managing conflicts. Other examples of registries around different programming languages include PyPI for Python, CRAN for R, Rubygems for Ruby, and NPM for Node. We use Cargo as a source of technical dependencies and downloads for Rust. These are available as part of a daily dump from crates.io, accessible via: https://crates.io/data-access.

The data can be directly imported in a PostgreSQL database, and contains package names and creation dates, their versions, a list of dependencies for each version with the semantic versioning (semver) syntax associated to them, and the daily downloads per version of each package. For a relevant discussion of the importance of semantic versioning in OSS ecosystems, see recent work by Decan and Mens[25]. Packages are also often associated to a repository URL and a documentation URL, but those are not always provided and depend on maintainer input.

Code repositories: github, gitlab and other git platforms. To understand who contributes to which library, we turn to the social coding platforms on which these packages are hosted. In the case of Rust, nearly all packages in Cargo are hosted on either GitHub or GitLab. Specifically, 74,829 packages had links to either platform. Of these links, 51,657 were unique and 46,895 of them could be cloned from either GitHub (44,893), GitLab (1,498) or other git platforms (504). The inclusion of data from GitLab represents an important extension over the most widely used databases in OSS research, GHTorrent and GH Archive, which only use data from GitHub. Both of these platforms use Git version control, making projects hosted on either alternative comparable.

The elemental code contributions to OSS projects using the Git version control system are called *commits* and are associated to email addresses belonging to the author and contributor (in practice these are usually the same). GitHub and GitLab both host Git projects (called repositories or *repos*, for short), which we downloaded and used to extract information about activity and collaboration. The mapping between repos and the libraries hosted on Cargo is not one-to-one and requires additional processing, described below. The Git version control history of a project allows us to examine in detail the contribution histories of all developers working on a project. Indeed, previous work has shown how this highly granular data can be exploited to study collaboration and interactions among developers[26,27]. To do so, we "clone" (download) each repo locally. We also make use of the GitHub and GitLab GraphQL APIs to disambiguate contributors.

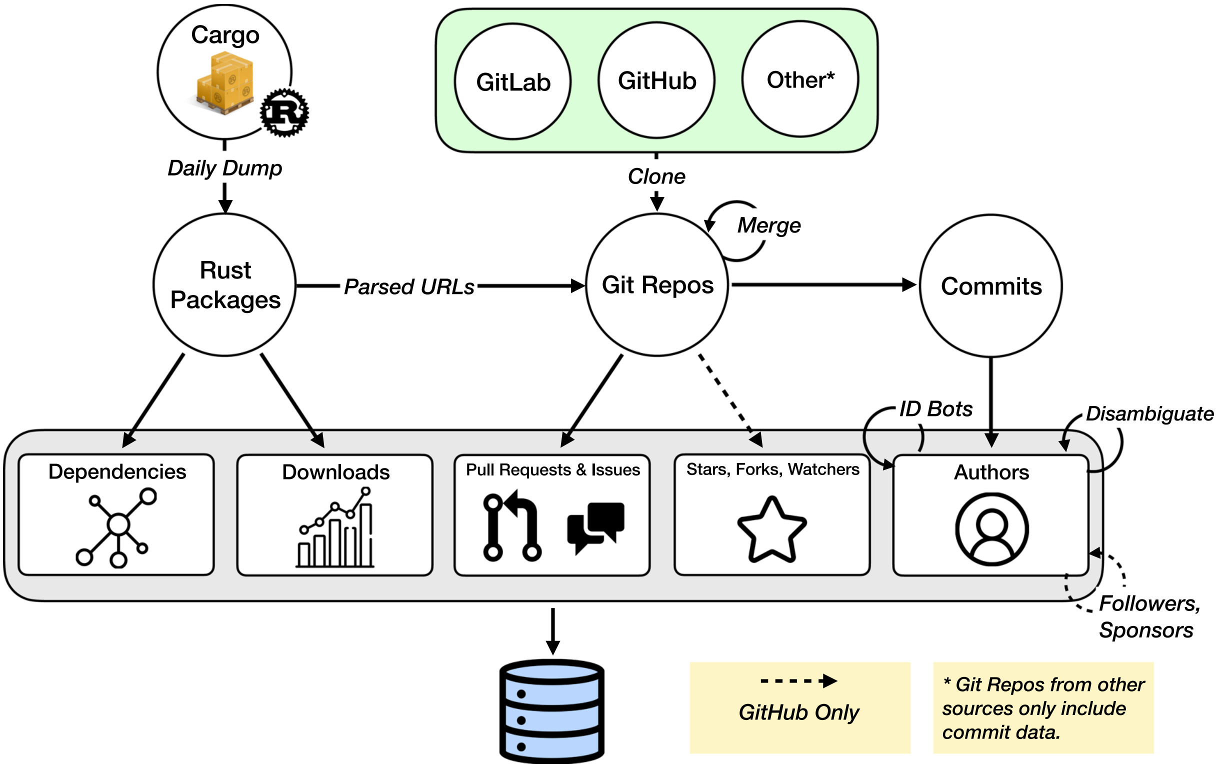

Fig. 1 Data processing pipeline. We collect data from Cargo, the package registry of the Rust programming language, and complement it with data from the code hosting platforms GitHub and GitLab. The processed result integrates information on package dependencies, use (downloads and stars), and authors.

