# Aggregation of Development Data for Inner Source Software Development

BACHELOR THESIS

**Stefan Pfahler**

Submitted on 25 September 2023


Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Stefan Buchner
Prof. Dr. Dirk Riehle, M.B.A.


Friedrich-Alexander-Universität
Faculty of Engineering

# Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

---

Erlangen, 25 September 2023

# License



---

Erlangen, 25 September 2023

# Abstract

Besides the Open Source approach for software, there is a steadily growing adoption of the Inner Source concept for projects in the software development domain. Inner Source promotes an Open Source way of thinking inside the boundaries of a company and enables closer collaboration and code reuse among different projects. To get a better grasp on how well Inner Source performs and also potentially support projects in improving their Inner Source processes, development data of projects should be gathered and analyzed. This thesis focuses on extracting, cleaning and unifying development data from various data sources to provide a foundation for various types of analysis.

# Contents

# List of Figures

# Listings

# Acronyms

**API** Application Interface

**CICD** Continuous Integration Continuous Deployment

**DevOps** Development and IT Operations

**ETL** Extract-Transform-Load

**HTTP** Hypertext Transport Protocol

**JSON** JavaScript Object Notation

**REST** Representational State Transfer

**SDK** Software Development Kit

**SVN** Apache Subversion

**URI** Unique Resource Identifier

**MSR** Mining Software Repositories

**YAML** Yet Another Markup Language

# 1 Introduction

The term *Open Source* is a relatively new term in software development. Although the idea of free software reaches further back in time, the official *Open Source Definition* is based on the Debian Free Software Guidelines which were established in 1997. (Perens et al., 1999) Debian itself is a Linux distribution that was entirely built on free software and had issues with defining what *free* actually meant, hence the release of said guidelines.

The Open Source definition expands on the free software movement by demanding criteria not only focusing on code. (Perens et al., 1999) Today, projects that follow this Open Source definition are often hosted on various platforms, such as GitHub. (Alamer and Alyahya, 2017) GitHub has had large success in the recent years judging by their user count which reportedly has grown from 3.5 million users in 2013 (Lima et al., 2014) to over 100 million in 2023 (Dohmke, 2023).

Besides only hosting Open Source software - based on the Git version control system - GitHub offers additional features to its users. These include among others: project management capabilities, automatic testing and code deployment. Another Git hosting platform that provides this feature set is GitLab. GitLab is an Open Source project itself and in addition allows its service to be self-hosted. (Schreiber et al., 2021)

By leveraging the self-hosting feature of GitLab, many companies try to project the benefits and culture of Open Source onto their own projects. The movement of doing so is termed *Inner Source*. (Capraro and Riehle, 2016) It promises multiple benefits compared to a conservative approach of software development in companies. Some of these are "more successful reuse [of code, ] more flexible utilization of developers [and the] overcoming of organizational unit boundaries". (Capraro and Riehle, 2016)

To prove these claims and back the adaption of *Inner Source*, development data of software projects should be extracted and analyzed. In the scope of this thesis, plausible data sources were identified and requestors for them were implemented as Extract-Transform-Load (ETL) pipelines. Based on the extracted data, a metric was computed to showcase the datas usefulness for analysis purposes.

# 2 Literature Review

To present an overview of the work that has already been done on the topic of gathering and analyzing software development data, this section features a collection of related scientific publications.

Studies have been conducted showing that developers often times have information needs in software projects. Ko *et al.* identified some of these needs through interviews and pointed out that automating information acquisition could prove useful. They also concluded that there is a demand for innovative tools that help in these regards. (Ko et al., 2007)

Adding to the work of Ko *et al.*, Buse and Zimmermann present information needs of project managers as they also depend on meaningful project insights to be able to make good decisions. (Buse and Zimmermann, 2012) Through surveys of 110 software engineers and project managers, they found that part of these insights can be delivered by artifacts generated throughout the lifetime of a software project.

