Embedding Visual Software Analytics on the GitHub Platform on the Example of Static Source Code Metrics and Software Maps

Willy Scheibel, Jasper Blum, Fransziska Lauterbach, Daniel Atzberger, and Jürgen Döllner

University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute
{willy.scheibel, daniel.atzberger}@hpi.uni-potsdam.de, {jasper.blum, franziska.lauterbach}@student.hpi.uni-potsdam.de, doellner@uni-potsdam.de

Abstract. Readily available software analytics tools are often operated as external services, where measured software analysis data is kept internal, and no external use of the data is available. We propose an approach for embedded visual software analytics on the GitHub platform by leveraging GitHub Actions and the GitHub API. The proposed approach collects static source code metrics for each commit and augments the commit by the resulting software analytics data that is stored as git objects within the same repository. We show that this approach is feasible to be integrated into software projects as it poses only little overhead to CI runtime and storage overhead. The stored software analytics data is externally accessible to allow for external visualization tools, such as software maps. This enables a large amount of projects to have access to software analytics as well as provide means to communicate the current status of a project.

Keywords: Software Analytics · Continuous Integration · Treemaps.

1 Introduction

During the software development process, a large amount of data is created and stored in the various software repositories. For example, changes to the code are managed in a version control system, tasks are organized in an issue tracking system, and errors that occur are documented in a bug tracking system. Software analytics uses data analysis and visualization techniques “to obtain insightful and actionable information from software artifacts that help practitioners accomplish tasks related to software development, systems, and users” [38]. The applications in which software analytics is used are diverse [26], e.g., effort estimation [28], social network analysis [39], or using visualization to support program comprehension tasks [21,16,27]. Of particular relevance is the analysis of git repos [3], as the most widely used version control system, and GitHub as the most prominent social coding platform [19]. Various platforms have been developed to provide software analytics services to stakeholders [36,7,24]. These
either integrate directly into the CI pipeline or must be controlled from outside [33].

While the techniques and tools are available for open source and industry projects, the processing steps as well as the data storage of software analytics data is usually considered separate to the source code repository. For example, the source code of open source project like Angular can be hosted on GitHub, but regular processing of the source code by means of a CI is added through the use of CircleCI. Although the source code of Angular is publicly available, the derived information by means of source code metrics and higher-level build artifacts are not. Further, the analyzed data is not accessible for further processing by other services. Especially for software visualization, software can be measured during the CI pipeline. However, the detailed results are usually not considered part of the public results. Only a highest-level status – CI is passing or failing – is communicated back to an external developer. Last, some software analytics services allow for regular analysis, but their results are kept internal.

To summarize, current state of the art has the following limitations:

1. Readily available software analytics tools are often operated as external services,
2. where measured software analysis data is kept internal, and
3. no external use of the data is available.

We propose an approach to process software data during the execution of the CI pipeline and store the resulting analysis data at an accessible storage location – the project source code itself – that is available for git projects on GitHub. For this, we propose a default component to run using GitHub Actions, such that software metrics are computed on a per-commit basis. As accessible storage location, we use the git object database and mirror the commit graph structure to augment existing commits with software analytics data. Last, we use the GitHub API to store and load the software analytics data and use them for software visualization (Figure 1). We further perform a case study on a number of open source GitHub projects and show the performance impact on the CI and memory impact on the git repository.

The remainder of this paper is structured as follows. Section 2 introduces related work. In Section 3, we present the use of GitHub and its components for
embedding software analytics. In Section 4, we describe our case study and evaluation of runtime performance and memory overhead of the proposed technique. We discuss the approach and results in Section 5. With Section 6, we conclude this work.

2 Related Work

Our presentation of related work covers three areas. First, we present techniques for visualizing source code as the main artifact of a software project. Then, we give an overview of tools and infrastructures for extracting data from software repositories, which can then be used for Software Analytics tasks.

Software Visualization. Software has no intrinsic shape or gestalt. Software visualization develops techniques for representing software projects’ structure, behavior, or evolution for supporting the stakeholders in different program comprehension tasks. In many cases, the layout of a visualization is derived from a project’s folder hierarchy [32], e.g., when using treemaps [18]. Software metrics can be mapped on the visual attributes of treemaps, e.g., texture, color, and size [14]. Especially, 2.5D treemaps provide many visual attributes, which motivates their use for exploring large software projects [21]. Besides hierarchy-preserving visualizations, layouts can also be generated based on the semantic composition of software projects [1]. In this case, abstract concepts in the source code are captured by applying a topic model, which results in a high-dimensional representation of each source code file. The local and global structures within the high-dimensional representation are captured in a two-dimensional scatter plot after using a dimensionality reduction. By enriching the visualization with cartographic metaphors or the placement of glyphs, software metrics can be mapped in the visualization.