Measuring use, visibility, and success: downloads and stars. We quantify two dimensions of the use of libraries: the number of times they are downloaded and the number of times they received positive social feedback (stars) on GitHub. The two metrics highlight different aspects of use. Downloads, sourced directly from Cargo, present a more technical measurement of use. GitHub's stars are more suggestive of visibility. For example, a highly technical library that provides background functions may be downloaded many times but have relatively few stars. The social aspect of GitHub and platforms is known to play an important role in collaborative software engineering[28,29]. GitHub stars and other forms of social feedback including followers and sponsors matter as information signals in the software development community for both libraries and contributors[30,31], for example on the labour market[32]. Of course, library owners can and do seek to raise GitHub stars for their libraries by promoting them[30]; stars are by no means an objective measure of library popularity. That said, we collected time-stamped GitHub stars and forks, a measure of code reuse, for repos directly from the GitHub GraphQL API. For GitHub users, we also collected the network of followers and sponsors each user has using the same API. GitHub implemented sponsorships for developers in 2019[33], enabling developers to crowdfund from users who appreciate their work directly on their GitHub pages.

Data processing.

Several additional curation steps need to be taken to insure the data collected is useful for the purposes of research into digital collaboration. We describe three specific steps here.

Repositories, packages, forks. One would easily assume individual packages of the Rust ecosystem tracked by Cargo correspond one-to-one with repositories on GitHub or GitLab, and this is generally the case. However, some repositories host several packages. For example modules or plugins that extend a core package are often hosted together in the same repo, but are distinguished in Cargo. This presents a data challenge: dependencies are recorded between packages, while contributions are recorded at the repo level.

For each package listed in the registry, one or several URLs are typically provided. They correspond to a link to the code and/or the documentation. Here, we take the URL corresponding to the code, and if empty, coalesce it with the documentation URL. We parse the individual URLs by recognizing the prefixes synonyms of *github.com* and *GitLab.com* and the pattern `github.com/<owner>/<name>` corresponding to a repository.

We also try to identify other git platforms by matching the pattern `<platform_root_url>/<owner>/<name>`, and using `git ls-remote` on a maximum of 5 repos per candidate to check if it is indeed a git platform.

Repositories are sometimes renamed, and both URLs can be present in distinct packages. The old URL typically redirects to the new one. To resolve this and be able to merge repositories under one entity, we use the

`<owner>/<name>` returned by the GraphQL API (of both GitLab and GitHub) when querying about the repository, for example when collecting information about forks.

After downloading the repositories (also called cloning), we analyse the commit data and retrieve commit hash, author and contributor emails and names (usually the same), and commit parents. We also compute the number of lines added and deleted for each commit. We analyse all available branches, and the unique hash ensures that we do not count commits twice per repository. However, commits can appear in several repositories, when one repository is a fork of another. We keep attribution of commits to each repository where they appear, but we also attribute each commit to a main repository, supposed to be its origin. For this, we retrieve the information about forks from the GitHub or GitLab GraphQL API, and take as origin the highest repository in the fork tree containing the commit. When this method is unsuccessful (e.g., undeclared forks, or forks between different Git platforms), we take the repository having the oldest package, using its creation date from the package manager.

Collaborative coding events. For repos hosted on GitHub and GitLab, we also collect data on collaboration events including issues and pull requests. Developers can open issues, highlighting bugs and problems with a code base. Maintainers can respond in comments and close issues, indicating whether they have been addressed. Pull requests are how non-core developers contribute code to a project - these can be commented upon and merged into a project, or rejected. These events, comments on them, their labels, status, and emoji reactions to them are all recorded in our data with time stamps. Similar data (comments and emoji reactions) for commits on GitHub are provided as well. Seeking to preserve developer privacy, we do not provide the texts associated to these events.