Working out software project insights has already been done for quite some time, as Weiss and Basili discussed on how to obtain information since 1979. (Weiss and Basili, 1985) They followed up on this work by presenting insights on a data collection from five different software development projects, mostly concerning software errors.

In 1993, Weller conducted code inspections also focusing on defects in code. (Weller, 1993) Through these inspections and by feeding insights from them back to the software projects, developers and project managers were able to improve product quality and also their decision making.

The approaches of Weiss and Basili, as well as Wellers, mostly depended on manual data acquisition which proves to be quite time consuming. In modern software development, projects accumulate information by default as they often use a multitude of software tools, ranging from issue tracking programs up to version control systems that generate data. Olatuji *et al.* presented a comparative analysis of Mining Software Repositories (MSR) tools that allow automated ana-

lysis of software development projects based on such data, including SoftChange (Crawford et al., 2003), Hipikat and Dynamine.(Olatunji et al., 2010) Another similar analysis was done more recently by Siddiqui and Ahmad covering many of the same MSR tools. (Siddiqui and Ahmad, 2018) In more recent years, additional mining projects emerged. Counting to them are among others Gitana (Cosentino et al., 2018), Kibble (Apache, 2022) or GrimoireLab (Duenas et al., 2021). Toolsets like the ones mentioned provide a variety of options for collecting data while also providing means of analysis and visualization. (Duenas et al., 2021)

Judging from these scientific publications there was and still is a high demand for software development data analysis. This analysis is mostly done in the context of data mining, which usually requires large data sets. GrimoireLab and other projects try to provide means of extracting data sets from various data sources, such as GitHub or GitLab, but usually miss extracting some worthwhile information.

This leaves room for further improvements and implementation efforts regarding data extraction. The decisions on what is going to be implemented will mainly revolve around the GrimoireLab toolset.

# 3 Requirements

To be able to define a scope of what is to be implemented for this thesis, it is important to define some requirements. There are two types of requirements to distinguish - functional and non-functional ones.

Functional requirements describe mandatory features a result has to cover. If one or many of these are not met, it can safely be assumed that the aimed for goal was not achieved. Non-functional requirements further define properties concerning quality or constraints inside of which the thesis results should operate (Glinz, 2007).

The following two chapters define both functional and non-functional requirements this thesis should meet.

## 3.1 Functional Requirements

**Complete Data Extraction of Chosen Data Sources:** The data gathering part of the implementation should cover all endpoints, which serve currently unsupported development data. The decision on whether a data endpoint is viable or not remains subjective, thus it should be discussed at least once with the supervisor of the thesis.

**Cleaning of Data:** The Application Interface (API) responses of a data source are prone to contain data that might not be necessary for analysis. This type of data should be identified and removed.

**API for Calling the Functionality:** The implementation should not be isolated, but callable by other users or programs. For this to be possible, it should provide an API that allows configuration and execution of its functionality.

**Implement a New Data Metric:** To prove the viability of the newly gathered data, a metric should be computed showcasing exactly that. The metric has to have a reasonable meaning that may e.g. be used to optimize a process within a software project.

## 3.2 Non-Functional Requirements

**Easy Onboarding of New Developers:** As the implementation is meant to be used in a running project, the ease of onboarding new developers should be taken into account. By reducing onboarding time, developers can transition to be effective project members more quickly. The architecture and design of the program should thus avoid complexity and be as easily understandable as possible.

**Seamless Integration of Additional Data Sources:** With time a need for the integration of additional data sources may arise. To keep adding data sources as simple as possible for the developers, this should follow a simple process.

**Lightweight and Intuitive API Design:** The API is the only interface a developer has to the implementation. For a convenient developer experience this API should be lightweight and intuitive to use.

**Comprehensible User- and Developer Documentation:** Good documentation is often times crucial for the adaption of a library or program, as it minimizes errors and frustration. By providing a thought-through documentation, the program should become even more accessible to new users and developers.

