Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 207 (2022) 1261–1270

Procedia Computer Science

www.elsevier.com/locate/procedia

26th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2022)

Content queries and in-depth analysis on version-controlled software

Jannek Squar[a,*], Niclas Schroeter[a], Anna Fuchs[a], Michael Kuhn[b], Thomas Ludwig[c]

[a] *Universität Hamburg, Bundesstraße 45a, Hamburg 20146, Germany*
[b] *Otto von Guericke University Magdeburg, Universitätsplatz 2, Magdeburg 39106, Germany*
[c] *Deutsches Klimarechenzentrum GmbH, Bundesstraße 45a, Hamburg 20146, Germany*

Abstract

Writing scientific code usually implies the need to coordinate and conflate the contributions of several scientific programmers. Using Git hosting services eases this process, because the hosting services offer many features, which assist in collaborated work on code. The well-established hosting service *GitHub* has seen continuous growth in terms of number of users, repositories and commits over the last few years; therefore it offers a large data source of scientific codes as well as social interaction of associated scientific programmers.

We present a tool, which allows to easily search through relevant GitHub repositories and perform more advanced analyses, which cannot be conducted solely with the GitHub API. Our tool combines benefits from online as well as offline approaches to retrieve and analyse data to optimise time of execution and consumption of storage.

We discuss possible use cases and demonstrate the tool's capabilities by investigating the popularity of OpenMP directives in the scientific community.

Keywords: GitHub; OpenMP; Software Statistics; Data Mining; Information Search

1. Motivation and Background

Developing scientific software is usually a process, which never really ends: new scientific methods are developed and integrated into the application, new developers join the team to add their expertise on their field of work or the application simply needs to be ported to the next generation of computer architecture. Domain experts across institutes and even across countries collaborate on the same topic and contribute to the same code base. Git hosting services provide a great technical platform for development, maintaining and publishing software to support those developers. They allow the developers to easily release updates to their software, get feedback from their community and publish it under an appropriate license.

* Corresponding author
E-mail address: jannek.squar@uni-hamburg.de

1877-0509
10.1016/j.procs.2022.09.182

Looking at different software projects can provide a very helpful insight of how people use or don't use certain functionality or libraries. Especially when developing service or middleware libraries, one should keep track of how the "customers" make use of the provided features. This can influence how developers continue their work on such libraries, strategical decisions can depend on the behavior of the user community. If the users put their code on the same hosting service as well, this allows to deduce new insights, since all interactions are reflected on this service.

GitHub is a well-established internet hosting service for Git repositories, with over 293 million repositories (public and private) [1] and more than 73 million users. 84% of Fortune100 companies make use of a paid subscription of GitHub (GitHub Enterprise) [2]. While becoming common in industrial software development, the integration of scientific projects on GitHub is not natural though. Research like [3] shows the growing relevance of GitHub. But it is not the only platform, there are other services as well like GitLab or SourceForge. We focus on GitHub for now, because it is the most popular one and promises a great gain in knowledge [4], but we will keep an eye on other platforms too. Furthermore, we take care of applications, which are not hosted anywhere public (for reasons of license or lack of maintenance) as we provide a way to embed offline code bases.

In this paper, we introduce a tool, which allows the user to automatically execute a customised search for specific (code) phrases in a wide range of software projects on GitHub and perform relevant evaluations on those findings. Since there are so many publicly available repositories, which are widely distributed amongst many topics, this gives us a good opportunity to gain a generally valid view on applications. There are several use cases, which can be easily answered for individual repositories by simply manually evaluating them, but this is not feasible anymore if fundamental statements for many more repositories with the same background shall be made. Our tool automatises the data collection and primal evaluation so that the user can focus on the interpretation of the generated results.