Dependencies. When analysing the dependencies between packages sourced from Cargo, and aggregating the network to dependencies between repositories, we noted the presence of cycles. In this context a cycle represents a pattern like: Package A depends on Package B, Package B depends on Package C, and Package C depends on Package A. Though there were only few of such examples, we decided to prune dependencies to remove such cycles for two reasons. The first is that they represent a logical inconsistency in what a dependency means. Second, without cycles, the resulting dependency network is a directed acyclic graph (DAG). DAGs are themselves interesting data structures appearing in a variety of data science contexts[34]. Given the small number of packages involved in cycles, we manually inspected them. Dependency cycles in package space correspond mainly to unnecessary dependency links or even fake packages for the sake of testing dependency declarations. Repository space cycles are more complex to prune. We adopted the heuristic to remove the dependencies (in both repo and package space) in cycles of length 2 by pruning the dependency of the oldest node to the newest (by creation date or earliest date of the repo's corresponding packages), and naturally removing dependency cycles of length 1. The remaining cycles were inspected manually, and the cycles were broken by removing the dependency links where it made more sense, in most cases from a repository having one of the highest download counts to one having one of the lowest. One remaining repository, although corresponding to numerous downloads, has been pruned from all dependencies to it because of the high number of cycles of the dependency network involving it. We included these pruned dependencies in the dataset for the sake of completeness, but we flagged them for easy removal, or for letting the possibility to investigate other link removal policies. We guarantee absence of cycles for the state of the dependency network at the end of the dataset (2022-09-07) and at the end of the preceding year (2021-12-31) in both spaces, but not at any arbitrary timestamp.

Merging of developer identities. There are many potential ways to disambiguate the identities of contributors[12,13], each presenting tradeoffs. In general, Git commits are signed by an email address, not a platform-specific username. Developers often commit code from different computers or environments with different email addresses in their configuration files, and this can result in a significant disambiguation problem. Rather than attempting to infer which email addresses potentially refer to the same person, we query the GitHub API for the GitHub account linked to each commit[35]. While this ignores the potential that developers use multiple accounts, we argue that it makes a larger amount of highly justifiable merges among commit author identities than an email address based approach. Email address-based author identity disambiguation would scale better in larger systems at the cost of accuracy.

For each email address, we carry out this process for the most recent commit registered for that email, and if this fails to return an account, we try again with a randomly sampled commit among all those corresponding to this email. After doing the same for GitLab, we also merge matching GitHub and GitLab logins (finding 87 such cases). An additional step is to parse the emails and spot those obviously belonging to GitHub, following the patterns `<login>@users.noreply.github.com or <randomint>+<login>@users.noreply.github.com`.

Bot accounts. Bots play an important role in modern software development[36,37], but need to be handled with care in any study of software systems, as they can make orders of magnitude more contributions than any human developer. Pooling them and their activity with that of human developers would skew any analysis of social cooperation and collaboration[11,14]. While bots have interesting effects on project evolution[36], we chose to detect and mark bots and more generally invalid accounts in our dataset with a view to excluding them as we are primarily interested in the patterns of contributions of developers. To identify bots, we used a two-step filtering process. First, we extracted all bots on a curated list used as ground truth for bot detection in the software engineering community[14]. We then filtered remaining GitHub accounts with the substrings "*[bot]", "*-bot", "*-bors", "bors-*", "dependabot-*" in their usernames, and finally inspecting manually each individual account with pattern "*bot*". After filtering out bot accounts labelled this way, a manual inspection of the 100 most active remaining accounts was conducted, as well as of all accounts containing the substring "bot". The manual inspection took the following steps: checking the GitHub webpage of a user for a clear name, looking at the description, looking at the commit/PR comments. The 100 most active accounts by number of commits across the full timespan of the dataset were considered, as well the 100 most active accounts in the year 2020.

For users that could not be associated to a GitHub account, their emails are filtered when the last part of their prefix (separated by., − or +) is equal to "bot", "ghbot", "bors", "travis" or "bot". A few remaining email strings without "@" were discarded, like "localhost", "N/A" or empty string. Manual inspection of the most active 100 emails without a GitHub account revealed a few more bots. The list of manually discarded bots is available in the file `botlist.csv`. Our dataset includes the bots among the users, flagged with a Boolean "is_bot" attribute to enable filtering.

Data Records

We host our data on Figshare[38] and the code used to collect and process the data from our sources on GitHub (https://github.com/wschuell/repo_datasets). Both platforms track the history of the data and code, allowing researchers to use any version they prefer as we continue to update and extend both. We share data in several formats, noting that in no format does the data exceed 6 GB when compressed.

Before we describe the data, we discuss data pseudonymisation. To preserve developer privacy, we provide data that is scrubbed of information that can directly be used to identify individuals. We do so in the following way: we discard name attributes and hash (via MD5 with a random salt) email addresses and GitHub/GitLab logins. Researchers interested in studying social or demographic characteristics of developers, such as gender[39,40], geography[41–43], or both[44], could adapt our approach to data collection and analyse these attributes. However, they should consider potential ethical issues that arise when associating such information to users[45].

In Table 1, we list the tables in the database along with a description of their content and purpose. For the sake of brevity, we refer the reader to the accompanying materials on Figshare[38] for a schema of the database and a description of the variables included.