**Make Use of the Provided Pipeline Framework:** With the goal in mind that the implementation results of this thesis are going to be used in a project and multiple people might work with it, the code should follow common patterns. To guarantee this, a provided ETL pipeline framework should be utilized.

# 4 Analysis

Before implementing a program that fulfills the previously mentioned requirements, a few questions have to be answered.

- What functionality does GrimoireLab currently support?
- Which data sources are viable for an implementation?
- How can data be extracted from these data sources?
- How do their API responses look like - can some be semantically unified?
- What would be a good way to visualize a metric?

For this, an extensive analysis was conducted that provided a ground for future work done on the practical part of this thesis.

## 4.1 Capabilities of GrimoireLab

GrimoireLab is the main tool that the supervisor currently uses to extract and analyze data. The data (sources) implemented by GrimoireLab can be put into a range of different topics. The official GrimoireLab website gives a quick overview on what is already supported by providing a short list of said topics (Fig. 4.1).

As this thesis mainly focuses on retrieving the development data of software projects, the relevant topics consist of **Source Code Management**, **Issues / Task Management** and **Source Code Review**.

GrimoireLab already supports a vast range of data sources that would have also been viable for an implementation on behalf of this thesis, namely **Git**, **GitHub Issues (and Events)** and **GitLab Issues**.

The service in GrimoireLab that is responsible for gathering data is called Perceval. As the code for Perceval is open sourced, a more elaborate look at the implementation of the GitHub Issue Events requestor showed, that not all GitHub events are accounted for.

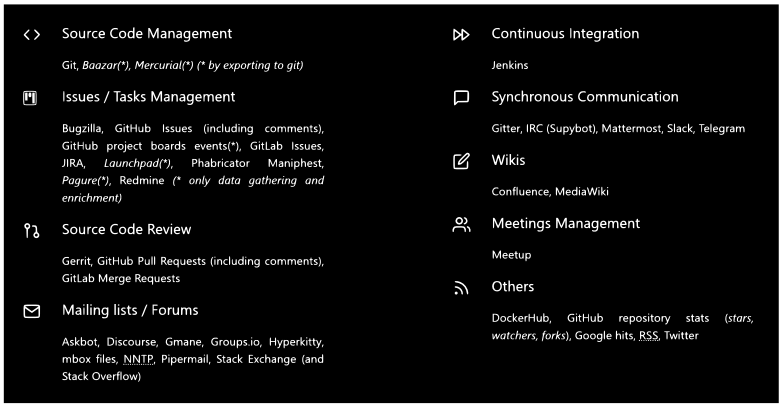


**Figure 4.1:** Screenshot of GrimoireLab Supported Data Source Topics

In order of getting an even better overview of GrimoireLabs coverage, a publicly hosted analytics dashboard was browsed. The dashboard uses the GrimoireLab toolset to extract and visualize demo data from a range of data sources. Supported data sources that were identified that way include **GitHub Repositories** and **Git Releases**.

## 4.2 Research of Viable Data Sources

With having the data sources and API endpoints in mind that are already supported by the GrimoireLab tool chain, additional viable data sources and API endpoints can now be identified.

A starting point for doing so is g2.com, which is a website that gathers reviews of software- and service users to derive data that enables others to find the best fit software for their specific use case. For this thesis, data of interest could be discovered by browsing the category **Version Control Hosting Software**. After analyzing the search results, following list of version control hosting software was considered. (g2.com, 2023)

- Azure DevOps Server
- Bitbucket
- GitHub
- Gitlab
- Jetbrains Space
- JFrog
- SourceForge

**Azure DevOps Server** is a viable option for implementation in the context of this thesis, but was not prioritized after consulting with the supervisor.

**Bitbucket** does not have a broad REST API to fetch data from and was not prioritized for this thesis.

**GitHub** is besides GitLab and Bitbucket one of the most used hosting platforms available. It provides an extensive API that is already partly implemented by GrimoireLab. As GitHub hosts many large Open Source projects and can also be used for Inner Source practices it is a good fit for this thesis.