2. Related Work

To apply methods of data science on a large data source like GitHub, support through tools is needed to find, retrieve and handle the amount of relevant data. There are many tools available, which ease the collection and analysis of repositories from GitHub. There are two different ways to perform research on GitHub (meta-)data:

offline The GitHub data is completely/partially downloaded and searches and analyses are performed locally.

online A request is generated and used to retrieve data from GitHub itself: Generally links to repositories, which fulfil the request, and their metadata (but not the data within the repository itself) are returned. The request can be submitted by using the official GitHub APIs (*REST* [5] and *GraphQL* [6]) or the more specialised *GitHub Code Search* API [7]. Alternatively a webcrawler could be used.

2.1. Offline

Instead of searching through GitHub repositories on the fly via its API one can also make use of archives, which provide a complete mirror of GitHub. Two notable examples are *GHTorrent* [8] and *GH Archive*[1]. This allows to perform offline analyses on the whole dataset without the restrictions through network and API request limits. But there are several drawbacks to this approach:

1. You have to download the complete dump even if you are just interested in a small part of the data. Currently, in case of GHTorrent this would be about 102 GB solely for metadata; the database, which holds the real data itself is larger than 18 TB [9]. Since both archives are also accessible via *Google BigQuery*, this problem can be avoided; however Google BigQuery offers only a limited subset of features for free use [10]. There have been done improvements to allow users to customise the content of the dump, before it is downloaded, but the creation could take a long time (up to a magnitude of several days) or lead to incomplete data [11].
2. Depending on how you access the archive data, you retrieve a full or partial data dump only at discretised timesteps.

[1] https://www.gharchive.org/

3. The archived data might be outdated, while getting recent data entails additional efforts in form of authentication steps or versions diff-processing.

2.2. Online

To circumvent the drawbacks of the offline search an online approach via the GitHub API or a webcrawler can be used. A webcrawler can be used to perform a so-called *focused crawling*, to retrieve only relevant data from a target website [12]. Such a tool has been proposed by [13], which collects repositories. However, there are several substantial disadvantages to use a webcrawler:

- They rely on a certain layout of the website or the URL, which could change over time.
- If the website has a bot check the webcrawler needs to imitate human behaviour to avoid being (soft) blocked [14].
- The legal aspects of using webcrawlers are still quite controversially discussed [15].

A more robust and with respect to legal aspects better approach is to use the official API, which is provided by GitHub. Constructing and tuning the requests to this API allows to narrow down the volume of the data, which will be requested and then analysed locally. Aside from that using the requests guarantees that always the freshest data will be retrieved, there is no need to update a complete dump with consecutive partial dumps. In fact, it would be optimal to perform the analyse online as well e.g. by using Google BigQuery, but because of its pricing model this is no generally valid alternative.

There are some tools available, which do not only submit general requests via the GitHub API but also allow to reduce and optimise the amount of found repositories by setting user-defined filters [16, 17]. They allow to preselect repositories by setting limits, which they need to fulfil and focus on the repositories' metadata like dates, number of commits, stars, etc. In comparison our tool significantly extends the possible user-set limits and introduces on top of that much more complex analyses (e.g. intersections of multiple search requests).

Another example for a tool, which performs a more complex online analysis is *AUSearch* [18]. It allows the user to look for code examples of a specific Java API. The tool uses the GitHub Code Search API to find corresponding Java source files, processes them and gathers additional information from the Maven Package Search API, to present usage examples to the user in the end. But since this tool is tailored closely to the Java use case and we have more general purpose use cases in mind, this tool does not fit our needs.

Our tool combines an online and offline approach to utilise the advantages of both worlds while reducing their negative impact. We perform a preselection of repositories by generating a fine-granular request, which we submit to the GitHub API. But unlike the aforementioned tools our framework is not limited to metadata filters or a specific language but considers the repository content in general: The returned repositories are cloned to a local disk, where then our in-depth searches and analyses are performed. In addition our framework utilises a caching mechanism. Assuming that a user performs several requests, which could be correlated to each other (because they are interested in a specific topic for example), we suppose that successive user searches hit some repositories several times. Storing some results selectively in a local database allows to speed up following user searches. The advantage of this approach is that our framework does not need to download a large dataset but limits the storage requirements on relevant findings.