Technical Validation

In this section, we report statistics on the completeness of our data. An advantage of defining the Rust ecosystem as all those packages hosted on Cargo, is that we can precisely measure how many packages we can successfully integrate into our database. In particular, we can report the share of packages that we can connect to repos on the social coding platforms GitHub and GitLab. As we will see below, we have a very high rate of linkage. Moreover, the packages that we could not integrate are typically those with very few downloads. This suggests that most projects on Cargo that do not appear on GitHub or GitLab are small personal projects or preliminary work.

Package coverage among repositories. Some Rust packages hosted on Cargo could not be linked to repos on GitHub or GitLab. They either are on a different platform, for example on Bitbucket, Google Cloud, Sourceforge, or on personal websites. Still others had a link to a GitHub or GitLab domain (i.e. a repo) but could not be cloned. This can happen if a link was incorrectly transcribed, if the repo was deleted, or if the package is listed only for name squatting or test purposes and does not correspond to any repository. Specifically, out of 91,437 packages, 74,829 were linked to a repository on GitHub, GitLab or another git platform and 68,239 of these were successfully cloned, although only 51,657 packages pointed to distinct URLs (and 46,895 were cloned). Hypothesizing that the most important packages are the most downloaded ones, we can see in Fig. 2 that our coverage increases among the most important packages, measured by use (downloads).

Across the cloned repos, we gathered 5,656,407 commits, the elemental units of contribution in the git version control system. Excluding 219 bots identified among the GitHub accounts, these contributions were made by 58,329 GitHub users and 450 GitLab users. The raw data contains 89,399 identifying email addresses, highlighting the significant amount of disambiguation of author identities our pipeline implements. 14,266 of them could not be associated to a GitHub or GitLab account.

Comparison with other data sources. We first compare our data with data collected in the GHTorrent project[15]. The GHTorrent project aims to collect all activity on GitHub for use in research. As we have already noted, most activity in the Rust ecosystem takes place on GitHub, with a small but significant share taking place on GitLab. Besides the inclusion of GitLab data, we observed that our data contains a significant amount of activity hosted on GitHub that is missing from the GHTorrent database (the SQL version), when comparing the data in our dataset tagged as happening before the last date of user creation in the GHTorrent database–May 31st 2019, just before midnight–and corresponding to the repositories that could be cloned. Specifically, we found only 14,563 unique users (vs. 21,989 GitHub users in our database), 13,623 unique repos (vs. 16,069 identified GitHub repos in our database), and 1,299,414 unique commits (vs. 1,942,997 GitHub hosted commits in our database before the last date of GHTorrent). The GHTorrent project uses the GitHub REST API and collects data from the public event timeline using user-donated API keys. Outages on either the GHTorrent side or on the GitHub REST API, or rate limited API keys may explain missing data. While GHTorrent remains an excellent source of dataset for all of GitHub, these comparisons suggests that for a focused look at a single ecosystem, a customised pipeline can significantly increase data coverage. More detailed statistics concerning the comparison can be found on Figshare[38] in the file ghtorrent_comparison.yml.

We also note that our data collection pipeline is not the only way to collect similar data on OSS ecosystems. For example, the GrimoireLab toolchain is a collection of tools to gather and analyze data on software[46]. These tools provide users with sophisticated analyses of the health and activity levels of particular projects, and groups of projects. In particular its Perceval data retrieval module can do many of the same things as our collection

Table Name	Description
commit_comment_reactions	Individual emoji reactions to commit comments
commit_comments	Comments to commits
commit_parents	Parenthood relationships between commits. Typically one per commit, can be 0 or more.
commit_repos	Repos to which commits belong. At least one, but can be several (e.g. with forks).
commits	Listing metadata about specific commits
filtered_deps_package	Packages wich are filtered when appearing as a dependency to avoid cycles
filtered_deps_packageedges	Dependency edges between packages filtered to avoid cycles
filtered_deps_repo	Repositories wich are filtered when appearing as a dependency to avoid cycles
filtered_deps_repoedges	Edges directly discarded in the dependency graph
followers	Listing followers of GitHub accounts
forks	Listing forks declared on GitHub
identities	Listing each individual identity of the developers (email, GitHub account, Gitlab account)
identity_types	Listing of identitiy types (email, GitHub account, Gitlab account)
issue_comment_reactions	Individual emoji reactions to issue comments
issue_comments	Comments to issues
issue_labels	Individual labels of each issue
issue_reactions	Individual emoji reactions to each issue
issues	Listing of issues per repository
merged_identities	Keeping track of identities having been merged, for information purposes
merged_repositories	Repositories having been merged (after identifying renaming or typo in URL)
org_memberships	Membership of organization declared on GitHub for GitHub users.
organizations	Organizations or work groups as declared on GitHub
package_dependencies	Listing package dependencies (version to package with semver)
package_version_downloads	Listing daily downloads of package versions
package_versions	Listing versions of packages
packages	Listing packages
pullrequest_comment_reactions	Individual emoji reactions to each pull request
pullrequest_comments	Comments to each pull request
pullrequest_labels	Individual labels of each pull request
pullrequest_reactions	Individual emoji reactions to each pull request
pullrequests	Listing of pull requests per repository
repo_languages	Listing language composition of repositories (GitHub)
repositories	Listing repositories
sources	Listing data sources
sponsors_user	Listing sponsorships of developers
stars	Listing starring events of repositories
urls	Listing retrieved URLs, and their parsed equivalent
user_languages	Language composition of contributions of GitHub users (year before collection).
users	Listing users (who can have several identities: email, GitHub, Gitlab)
watchers	Listing watchers (developers being notified of changes) of each repository