Mining Software Repositories. Version control systems, especially Git, enable collaborative work on software projects. All activities and the entire history of a project are stored in a repository, which provides much information for further analysis. Example applications for analyzing Git repositories include capturing static and dynamic software metrics [15,34,5] or locating expertise among developers [2]. The extraction of relevant data requires efficient processing tools, e.g., for compiling software metrics [22]. An example of such a tool is *PyDriller* which allows efficient extraction of software metrics from a Git repository. By combining different optimizations, e.g., in-memory storage and caching, *pyrepositoryminer* provides an alternative tool, that shows better performance. Other examples are *ModelMine* [30], a tool focusing on mining model-based artifacts, *GitcProc* [4], a tool based on regular expressions for extracting fine-grained source code information, *LineVul* [10], an approach for predicting vulnerability within source code, and *srcML* [6], an infrastructure for the exploration, analysis, and manipulation of source code. In addition to efficiently processing individual projects, it is often necessary to process entire collections of projects,
for example, to generate data for training ML procedures. One of the first attempts to make data from GitHub accessible for research is Boa [9]. Besides the infrastructure, it provides a domain-specific language and web-based interface to enable researchers to analyze GitHub data. Similarly, GHTorrent provides an infrastructure for generating datasets from GitHub [11], which can further be made available for local storage [25]. An infrastructure that also provides a frontend is given by SmartSHARK [35]. A technical hurdle in crawling large datasets from GitHub is the limitation of API requests. Crossflow addresses this problem through a distributed infrastructure [20]. An alternative approach, that relies on the GitHub GraphQL API, was presented by Jobst et al. [17]. Besides source code, other software repositories, e.g., issue tracking systems or mailing lists, are also suitable for collecting information for subsequent analyses [8].

Software Analytics. Various Software as a Service (SaaS) platforms have been developed to gain insights from the development process and support developers in their work. One example is CODEMINE, which was developed at Microsoft [7]. Another example is Nalanda, which comprises a socio-technical graph and index system to support expert and artifact recommendation tasks. It further forms the basis for a software analytics platform with an integrated news feed [24]. HIVISER is another SaaS platform for software analytics tasks [31]. It provides an API for visualizing for tree-structured data on the web.

3 Approach

Apart from readily available tools, infrastructures and full-featured software analytics services, we propose an embedded approach to visual software analytics on the GitHub platform. This approach uses GitHub Actions to provide per-commit software analytics while storing the results as blobs in the git objects database of a project. The results are available for further processing and visualization for internal and external processes, e.g., software visualization (Figure 1). The proposed process integrates into the general GitHub CI process (Figure 2). It is separated into two phases: the analysis phase including storage of the results (1 – 3), and the visualization phase (4 – 5). The analysis phase is started when a developer creates and pushes a commit to the git repository, starting its CI (1). After project-specific analysis (2), the software analytics data is added to the repository as git blob objects (3). This allows to annotate each commit of a repository with project-specific software analytics data, such as source code metrics. Later, this data can be queried and fetched from a client component (4) and used for visualizing the software project (5).

Analysis. The analysis is considered to be part of the CI process. As such, we designed an extension to available CI processes on GitHub by means of a GitHub Action. This action is specifically designed to analyze the software for a given commit (1), e.g., the CI can be configured to execute this action on push to a branch. The general processing approach for this action is to collect the source
code, apply static source code metrics, and store the results. However, choosing metrics for analysis is highly dependent on the used programming languages, the quality goals, and available implementations. As such, we see this as a major point of variation for specific implementations. The interface for GitHub Actions for integrating potential metrics implementations is a Docker container, which allows for a highly flexible use of available tools and own developments of metrics.

Storage. The output of the analysis phase is then stored within the git repository. Specific to our prototype, we use a CSV file format where each line contains the measurements for a source code file, identified by its file path. This file is then committed to the git repository using a commit-specific git refs tree in the location `refs/metrics/{sha}` (Figure 3). This allows to query the metrics file within the `refs/metrics` subtree from a given git SHA later on. For convenience, we create and maintain specific git refs to branches as well.