3. Use Cases

In this section we describe a few of the wide variety of use cases for our framework. Like other tools, which have already been presented to some extent in section 2, our tool incorporates a module to fetch data from the official GitHub API. But since we integrate a more advanced analysis component, we can perform more complex analyses. To demonstrate the capabilities of the tool we will present two use cases. In section 5 we will show results of the first use case (composition of OpenMP usage) in further detail, though the latter use cases will just be outlined and evaluated in a follow-up publication.

3.1. Use Case I: Popularity of an API

At first we address the question how frequently a specific library feature or functionality is used in real world applications. The outcome could affect the further project development and design if specific functionality is unexpectedly unused. As soon as the project developer gets notice of such an imbalance they can take measures against it. They could improve the documentation of neglected features or offer additional tutorials or examples to make users more familiar with such features. Even more radical measures like changes to the API could be conceivable. But whatever measures are taken the first important step is to take notice that there is a need for action.

Knowing such an imbalance can also be taken advantage of:

- If developers try to improve the code base they could for example focus first on the most popular constructs to achieve maximal improvement regarding their invested working hours.
- If a tool is constructed, which shall perform its actions on specific API constructs; then it's much more beneficial to start with the most popular constructs, to achieve extensive first results quickly.

We will use OpenMP [19] as an example for this use case. It offers an API, which is supported by most established compilers, and offers functions and compiler directives to utilise threads on shared memory architecture. In comparison with other thread APIs like for example *POSIX Threads* or Intel's *Threading Building Blocks*, it is quite easy to use and only requires minimal code changes. Therefore in HPC it is very often used for parallelisation using threads.

Using our tool allows to automatically iterate fitting repositories and to search within each repository for used compiler directives. The results (cf. section 5) can then be used to estimate if a new feature of OpenMP has already been well adapted by the community or if (probably after many years) still additional efforts at persuasion are needed. Developers of tools, which optimise code, if specific (OpenMP) constructs are used, can also benefit from this information. Such an example is `CATO` [20], which performs automatic replacements based on found OpenMP directives. Implementing new replacement patterns in `CATO` should focus on the most popular directives, first. As a result, the new replacement pattern can be applied onto many applications using OpenMP in a timely manner. Otherwise - without any knowledge of the directives' popularity - the selection of which OpenMP directive should be considered first would be more fuzzy and it would take more time to reach the same level of consideration.

3.2. Use Case II:

When developing system components for specific usage, it often lacks for sample applications to benchmark the own work. Developers often use only a few applications they are familiar with, which could cast doubt on the meaningfulness of the limited results. With our tool we can simply create a pool of appropriate applications.

One of our related project deals with file system compression. To compare the feature performance, it is insufficient to use synthetic I/O benchmarks, even if they support compression techniques, since the performance very much depends on the I/O patterns and the computation load of the application. A pool of real world scientific applications using specific data formats like HDF5 or netCDF which allow for compression is required for extended studies.

Additional valuable outcome provides the overview whether scientific applications use compression at all, because they often do not. Status quo studies with our tool can help to argue for the new compression feature, which would not require any application-side code, and also help to find the reasons why the applications do not use it. For example, to allow compression, HDF5 requires specific properties to be set (like `H5FD_MPIO_COLLECTIVE_IO`) and has limitations on data transformations for parallel writes when using compression. If most of the applications appear to perform these transformations, they cannot benefit from middleware compression and have no other option than file system compression. The remaining ones could be used for comparison which feature performs better.

Middleware libraries like HDF5 and netCDF are dominant in oceanographic and meteorologic fields, while other fields like astrophysics and bio-informatics have their own formats. Due to a lack of well maintained documentation, it is very hard to find out whether they use compression features. Code investigations for specific keywords or compression algorithms acronyms can simplify the search.