**GitLab** is a Git hosting platform with a broad set of features that also enable Development and IT Operations (DevOps) and project management processes. It is used by companies for Inner Source development which makes it an eligible option for further implementation.

**Jetbrains Space** provides an extensible REST API to gather data from, but is mainly focused on closed teams. As it seems like there is no support for open- or Inner Source development planned, it is not a suitable implementation candidate.

**JFrog** is a widely used DevOps and Artifactory platform. It currently doesn't support any project management related functionality such as issue boards and thus is not prioritized.

**SourceForge** is not particularly used for hosting repositories, but rather binaries or executables. Git or Apache Subversion (SVN) repositories are at most linked to, which eliminates SourceForge from being a viable implementation candidate.

Besides the version control hosting systems that are predominantly built for Git, SVN also makes for an interesting data source. SVN predates Git and there are assumably also still companies that use it for version control.

After evaluating each of the considerable data source options, reasonable ones ended up being GitHub, GitLab and SVN. Only limited effort will be put into the parsing of SVN commit logs, as there's also a SVN to Git repository migration possible. This migration allows any SVN repository to be analyzed by the existing Git capabilities of GrimoireLab.

### 4.2.1 GitHub

GitHub offers a vast REST API that lets users request data ranging from communicational and organizational data to development data. The latter is the one interesting for this project. After perusing the API, the following endpoints were selected for implementation after also discussing them with the supervisor.

- Commit Statuses
- Deployments and Releases

- Environments
- Issue Events (missing ones)
- Workflows and Workflow Runs

### 4.2.2 Gitlab

While not being as elaborate as the GitHub API, the GitLab API still provides many endpoints with semantically similar or even the same data as GitHub. Most of the selected development data endpoints are - even though with slightly different names - covered by the list in the GitHub section.

### 4.2.3 REST and GraphQL APIs

Both GitHub and GitLab offer a Representational State Transfer (REST) API as well as a GraphQL API to retrieve data from. To be called a REST API a programming interface has to implement specific behavioral characteristics. (Fielding et al., 2017) REST is closely coupled with the Hypertext Transport Protocol (HTTP), which is e.g. used to power web services.

The protocol is built upon so called resources. Every resource is identified by a unique address. That said, a GitHub API address that points to a list of commits of a GitHub REPO(sitory) created by some OWNER would be **https://api.github.com/repos/OWNER/REPO/commits**. As many programming runtimes support the HTTP protocol natively, there is often no need for additional libraries to connect to a REST API.

GraphQL on the other hand is a language specifically invented for querying a programming API. As it is not based solely on commonly known technologies, additional libraries have to be installed and some extra learning effort is required to get it running.

The key difference between REST and GraphQL APIs is their philosophy on requesting data. The resources for a REST API are statically **predefined by their developers**. If there are for example two different resources for *users* and *commits* and the goal is to have this data merged, two API requests would have to be issued to accomplish this. GraphQL servers on the other hand only offer a single endpoint that accepts queries. These queries are **user generated** and highly dynamic. For the same task of merging *user* and *commit* data in GraphQL, it would be sufficient to construct a single query and thus issue a single API request. (Hartig and Pérez, 2018)

The choice between using the REST or the GraphQL approach comes down to a question of complexity. As GraphQL requires additional learning effort, it is

not suitable for the scope of this thesis. Using the REST APIs seems to be more feasible.

### 4.2.4 JSON as the Main Data Format

Responses from both GitHubs and GitLabs REST API are mostly messages in the JavaScript Object Notation (JSON) format. The JSON format is, just as HTTP, a widely adopted standard in the web service domain. (Peng et al., 2011) A JSON object consists of key-value-pairs that are used to store information.

```
{
  "author": "John Doe",
  "sha": "9d840faf0c1c1cc283ed9349b86a7a5a1838cc9f",
  "created_at": "2023-07-31T11:33:23Z"
}
```

**Listing 4.1:** A Simple JSON Object

This example shows how some basic commit information could be encoded in JSON. This specific commit has a sha starting with *9d840faf* and was issued by *John Doe* on the *31.07.2023 at 11:33 UTC* time.