Table 1. Table of tables in the database. A full database schema is available at the Figshare repository and at https://github.com/wschuell/repo_datasets.

pipeline. Our tool focuses rather on one thing: collecting a nearly complete dataset on the Rust ecosystem quickly (i.e. maximizing GraphQL API calls vs. REST API calls), with minimal requirements for users seeking to replicate or refresh the dataset. In this sense the two can be considered as potential complements, rather than substitutes: an analyst studying specific libraries in the Rust ecosystem can easily use the GrimoireLabs to obtain metrics on those libraries and additional data about them, for example from Jira or Twitter.

Usage Notes

To demonstrate how to read in and analyse the database, we provide short Jupyter notebooks that extract data and carry out elementary data manipulations, with the data aggregated at the monthly level. These notebooks are included with the other software in our materials. In one, we create the dependency network of the Rust ecosystem at different times and measure its growth. This data can be used to study ecosystem health, as errors and issues are known to spread through these networks[6]. In another, we plot the time series of downloads and new pull requests to specific repositories over time. Such time series can be used to study the dynamics of collaboration, use, and success at a fine-grained level[8]. In a third, we show how to load in the data of developers and packages as a rectangular matrix, which can be analysed as a bipartite network or, after a projection, as a developer-developer collaboration network. The bipartite network could be used to study the overall complexity

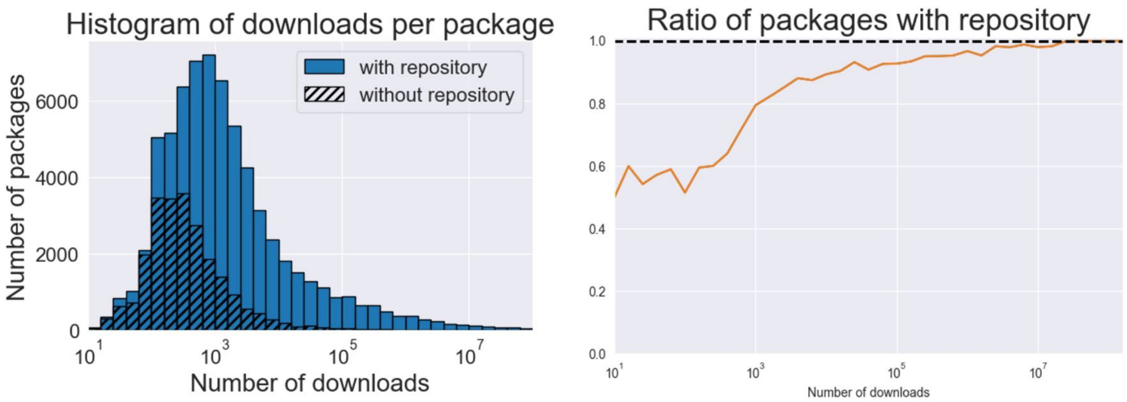

Fig. 2 Data coverage: we check the number of Rust packages for which we could identify and download a corresponding Git repo (from GitHub or GitLab) in terms of their use, measured in downloads. We could link a large majority of packages to repos, and have a significantly higher success rate if we consider packages that have been downloaded more often.