Developers of middleware, on the other hand, can also benefit from these findings and take the insights as an opportunity for improvements on their software. It can be interesting for strategical decisions to see whether deprecated functions are still widely used, or in opposite, new features remain unused.

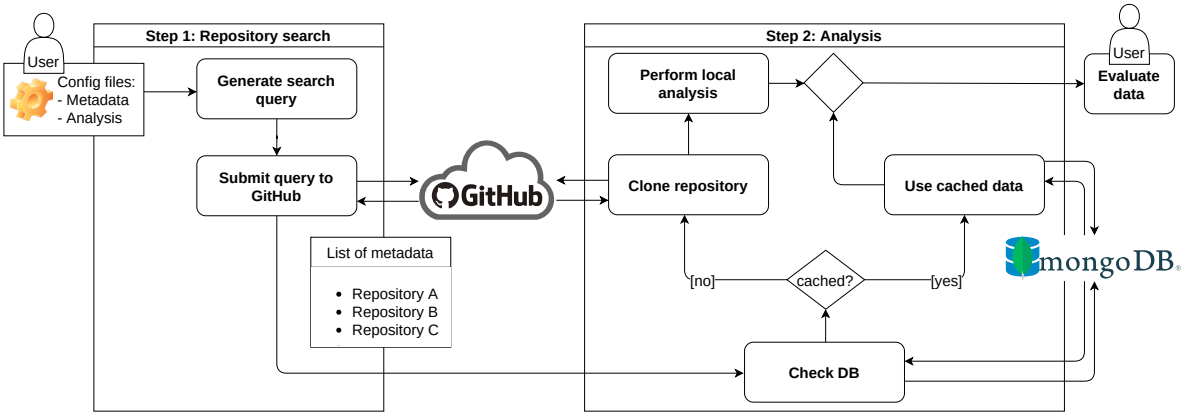

Fig. 1. Component overview and work flow of our tool

4. Tool Design

Our tool consists of several components, which handle the collection of repositories, execute search queries and perform user-defined analyses on the repositories' content. We have already discussed in section 2 online and offline approaches to work on GitHub data. Our tool combines both aspects: We use an online approach to iterate through GitHub to preselect a set of repositories and then perform an offline analysis, which could not be (easily) done with an online approach only. The interaction of the tool's components can be seen in fig. 1 and we will discuss the most important ones:

1. GitHub queries
2. Repository queries
3. Repository analysis
4. Database

The tool's code is publicly available under an open source license on GitHub [21].

4.1. Step 1: Search through GitHub

The first component is concerned solely with constructing and sending the necessary requests. We use the official GitHub API to perform an online search through GitHub's hosted public repositories. GitHub offers two different APIs: REST and GraphQL. We have decided to integrate the REST API first, because it is a well-established paradigm and offers about the same features like the GraphQL API.

When the user executes our tool they may provide a config file with additional attributes, which will be used as a filter in the REST requests. This allows the user to define boundary conditions on the metadata of the repositories so that only repositories are returned, which meet the user's requirements. They can for example limit the search to repositories, which fulfil technical requirements like a specific topic or primary language. But also social limits can be set like the minimal number of forks, stars or the date of the last commit. The configuration files are kept simple, each line is an individual statement, which is interpreted by our tool. Listing 1 and listing 2 show simple examples of both configuration files, which the user can pass to the tool.

```
keyword: openmp
stars: >10
pushed: 2018-01-01..2022-01-01

```

Listing 1. User-defined configuration file for preselection

```
pragma omp barrier
pragma omp parallel for
pragma omp task

```

Listing 2. User-defined expressions for search step

The first one regulates the preselection step: In this example only repositories are considered, which conform to the keyword, the least number of stars and the time of the last commit. This is not only important to limit the result set to repositories, which fit to the user's search intent but also to perform some kind of preselection. Repositories can suffer from several impairments, which degrades their benefit for the user. For example, depending on the user's intentions it could be useful to exclude repositories, which have not been updated for a very long time (content might be outdated) or have only few stars or contributors (the majority of repositories on GitHub are personal and might be of no interest) [4]. All limits are taken into account, when the tool generates and submits the REST requests. The result of this request consists of a list of repositories, which fulfil all set conditions.