### 4.2.5 Unification of Data Responses

While different vendor APIs may practically offer the same data, their structuring of data might still be completely different. This is also the case for two of the data sources to be implemented - GitHub and GitLab. While they e.g. both offer an endpoint to retrieve information about issue label changes, their API responses differ drastically. The GitHub API for instance returns many attributes that are not mandatory for data analysis. These include among others links to user related resources.

To not trouble a data analyst with polluted data of this kind and enable him to only focus on important information, some data cleaning has to be conducted. The task of data cleaning in this context means removing unnecessary attributes on one hand and the introduction of a common naming scheme on the other. The latter is most important as attributes conveying the same information may have received different names from their vendors and should not be confused to resemble different things.

A simple example that demonstrates this, is the handling of user information. In GitHub JSON responses, a user is referenced through different terms that range from *creator* over *reviewer* to *author*. GitLab on the other hand almost always refers to a user as *user*. Such differences are most likely not exclusive to the user term, but could affect many more.

GrimoireLab has addressed this issue by introducing a common vocabulary for core terms. This vocabulary acts as an abstraction for the vendors individual naming schemes. By reusing it and introducing new common terms for not already defined ones, API responses should be cleaned into unified data objects. This task is to be done for all responses that share a similarity with another response from a different data source, e.g. issue label changes.

## 4.3 Visualizing a Data Metric

Besides integrating data sources and cleaning their data responses, another goal of this thesis is to compute a meaningful metric. Not visualizing said metric though would mean trouble interpreting it. That said, it's reasonable to find a fitting visualization tool to present the metric properly.

There is a potentially unmanageable amount of libraries that are applicable for this use case. To narrow the options down, a few constraints had to be made. First, the library should be **accessed on a high level** only. High level in that case means, that the visualizing process should be powered by writing configurations rather than code. This should minimize the learning effort and thus decrease complexity. Also, if possible, the library should have a **scientific background**.

After conducting a research with these constraints in mind, two libraries seemed to fit the criteria.

- Observable Plot
- Vega Lite

**Observable Plot** is - judging by its GitHub repository insights - a relatively young project, having its first commit in 2020. It is developed by the same team that maintains the D3.js (Bostock et al., 2011) library and is also powered by it. D3 itself is one of the most widely adopted JavaScript charting libraries and is also used to power others, similar to Observable Plot (C3.js, NVD3.js). It is based on scientific work of the Interactive Data Lab. (IDL, 2013)

**Vega Lite** (Satyanarayan et al., 2017) on the other hand is not based on D3, but built from ground up. It was also first announced and developed by members of the the Interactive Data Lab. (IDL, 2013)

Both of these libraries serve a very similar purpose and an official article from the Observable Plot team also states that their product is indeed inspired by Vega Lite. (Johnson, 2023) The article further explains the differences between the two and ends with the statement that Observable Plot is still evolving, while Vega Lite has been used over the past decade and thus is more mature. Based on this fact, the library used for visualizing the metrics will be Vega Lite.

## 4.4 Conclusion of Analysis

After researching the available data sources and arguing their viability for the project, the relevant ones that are to be implemented are GitHub, GitLab and SVN.

An analysis of GitHubs and GitLabs APIs and JSON responses suggested to make use of their REST interfaces and to conduct some data cleaning in order to generate unified data objects, only containing crucial information. Finally, to visualize a metric, Vega Lite will be used.

The next task, following the research and analysis of the projects key components, is getting familiar with the proposed code architecture. A current state of the art approach for extracting, transforming and saving data, is using ETL pipelines.

# 5 Architecture

Software architecture in principle is the description of a systems gross structure. This structure is most often displayed in one or more views that explain design decisions, interactivity between components and the key properties of a system. (Garlan, 2008)

Through this kind of explanation, software architecture is able to assist in multiple aspects of software development. It helps in better understanding a system as a whole, promotes component reuse and can provide a blueprint for development efforts.