Although these metric files are created within a Docker container, this container has no direct access to the git repository for committing these files. We use the GitHub API to store these files instead. The API allows to manipulate the git trees and refs using the `/repos/{owner}/{repo}/git/trees` and `/repos/{owner}/{repo}/git/refs` endpoints, respectively. The sequence of requests is as follows. We first create a tree by sending a POST request to the `/repos/{owner}/{repo}/git/trees` endpoint. The API will response will contain a SHA-1 hash of our newly created tree. We then create a reference under `refs/metrics/{sha}`, storing the SHA reference to the tree by sending a POST request to the `/repos/{owner}/{repo}/git/refs` endpoint. This ensures that the blob tree is retrievable for every commit. Last, we populate the tree with the metrics file.
Visualization. In the second phase of our approach, the per-commit software analytics data is available for fetching and visualization. Specific to our prototype, we chose to measure software metrics per file that is organized in a file tree. Thus, we consider this metrics dataset tree structured and derive a software map. Both the fetching and visualization are part of an external component, e.g., a website, that can be served by GitHub as well.

The data retrieval uses the GitHub API as follows. We first request the metrics reference for a certain commit using a GET request to the endpoint /repos/{owner}/{repo}/git/refs. We then use the retrieved tree SHA to request the blob tree at the /repos/{owner}/{repo}/git/trees endpoint. This gives us a tree that stores the SHA-1 reference to the blob containing our metrics data. We use this hash to request the blob using another GET request, this time to the /repos/{owner}/{repo}/git/blobs endpoint. Once we retrieved the blob, the last step is to decode the base64-encoded content of the blob to retrieve the metrics content we stored as a CSV string.

4 Evaluation

We integrated our approach as GitHub Action into eight open source TypeScript projects of various sizes. Then, we benchmarked the performance and resource consumption of this action. Specifically, we compared the size of a single stored metric blob, the pure metric calculation time for all TypeScript files in the repository and the total execution time of our GitHub Action. The integration process consisted of forking and adding the GitHub workflow file to each of the repositories. This took approximately two minutes per project.

Datasets. The projects were chosen to use TypeScript as their main programming language while being either known to the authors or popular within the
Table 1: The TypeScript repositories used for evaluation. The number of files represent the number of TypeScript source code files. The lines of code (LoC) are the lines of code from the TypeScript source code files.

<table>
<thead>
<tr>
<th>Project</th>
<th>Location</th>
<th># Files</th>
<th># LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsae</td>
<td>hpicgs/github-software-analytics-embedding</td>
<td>11</td>
<td>729</td>
</tr>
<tr>
<td>Vue 2</td>
<td>vuejs/vue</td>
<td>148</td>
<td>16538</td>
</tr>
<tr>
<td>webgl-operate</td>
<td>cginternals/webgl-operate</td>
<td>180</td>
<td>43473</td>
</tr>
<tr>
<td>Apollo Client</td>
<td>apollographql/apollo-client</td>
<td>225</td>
<td>80881</td>
</tr>
<tr>
<td>TypeORM</td>
<td>typeorm/typeorm</td>
<td>493</td>
<td>105865</td>
</tr>
<tr>
<td>Babylon.js</td>
<td>BabylonJS/Babylon.js</td>
<td>1548</td>
<td>388588</td>
</tr>
<tr>
<td>Angular</td>
<td>angular/angular</td>
<td>4841</td>
<td>661594</td>
</tr>
<tr>
<td>vscode</td>
<td>microsoft/vscode</td>
<td>3991</td>
<td>1136062</td>
</tr>
</tbody>
</table>

community (see details in Table 1). The size of the projects ranges from only a couple of files with a few hundred lines of code to almost 4k source code files with above 1.1M lines of code. The eight selected projects are:

- **gsae** github-software-analytics-embedding is the repository containing the analytics component and treemap visualization frontend presented in this paper.
- **Vue 2** The library Vue 2 is the third most popular JavaScript frontend framework for building UI and single-page applications and has over 200k stars on GitHub. It is open-source and features a model-view-controller architecture and full TypeScript support.
- **webgl-operate** This project contains a TypeScript-based WebGL rendering framework with focus on applications in teaching, research, and prototyping.
- **Apollo Client** The Apollo Client projects provides a state management library written in TypeScript to manage both local and remote data with GraphQL.
- **TypeORM** TypeORM is an Object Relational Mapping (ORM) library for the JavaScript ecosystem written in TypeScript that allows users to interact with databases in an object-oriented fashion.
- **Babylon.js** Babylon.js is a powerful web rendering engine and features full typing support.
- **Angular** Angular is a well-known platform and frontend framework for building mobile and desktop web applications developed by Google.
- **VS Code** The VS Code project provides is a source-code editor developed by Microsoft for Windows, Linux and macOS.