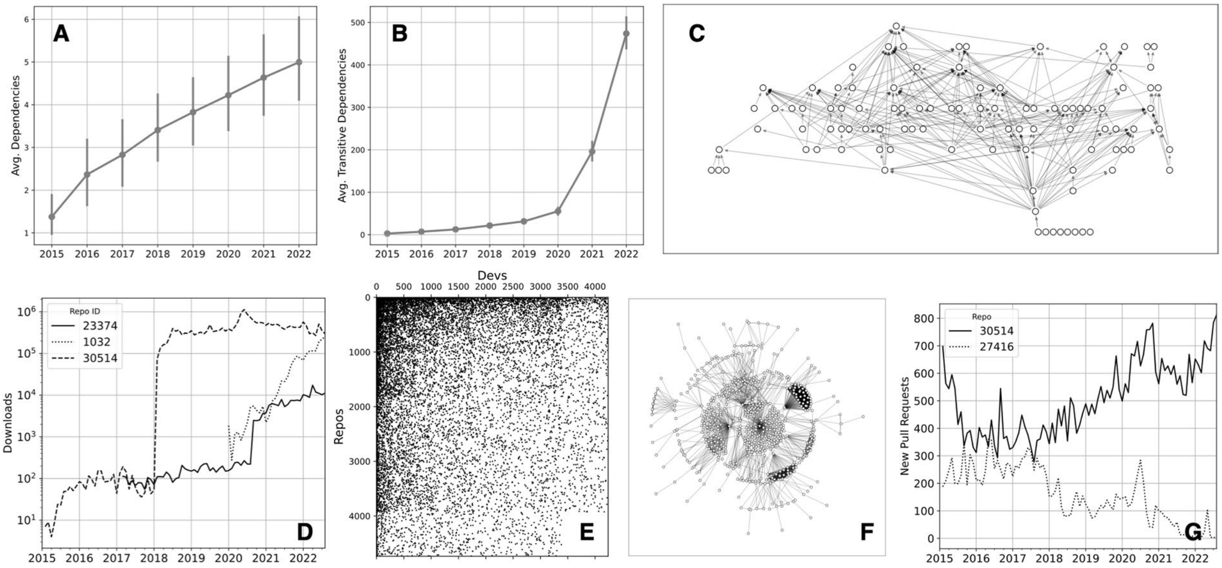

Fig. 3 Illustrative plots from demonstration notebooks indicating potential data-processing workflows. (**A**) Average number of dependencies per package at the beginning of each year. (**B**) Evolution of the number of transitive dependencies per package. (**C**) The dependency network of the 100 most downloaded Rust packages in September 2022. (**D**) Time series of monthly downloads of three successful Rust packages. (**E**) Bipartite adjacency matrix of users/developers and the repos they work on in the year 2021, lightly filtered. (**F**) Developer-developer collaboration network in 2021, filtered for developers collaborating on at least three repos. (**G**) Monthly time series of new pull requests to two initially successful Rust libraries.

of the ecosystem[47,48], while the developer network reveals patterns of collaborations between projects[49]. In Fig. 3 we present several illustrative examples of descriptive analyses resulting from the demonstration notebooks.

Our dataset can also be used as an input to study current themes of the empirical software engineering community. For example, our data can extend in time the work of Decan *et al.*[6] on transitive dependencies and library centrality in Rust. These measures themselves provide interesting ways to quantify success and importance of libraries.

Another interesting area of research that our data might be applied to is the concept of code smells, in particular community smells[50]. Code smells are problematic kinds of patterns in software, while community smells refer to such sub-optimal patterns on the social and collaborative levels of collective software development. For example, Tamburri *et al.*[50] describe several community smells such as the "lone wolf" - when a single developer acts in a unilateral and inconsiderate way - or the "organizational silo" - when developer teams working on different parts of a codebase only communicate through one or two team members. These smells are quantified in part by considering the collaboration and communication networks of developers. Our dataset can be used to calculate many of these concepts including collaboration network measures and socio-technical congruence[9] at the ecosystem level by considering which developers contribute to which libraries and their interactions in dealing with issues and pull requests. Social outcomes like turnover, which are used to test and validate measures of community smells, can also easily be measured.

A third line of software engineering research which our data can complement is the question of use and success of software. As mentioned earlier, GitHub stars are imperfect indicators of successful or high-quality software[30]. Recent work has sought to refine measure of software success by studying why users adopt specific software[51]. In other words: what factors and metrics do users take into account when picking an OSS solution for a problem? Beyond the metrics reported in our data such as stars, downloads, and watchers, many others mentioned in this literature can be calculated. These include metrics of community support and adoption (number of contributors, issue and pull request response times) and maturity (releases, number of forks, age), and to some extent quality (code size and rate of issue resolution).

More generally, that is beyond the broad research field of empirical software engineering, our dataset can be used to explore the interactions between social collaboration, technical dependencies, and the visibility and usage of components of a large software system. The dynamic interactions between these layers of the data offer significant potential for research relating to massive decentralized cooperation (similar to Wikipedia[52,53]), the dynamics of teams and their success in digital communities[21], and the evolution of software systems[54].

Code availability

Code to recreate the database is included in our Figshare upload[38] and can also be found in a dedicated repository https://github.com/wschuell/repo_datasets. The software is written in the Python programming language. The database can be created as either PostgreSQL or SQLite database. Version requirements are recorded in the project's Readme file.