There are several different software architectures suited for various use cases. (Richards, 2015) An architecture often used in the context of extracting and manipulating data, is the ETL concept. (Raj et al., 2020).

## 5.1 The ETL Concept

A program that follows the ETL concept is split into three phases - the extraction, transforming and loading of data. Each of these phases has a distinct job that its meant to fulfill. (El-Sappagh et al., 2011) The following phase explanations concentrate on their core tasks only.

### Extraction

The extraction phase is responsible for requesting data from potentially multiple different data sources. This phase is usually split into initially fetching all data possible from a data source and requesting solely data changes thereafter. (El-Sappagh et al., 2011)

### Transforming

After extracting data, it has to be processed. This is done by e.g. cleaning the data and also conforming it in a way that eliminates any ambiguities or inconsistencies it may present. (El-Sappagh et al., 2011)

#### Loading

The last phase of the ETL concept is called *loading*. Its job is to store the previously transformed data into target structures that are then accessed by applications and users alike. (El-Sappagh et al., 2011)

To assist in writing ETL typed programs that implement the mentioned three phases, a pipeline framework was provided prior to the start of this thesis.

## 5.2 The Provided ETL Pipeline Framework

The pipeline framework that this section is devoted to, was not written in the scope of this thesis, but provided by the supervisor. The basic motivation and design decisions of the implementation will be explained shortly, based upon provided documentation and discussions with the supervisor.

### 5.2.1 The Basic Concept

Using the pipeline framework, one should be able to construct pipelines which consist of multiple jobs. To resemble this in code, the framework offers the classes *Pipeline* and *PipelineStep*.

The responsibility of the Pipeline class is to coordinate the execution of multiple pipeline steps and pass arguments between them. The PipelineStep class itself is a generic wrapper for the actual ETL concept implementation, as it can incorporate any of its three phases - extraction, transforming and loading.

As the PipelineStep is only a wrapper for the actual logic, the framework foresees classes to be implemented that are derived from some abstract classes - a requestor, transformer and writer. These abstract classes have a one to one mapping to the ETL concept - the requestor being responsible for the extraction phase, the transformer and writer for transforming and loading, respectively.

Contained in their class definitions are certain methods that classes derived from them should or can implement.

```
1 class RequestImplementation(ABC):
2     @abstractmethod
3     def sync_request(self, pipeline_step, custom_args):
4         pass
5 
6     def delta_request(self, pipeline_step, custom_args):
7         pass
8     ...
```

**Listing 5.1:** The Abstract Requestor Class

An implementation (e.g. a GitHubIssueEventsRequestor) derived from a requestor could e.g. support the *sync_request* or the *delta_request* method. These two methods in particular resemble the phases an extraction process of an ETL pipeline should consist of. (El-Sappagh et al., 2011)

The same idea and steps for implementing a requestor goes for transformers and writers alike.

### 5.2.2 Orchestration of an ETL Pipeline

After implementing the required abstract classes, one can proceed in assembling a pipeline. This is done by creating objects of said classes, wrapping them inside *PipelineStep* objects and finally feeding them to a *Pipeline* object.

```
pipeline = Pipeline()

request_impl = SomeRequestor(...)
request = SyncRequest(request_impl)

transform_impl = SomeTransformer(...)
transform = CleanDataTransformer(transf_impl)

write_impl = SomeWriter(...)
write = Writer(write_impl)

step1 = PipelineStep(request, "request")
step2 = PipelineStep(transform, "transform")
step3 = PipelineStep(write, "write")

pipeline.add(step1)
pipeline.add(step2)
pipeline.add(step3)

pipeline.execute()
```

**Listing 5.2:** Assembly of a Pipeline

This illustration shows an example of how a pipeline might be assembled. After calling the *execute* method on the *Pipeline* object, it handles the sequential execution of pipeline steps 1 to 3. Any number of *PipelineStep*s can be processed by a single *Pipeline* object and individual steps can also be reused for assembling other pipelines.