After the repositories have been selected, the analysis is conducted by the next component. In this case the second configuration file contains a simple list of OpenMP directives, which are searched within each repository; the result is then saved to the database.

4.2. Step 2: Analyse single repository

There are already many tools, which allow to search through GitHub repositories and define limits regarding the repositories' metadata. Our tool allows a more advanced search by performing in addition a consecutive search on each single repository, which has been found by the previous REST request.

This is where the offline approach of our tool comes into play. The tool temporarily downloads each repository to disk by performing a shallow clone by using a history depth of one, since we are only interested in the current state. Omitting the dispensable Git history saves storage and time. Since the files can now be accessed directly more complex searches and analyses are possible, because the official GitHub APIs are quite limited to filtering based on metadata but not files' content. Currently we have integrated the possibility to search for specific strings or regular expressions within all files, which already offers a large variety of possible search queries. To get an impression what kind of advanced use cases are now possible, we list a few notable ones in section 3.

GitHub provides another API for code searching, without the need of downloading or cloning the code base. However, it has a number of limitations in usage [7] and is currently not available for everyone (as per April 2022). These can be very helpful to prevent escalation in automated processes: Only repositories with less than 500,000 files are searchable or only the default branch is indexed for code search. Some other restrictions on the other hand can be obstructive: A lot of wildcard characters (punctuation marks, brackets) are not allowed to be part of the search. Additionally, the results will include at most only two fragments of a file where the searched term has occurred. This is why we currently refrain from using the code search API.

4.3. Improvements

Since downloading and storing all repositories, which have been selected for advanced analysis, would quickly require a lot of capacity, the cloned data is only stored temporarily. As soon as the analysis has been conducted, the local repository is removed again. This saves disk storage but leads to the disadvantage that all repositories have to be downloaded every time they are selected in the initial GitHub search. Even if only a slight variation regarding the search configuration shall be performed, which would again select repositories from a prior search, everything would have to be downloaded again.

Therefore we extended our offline approach by integrating a database as a local cache. Since we expect to store data, which can be quite diverse, since many versatile search queries can be thought of, we decided to use a document store database and picked therefore MongoDB[2]. Significant metadata of all selected repositories are stored in it as well as results of performed analysis (e.g. all found OpenMP directives from the use case described in section 3.1). For example, storing the date of the last commit allows to define a time duration, after which the locally stored data within the database is automatically invalidated so that the next time, when this repository is selected, it's cloned again and the stored data within the database is updated accordingly. Storing search results in the database allows also to re-evaluate prior search results with no need to download repositories again, which have already been found and analysed. The user is also able to influence the collection in which the queried data is supposed to be stored in

[2] https://www.mongodb.com/

via arguments passed to the tool. Being able to provide this feature is another advantage of using a document store database compared to other possibilities, like relational databases, as the added flexibility enables users to structure their queried data more liberally. With these options, the construction of a vast, up-to-date and usable database is well supported.

Even more complex analyses are now possible, where data is required, which cannot be retrieved in a single search. One example would be to search for repositories, which make use of parallel I/O but do not enable compression even if it would be applicable. This is a crucial pattern, because parallel I/O without compression can lead to inefficient performance. Using only the negated corresponding keyword via GitHub search to check for abundance of compression is not reliable: Programers could use compression without stating this neither in `readme.md` nor in the topics. To rely on the keyword on this case would lead to a lot of false negatives and to avoid this the code has to be searched directly, which our tool is capable of.