**Visualization Technique.** We visualize the tree-structured software analytics data using a software map (Figure 1). The software map visualization technique is a 3D-extruded information landscape that is derived from a 2D treemap layout. The tree structure for the treemap layout is thereby derived from the tree structure of the file path. The available visual variables in the visualization are footprint area (weight), the extruded height (height) and leaf color (color).
Fig. 4: A screenshot of the prototypical client, showing the webgl-operate project.

Implementation Details. Both the analytics component and the viewer are available on Github in the project github-software-analytics-embedding\(^1\). Additionally, we provide the GitHub Action on the market place\(^2\). Adding this action to a repository enables the integration of the prototypical TypeScript source code metrics for new commits. An example client\(^3\) that is build with React is hosted on GitHub Pages (Figure 4). Our analytics module is written in TypeScript. We decided to use TypeScript as a programming language because it provides first-citizen support for TypeScript code analysis using the TypeScript compiler API. The analysis code first creates an abstract syntax tree (AST) for each TypeScript file in the specified repository path. The AST is then be used for our static code analysis. We decided to focus on a few simple software metrics, which include:

- Lines of Code (LoC)
- Number of Comments (NoC)
- Comment Lines of Code (CLoC)
- Density of Comments (DoC)
- Number of Functions (NoF)

The LoC metric returns the total number of source code lines a source file contains. NoC counts the occurrence of comments, counting both single-line com-

\(^1\) https://github.com/marketplace/actions/analytics-treemap-embedding-action
\(^2\) https://github.com/marketplace/actions/analytics-treemap-embedding-action
\(^3\) https://hpicgs.github.io/github-software-analytics-embedding
Fig. 5: Comparison of TypeScript projects with increasing size and complexity using treemap visualisation. The number of lines of code (LoC) is mapped to weight, the number of functions (NoF) is mapped to height, and the density of comments (DoC) is mapped to color.

Visual Examples. By navigating to specific commits in the client, a developer can get a quick overview of a software project using the software map visualization (examples in Figure 5). For example, the complexity of a software project by means of its subdivision into directories and files, and source code volume by number of lines of code is directly depicted. Further, the software map provides a recognizable representation.

Repository Memory Impact. The memory footprint of an analysis of a single commit scales linearly with the number of files within a project (Figure 6a). As the memory footprint seems rather high for large software projects as Angular or VS Code, smaller projects can profit from a low-threshold software analytics component. Further, the per-commit size is a trade-off between a full metrics file of all files and their metrics and only a metrics file for all changed files. While the
former approach allows to fetch all metrics for all files at once, which is especially suitable for visualization, the latter approach allows for a much smaller memory footprint.

**CI Execution Time Impact.** The time our metrics computation took scales linearly with the lines of code of a project. Even for large projects such as VS Code and Angular, the time to measure all files is limited to a couple of seconds (up to 11 for VS Code). However, the analysis component with the specific metrics is not meant to be used in actual real-world settings. For one, the limitation to TypeScript is strictly limiting the applicability to general software repositories. Further, metrics as well as their implementation was chosen for demonstration purposes. For an actual integration, we refer to more sophisticated software analytics tools that are all feasible for integration into the Docker-based GitHub Action.

## 5 Discussion

The proposed approach specifically focuses the cases where no external service should be used, the measures software analysis data is not kept internal, and the data is publically available. Thereby, the prototypical implementation focused on a small set of features, where broader considerations are desirable as well. While we focused on GitHub as a project management platform, GitHub Actions for the integration, git as the version control system, mid-sized software projects as examples, JavaScript and TypeScript as programming languages, and static source code metrics as software measurements, each of these specifics are target for alternatives and extensions.
**Software Development Infrastructure.** Although we focused on GitHub, GitHub Actions, and git for our prototype, the approach is applicable for differing development setups as well. For one, the proposed architecture and dataflow can be implemented using different techniques. The visualization expects CSV-files that are fetched from a location that serves the CSV files as augmentations to commits. The integration into the git project builds upon the general use of a git database and their internal objects. Alternatives are an explicit database or external file share, although this would divert the solution towards the current state of the art. The execution of the software analysis process is designed to be executed by the CI pipeline, but we also imagine to let a developer execute the software analysis process on its own machine. However, we deliberately focused on this kind of setup as it allows for an integrated, extensible software analytics process. For this, GitHub as project management platform was especially suitable as it combined all of the required features for fetching, augmenting, and integration processing.