Received: 12 May 2022; Accepted: 1 November 2022;
Published online: 16 November 2022

References

1. Eghbal, N. *Working in public: The making and maintenance of Open Source Software* (Stripe Press, 2020).
2. Lerner, J. & Tirole, J. Some simple economics of open source. *The Journal of Industrial Economics* **50**, 197–234 (2002).
3. Greenstein, S. & Nagle, F. Digital dark matter and the economic contribution of Apache. *Research Policy* **43**, 623–631 (2014).
4. Nagle, F. Learning by contributing: Gaining competitive advantage through contribution to crowdsourced public goods. *Organization Science* **29**, 569–587 (2018).
5. Nagle, F. Open Source Software and firm productivity. *Management Science* **65**, 1191–1215 (2019).
6. Decan, A., Mens, T. & Grosjean, P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* **24**, 381–416 (2019).
7. Zöller, N., Morgan, J. H. & Schröder, T. A topology of groups: What github can tell us about online collaboration. *Technological Forecasting and Social Change* **161**, 120291 (2020).
8. Sinatra, R., Wang, D., Deville, P., Song, C. & Barabási, A.-L. Quantifying the evolution of individual scientific impact. *Science* **354**, aaf5239 (2016).
9. Cataldo, M., Herbsleb, J. D. & Carley, K. M. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical Software Engineering and Measurement (ESEM)*, 2–11 (2008).
10. Jung, R., Jourdan, J.-H., Krebbers, R. & Dreyer, D. Safe systems programming in rust. *Communications of the ACM* **64**, 144–152 (2021).
11. Kalliamvakou, E. *et al*. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* **21**, 2035–2071 (2016).
12. Fry, T., Dey, T., Karnauch, A. & Mockus, A. A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits. In *Proceedings of the 17th international conference on mining software repositories*, 518–522 (2020).
13. Gote, C. & Zingg, C. gambit–An Open Source Name Disambiguation Tool for Version Control Systems. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 80–84 (IEEE, 2021).
14. Golzadeh, M., Decan, A., Legay, D. & Mens, T. A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments. *Journal of Systems and Software* **175**, 110911 (2021).
15. Gousios, G. & Spinellis, D. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 12–21 (IEEE, 2012).
16. Pietri, A., Spinellis, D. & Zacchiroli, S. The software heritage graph dataset: public software development under one roof. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 138–142 (IEEE, 2019).
17. Ma, Y., Bogart, C., Amreen, S., Zaretzki, R. & Mockus, A. World of Code: an infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 143–154 (IEEE, 2019).
18. Roberts, J. A., Hann, I.-H. & Slaughter, S. A. Understanding the motivations, participation, and performance of Open Source Software developers: A longitudinal study of the Apache projects. *Management science* **52**, 984–999 (2006).
19. Szell, M. & Thurner, S. Measuring social dynamics in a massive multiplayer online game. *Social Networks* **32**, 313–329 (2010).
20. Monechi, B., Pullano, G. & Loreto, V. Efficient team structures in an open-ended cooperative creativity experiment. *Proceedings of the National Academy of Sciences* **116** (2019).
21. Klug, M. & Bagrow, J. P. Understanding the group dynamics and success of teams. *Royal Society Open Science* **3**, 160007 (2016).
22. Ma, Y. Constructing supply chains in Open Source Software. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 458–459 (IEEE, 2018).
23. Zimmermann, M., Staicu, C.-A., Tenny, C. & Pradel, M. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, 995–1010 (2019).
24. Ohm, M., Plate, H., Sykosch, A. & Meier, M. Backstabber's knife collection: A review of Open Source Software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 23–43 (Springer, 2020).
25. Decan, A. & Mens, T. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* **47**, 1226–1240 (2019).
26. Scholtes, I., Mavrodiev, P. & Schweitzer, F. From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects. *Empirical Software Engineering* **21**, 642–683 (2016).
27. Gote, C., Scholtes, I. & Schweitzer, F. git2net-mining time-stamped co-editing networks from large git repositories. In *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 433–444 (IEEE, 2019).