5. Results

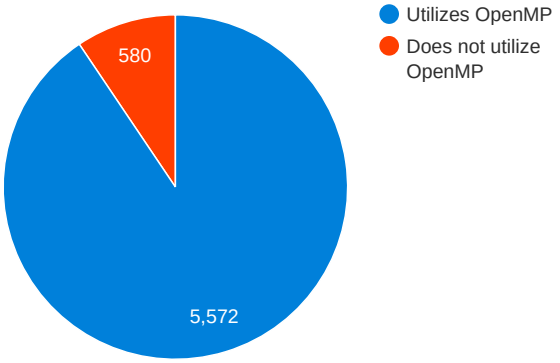

Fig. 2. Share of analyzed repositories that utilize OpenMP

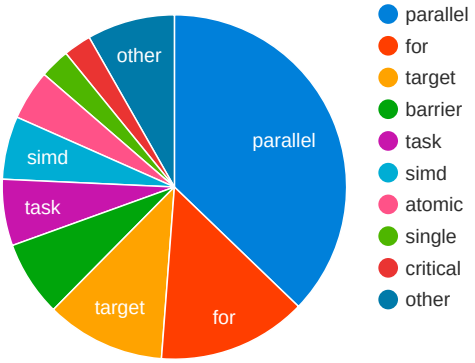

Fig. 3. Distribution of OpenMP pragmas after filtering

To demonstrate the capabilities of our tool, we evaluate the first use case, which we have presented in section 3.1: We check how many repositories make use of OpenMP threads to optimise their application performance-wise. It took about four hours in total and during this process about 6100 repositories have been cloned and analysed locally. The runtime is put together roughly as follows: Constructing the search queries, sending them to the REST API and processing initial metadata takes less than 30 minutes, the rest of the runtime is spent cloning the repositories. The time to search through each repository for the specified queries, which consisted of more than 60 directives, is miniscule and can therefore be neglected.

The outcome can then be used as a data basis for several questions; we will have a look on two:

1. Which OpenMP directives are the most popular ones?
2. Has the `task` directive already been widely accepted by the community?

The answer of the first question can be of interest for tool developers (which directives should be focused on) but also users, who want to start learning OpenMP by knowing which popular feature should be looked at first. In addition you get also a list of repositories, which make use of those directives and can be looked at, if an example of use is required.

The second question could be of interest for tutors, who teach OpenMP. Many trivial parallelisation approaches simply use `#pragma omp parallel for` to parallelise the execution of a loop. Tasks have been an important addition to OpenMP and have already been added with version 3.0 in 2008 [22]. Some even say that new beginners should start to learn tasks and only afterwards parallel loop constructs. Evaluating the second question would then allow an estimation, if additional teaching is required or the directive needs to be pushed more.

At first a set of repositories is created, which is preselected based on metadata. The second step is then the analysis, which OpenMP directives have been used in each repository. We performed a search for the keyword *openmp* and

limited the result set to repositories with at least two stars, whose primary language is C/C++ and which have been pushed to after 01.01.2020. About 10 % of all found repos did not contain OpenMP directives (cf. fig. 2). The reason for this is that the `keyword` search evaluates three characteristics:

- The name of the repository
- The topics
- The content of the `readme.md`

If in one of these places the keyword is matched, the repository is added to the result set. But since GitHub also evaluates hits on substrings as true, there are also some false positive results. For example the keyword *openmp* is also found in the name of the repositories for OpenMPI[3] and OpenMPT[4]. Our analysis step afterwards, which looks for OpenMP directives, improves therefore the data quality further by eliminating some false positives. The more interesting result is shown in fig. 3, which shows the distribution of used directives in all repositories, which have been found and use OpenMP. It becomes quite clear, that `parallel` and `parallel for` are predominant directives. The `task` directive comes only in the fith place.

This could draw the following conclusions:

- The `task` directive needs to be promoted more to reach a critical size.
- `CATO`'s developers (cf. section 3.1) should at first focus on `parallel` and `parallel for` instead of `task`, if they want to reach compatiblity with more codes on GitHub faster.