**Languages and Metrics.** Extending on our focused set of languages and metrics is expected and a required step for broader utility. With our approach, the metrics are parsed during the CI pipeline, which executes a Docker container with arbitrary program content. The main interface of the Docker container is the availability of the source code as input and an augmentation of the current commit as an output. Its implementation can be chosen at will. As such, the broad list of software analytics tools and own implementations become feasible [12]. The tools using during the CI pipeline define the possible languages and available metrics. Our approach specifically specifies the usage of the GitHub API to augment the commits within the git repository. From a purely technical point of view, such a CI pipeline can trigger external services for metrics computation as well, but this would divert the approach from our goals.

**Scalability.** Scalability for this approach is challenging as GitHub wants to ensure continuous service for all its users. The GitHub CI pipeline is strongly limited in available execution time for non-paying users. Further, the git repositories are limited in size. Executing the metrics computation process for each and every commit and storing the full dataset in an ever-growing software repository is bound to reach those limits. As such, we see primarily the adoption of our approach for small and mid-sized open source projects that are otherwise struggling to setup their own software analytics pipeline, e.g., using external services. Light-weight alternatives that could be applied individually and on-demand would include:

1. Exchange the GitHub CI pipeline with an external CI service.
2. Execute the software analysis process on a developer’s machine.
3. Store the measured software data in a separate repository.
4. Use an externally hosted website for the viewer.

**Security Considerations.** However, with our approach, we strive for low-overhead direct access to abstract software information. The proposed public, side-by-side
availability of software metrics is subject to security considerations as the measured software may represent sensitive information. The targeted use cases for our approach are open source repositories that want to apply lightweight software analytics on their already public source code. This public availability makes these repositories subject to external source code mining on a regular basis [23]. Anyone with software mining tools can download the source code, derive software metrics, host them anywhere, and analyze at their discretion. We’d argue that any security-related attack vector is introduced with publishing the source code and not with making own software metrics available. On the contrary, with our approach, we connect to the original idea of developing source code publicly. A broad community can participate and ensure a more healthy software development process and thus a more healthy software project. One adaption to our approach to protect the measured software data is to use an external database. This adaption, however, would prevent other use cases such as public availability of visualizations of the software project. Security considerations in the area of open source development remain their own field of study [13,37].

6 Conclusions

When a software development team wants to integrate software analytics to their project, selecting tools or services are a trade-off which usually results in (1) no control over metric computation, or (2) no persistent availability of low-level analysis results. We propose an approach to augment git commits of GitHub projects with software analytics data on the example of TypeScript projects and static source code metrics. The analysis is performed as part of a GitHub Actions CI pipeline, whose results are added to git project as own commits. These results are thus persistently stored within the project and accessible through the GitHub API. We used this approach to visualize GitHub projects using a basic React client. Thereby, the choice of analysis tool is designed to be exchangeable.

The proposed approach allows for small and medium-sized projects to introduce low-cost and low-effort software analytics into their development process. As another result, a broad integration of software metrics into the git repository would change availability and use of the data. For example, large scale evaluations of source code metrics can profit from already computed metrics within each repository [29]. Further, dedicated software analysis data repositories can be either derived directly from the software repositories, or be considered distributed datasets instead [23].

For future work, see a replacement of the analysis component to use one with a broad support for programming languages and software metrics. More specifically, we see the other areas of software metrics – dynamic metrics, process metrics, developer metrics – as well as higher-level KPIs that should be added. Further, the augmentation of a git commit by allows to shift the execution of the analysis to the committer, i.e., before pushing a commit, an the analysis component can analyze the commit and add the data to the repository.
Acknowledgements This work was partially funded by the Federal Ministry of Education and Research, Germany through grant 01IS20088B (“Knowhow-Analyzer”).

References


39. Zhang, W., Wang, S., Yang, Y., Wang, Q.: Heterogeneous network analysis of