28. Dabbish, L., Stuart, C., Tsay, J. & Herbsleb, J. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, 1277–1286 (2012).
29. Marlow, J., Dabbish, L. & Herbsleb, J. Impression formation in online peer production: activity traces and personal profiles in GitHub. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, 117–128 (2013).
30. Borges, H. & Valente, M. T. What's in a GitHub star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software* **146**, 112–129 (2018).
31. Moldon, L., Strohmaier, M. & Wachs, J. How gamification affects software developers: Cautionary evidence from a natural experiment on GitHub. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 549–561 (IEEE, 2021).
32. Papoutsoglou, M., Ampatzoglou, A., Mittas, N. & Angelis, L. Extracting knowledge from on-line sources for software engineering labor market: A mapping study. *IEEE Access* **7**, 157595–157613 (2019).
33. Shimada, N., Xiao, T., Hata, H., Treude, C. & Matsumoto, K. HGitHub Sponsors: Exploring a New Way to Contribute to Open Source. In *IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (IEEE, 2022).
34. Corominas-Murtra, B., Goñi, J., Solé, R. V. & Rodrguez-Caso, C. On the origins of hierarchy in complex networks. *Proceedings of the National Academy of Sciences* **110**, 13316–13321 (2013).
35. Montandon, J. E., Silva, L. L. & Valente, M. T. Identifying experts in software libraries and frameworks among GitHub users. In *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 276–287 (IEEE, 2019).
36. Wessel, M. *et al*. The power of bots: Characterizing and understanding bots in OSS projects. *Proceedings of the ACM on Human-Computer Interaction* **2**, 1–19 (2018).
37. Wessel, M. et al. Bots for pull requests: The good, the bad, and the promising. In *44th ACM/IEEE International Conference on Software Engineering (ICSE)*, vol. 26, 16 (ACM/IEEE, 2022).
38. Schueller, W., Wachs, J., Servedio, V. D., Thurner, S. & Loreto, V. Replication Data for Evolving collaboration, dependencies, and use in the Rust Open Source Software ecosystem, *figshare*, https://doi.org/10.6084/m9.figshare.c.5983534.v1 (2022).
39. Vasilescu, B. et al. Gender and tenure diversity in github teams. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, 3789–3798 (2015).
40. Rossi, D. & Zacchiroli, S. Worldwide gender differences in public code contributions: and how they have been affected by the covid-19 pandemic. In *IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022).
41. Rastogi, A., Nagappan, N., Gousios, G. & van der Hoek, A. Relationship between geographical location and evaluation of developer contributions in GitHub. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–8 (2018).
42. Braesemann, F., Stoehr, N. & Graham, M. Global networks in collaborative programming. *Regional Studies, Regional Science* **6**, 371–373 (2019).
43. Wachs, J., Nitecki, M., Schueller, W. & Polleres, A. The Geography of Open Source Software: Evidence from GitHub. *Technological Forecasting and Social Change* **176**, 121478 (2022).
44. Prana, G. A. A. et al. Including everyone, everywhere: Understanding opportunities and challenges of geographic gender-inclusion in OSS. *IEEE Transactions on Software Engineering* (2021).
45. Gousios, G. & Spinellis, D. Mining software engineering data from GitHub. In *IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 501–502 (IEEE, 2017).
46. Gonzalez-Barahona, J. M., Izquierdo-Cortázar, D. & Robles, G. Software development metrics with a purpose. *Computer* **55**, 66–73 (2022).
47. Hidalgo, C. A. & Hausmann, R. The building blocks of economic complexity. *Proceedings of the national academy of sciences* **106**, 10570–10575 (2009).
48. Servedio, V. D. P., Buttà, P., Mazzilli, D., Tacchella, A. & Pietronero, L. A new and stable estimation method of country economic fitness and product complexity. *Entropy* **20**, 783 (2018).
49. Singh, P. V. The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **20**, 1–27 (2010).
50. Tamburri, D. A., Palomba, F. & Kazman, R. Exploring community smells in open-source: An automated approach. *IEEE Transactions on software Engineering* **47**, 630–652 (2019).
51. Li, X., Moreschini, S., Zhang, Z. & Taibi, D. Exploring factors and metrics to select Open Source Software components for integration: An empirical study. *Journal of Systems and Software* **188**, 111255 (2022).
52. Brandes, U., Kenis, P., Lerner, J. & Van Raaij, D. Network analysis of collaboration structure in wikipedia. In *Proceedings of the 18th international conference on World wide web*, 731–740 (2009).
53. Mestyán, M., Yasseri, T. & Kertész, J. Early prediction of movie box office success based on wikipedia activity big data. *PloS one* **8**, e71226 (2013).
54. Solé, R. & Valverde, S. Evolving complexity: how tinkering shapes cells, software and ecological networks. *Philosophical Transactions of the Royal Society B* **375**, 20190325 (2020).

Acknowledgements

The extraction and construction of the database has been partially funded by Sony Computer Science Laboratories Paris under a dedicated contract with the Complexity Science Hub Vienna. This work has also been partially supported by the Austrian Research Promotion Agency FFG project # 882184. The authors thank Amélie Desvars-Larrive for advice regarding the manuscript.

Author contributions

W.S., J.W., V.D.P.S., V.L., S.T. coordinated the production of the dataset. W.S. designed the database schema, created the tables, and carried out the dataset validation checks. J.W. implemented the exploratory analyses and demonstrations. V.L. and S.T. supervised and mentored the team. All authors contributed to writing the data descriptor.

Competing interests

The authors declare no competing interests.

Additional information

Correspondence and requests for materials should be addressed to S.T.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.