6. Conclusion and Future Work

We gave a summary of why it can be beneficial to users and developers of tools or libraries to apply data science techniques on (meta-)data from GitHub repositories. Our tool allows to easily search through many GitHub repositories and execute more in-depth analyses, which in combination can result in much more complex results. Many use cases can be investigated, ranging from technical applications (e.g. *"Which repositories use a specific library/function/language?"*) to more abstract questions (e.g. *"How many repositories make use of compression in general?"*). We have demonstrated this on the basis of the OpenMP use case.

But, there are still some aspects, which we are still working on to improve the tool performance and range of its features.

6.1. New Features

Detecting transitive relationships is a challenging task and currently only possible to a limited extent. When looking for applications, which use (implicitly) a specific function, it might be not enough to look for the function call itself because there are many middle-layer libraries available, which prescind the used backend libraries and their functions. Searching for the use of backend functions would just return the middle-layer libraries but not the applications, which call them via the library.

Another point is related to the request backend: Currently our tool uses the REST API to retrieve data from GitHub. Since 2016 GitHub offers in addition to their REST API a GraphQL API [23]. It has several advantages regarding the resource consumption, since the requests can be constructed more precisely in comparison to REST. Multiple REST requests can be merged into a single GraphQL request and the amount of unsolicited data, which are returned by REST, can be reduced. All together it becomes simpler to use and can reduce the load on the GitHub API so that their limits last longer.

We have already laid a good foundation to accomplish these tasks and will extend the interface towards more complex and optimised searchings. Therefore we will work on an update to support transitive searching and offer GraphQL as an alternative request backend.

[3] https://github.com/openmpi/ompi
[4] https://github.com/OpenMPT/openmpt

6.2. Performance Aspects

Potential performance bottlenecks have to be investigated deeper, though we do not expect significant drawbacks. Network traffic and storage space required for cloning a repository should not be problematic in an average case. Though, we have to take care of edge cases where users have very large or dirty repositories. We would like to avoid hard limitations of file sizes, like the code search API partly does (only files smaller 384 KB are searchable), since we expect the software projects to grow in future. We can imagine to filter large repositories beforehand for specific undesirable file formats like binaries, pictures or videos. Cherrypicking of source code files is another possibility to avoid cloning garbaged repositories.

The performance estimation of a specific search request needs to be improved as well. It would allow to evaluate the efficiency of optimisation measures but would also assist the user. Befor they submit a search request, they could be given some kind of estimation of the requirements on time and storage. Then they could decide on their own if they need additional constraints on the search if they want to hold some requirements on storage and runtime.

To accelerate the analysis, different steps will be parallelized. Since GitHub-server requests are limited anyway it makes more sense to parallelize data transfer by means of putting additional instances to dedicated nodes of a cluster with a parallel file system. It needs deeper investigations to what extent the database requests can be a bottleneck and how parallel accesses may affect the workflow. There are other search backends available like for example the GitHub Code Search API or *GHCrawler*[5], which promise efficient accessing and storing respectively updating of (meta-)data. Using them could reduce the load on our database.

We will check if those features are feasible for our tool and how they can be integrated.

References

[1] GitHub, Search more than 293M repositories, online, last accessed: 2022-03-30 (Mar. 2022).
URL https://github.com/search

[2] GitHub, Github octoverse, online, last accessed: 2022-04-04 (2021).
URL https://octoverse.github.com/

[3] X. Lin, M. Simon, N. Niu, Releasing scientific software in github: A case study on swmm2pest, in: 2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science), 2019, pp. 47–50. doi:10.1109/SE4Science.2019.00014.

[4] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, D. E. Damian, An in-depth study of the promises and perils of mining github, Empir. Softw. Eng. 21 (5) (2016) 2035–2071. doi:10.1007/s10664-015-9393-5.
URL https://doi.org/10.1007/s10664-015-9393-5

[5] GitHub, Rest api, online, last accessed: 2022-04-01 (2022).
URL https://docs.github.com/en/rest

[6] GitHub, Graphql api, online, last accessed: 2022-04-04 (2022).
URL https://docs.github.com/en/graphql

[7] GitHub, Searching code, online, last accessed: 2022-04-01 (2022).
URL https://docs.github.com/en/search-github/searching-on-github/searching-code

[8] G. Gousios, The ghtorrent dataset and tool suite, in: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 233–236.
URL http://dl.acm.org/citation.cfm?id=2487085.2487132

[9] G. Gousios, Downloads, online, last accessed: 2022-03-31.
URL https://ghtorrent.org/downloads.html

[10] Google, BigQuery pricing, online, last accessed: 2022-03-31.
URL https://cloud.google.com/bigquery/pricing

[11] G. Gousios, B. Vasilescu, A. Serebrenik, A. Zaidman, Lean ghtorrent: Github data on demand, in: P. T. Devanbu, S. Kim, M. Pinzger (Eds.), 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India, ACM, 2014, pp. 384–387. doi:10.1145/2597073.2597126.
URL https://doi.org/10.1145/2597073.2597126

[12] M. AbuKausar, V. S. Dhaka, S. K. Singh, Web crawler: A review, International Journal of Computer Applications 63 (2) (2013) 31–36. doi:10.5120/10440-5125.

[13] G. Farah, D. Correal, Analysis of intercrossed open-source software repositories data in GitHub, in: 2013 8th Computing Colombian Conference (8CCC), IEEE, 2013. doi:10.1109/colombiancc.2013.6637537.

[5] https://github.com/Microsoft/ghcrawler

[14] D. Zhang, D. Zhang, X. Liu, A novel malicious web crawler detector: Performance and evaluation, International Journal of Computer Science Issues (IJCSI) 10 (1) (2013) 121.
[15] D. K. Mahto, L. Singh, A dive into web scraper world, in: 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), IEEE, 2016, pp. 689–693.
[16] S. Surana, S. Detroja, S. Tiwari, A tool to extract structured data from github, CoRR abs/2012.03453 (2020). arXiv:2012.03453.
URL https://arxiv.org/abs/2012.03453
[17] S. D. Joshi, S. Chimalakonda, Rapidrelease: a dataset of projects and issues on github with rapid releases, in: M. D. Storey, B. Adams, S. Haiduc (Eds.), Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, IEEE / ACM, 2019, pp. 587–591. doi:10.1109/MSR.2019.00088.
URL https://doi.org/10.1109/MSR.2019.00088
[18] M. H. Asyrofi, F. Thung, D. Lo, L. Jiang, AUSearch: Accurate API Usage Search in GitHub Repositories with Type Resolution, in: K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, M. Zhou (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020, IEEE, 2020, pp. 637–641. doi:10.1109/SANER48275.2020.9054809.
URL https://doi.org/10.1109/SANER48275.2020.9054809
[19] OpenMP Architecture Review Board, OpenMP application program interface version 5.2, online, last accessed: 2022-04-01 (Nov. 2021).
URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
[20] J. Squar, T. Jammer, M. Blesel, M. Kuhn, T. Ludwig, Compiler assisted source transformation of openmp kernels, in: 19th International Symposium on Parallel and Distributed Computing, ISPDC 2020, Warsaw, Poland, July 5-8, 2020, IEEE, 2020, pp. 44–51. doi:10.1109/ISPDC51135.2020.00016.
URL https://doi.org/10.1109/ISPDC51135.2020.00016
[21] J. Squar, N. Schroeter, A. Fuchs, M. Kuhn, T. Ludwig, AGSearch, online (Apr. 2022).
URL https://github.com/wr-hamburg/AGSearch
[22] OpenMP Architecture Review Board, OpenMP application program interface version 3.0, online, last accessed: 2022-04-01 (May 2008).
URL https://www.openmp.org/wp-content/uploads/spec30.pdf
[23] GitHub Engineering, The GitHub GraphQL API, online, last accessed: 2022-04-04 (Sep. 2016).
URL https://github.blog/2016-09-14-the-github-graphql-api/