# 6 Design and Implementation

With knowledge on what an ETL pipeline is and on how the provided pipeline framework can be used, a program can now be designed and implemented.

## 6.1 Design

The goal the implementation has to accomplish can basically be divided into two tasks.

1. Fetch and Clean Data
2. Compute a Data Metric

Multiple, but still a fixed amount, of data sources are to be implemented in the scope of this thesis. As for the metric computation, a single one has to be computed and show cased. The resulting code should nevertheless be easily extendable so it is able to host even more data sources and metrics if necessary. To make adding to the code as simple as possible, it's going to be based on the design idea that there are two types of pipelines - data gatherer and metric generator pipelines.

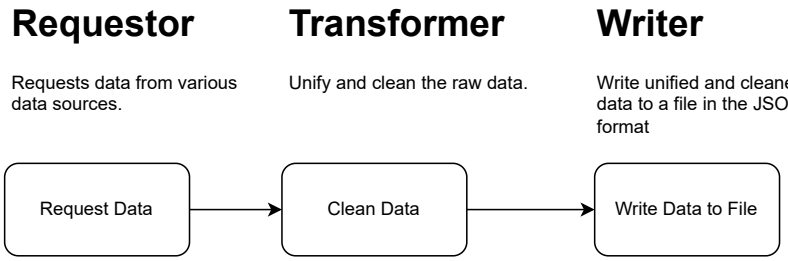


**Figure 6.1:** The Data Gatherer Pipeline

The first type of pipeline is the data gatherer pipeline. Its job is to request data from multiple determined data sources. After requesting, it should clean the individual data objects and parse them - if possible - into unified ones (e.g.

semantically merging GitLab pipelines and GitHub workflow runs). The last task it has, is to write the results of the transform step to a file.

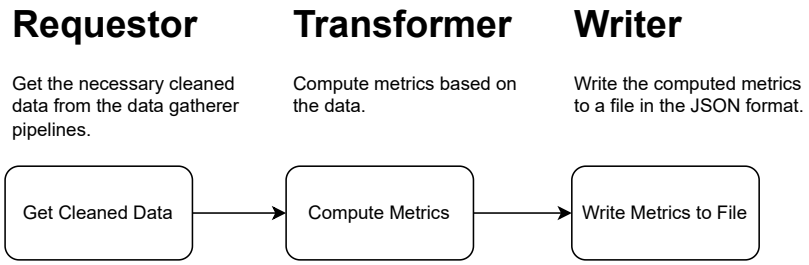


**Figure 6.2:** The Metric Generator Pipeline

The metric generator pipeline succeeds the execution of one or multiple data gatherer pipelines. Different to the requestors, it uses the already cleaned data to compute some kind of metric from it. After this, it stores the metric in a file so it can be read by another third party program that could e.g. visualize it.

Built upon these two pipeline design templates there is going to be a collection of pipelines implemented that gather data and one that computes a metric. The exact implementation details will be explained in the following sections.

## 6.2 The Data Gatherer Pipeline

As each pipeline follows the ETL concept, it is split into multiple phases which were discussed in the architecture and design sections. To best explain what has been implemented for each of these phases, they are going to be visited one by one and in doing so their logic components shall be illustrated.

There won't be too many code examples as this thesis mainly focuses on portraying the decision making process and the thoughts that went into the making of the implementation. In-detail descriptions of the inner workings of the code are, as demanded by the non-functional requirements, located in the corresponding code repository.

### 6.2.1 Requestor

The first step every pipeline has to go through is the one of data requesting. The goal is to have requestors for GitLab, GitHub and SVN data endpoints.

Prior research showed that for each GitLab and GitHub, open source Python Software Development Kit (SDK)s for their APIs already existed. These SDKs covered a lot of data endpoints, but not all of the needed ones, thus a few had to be requested from scratch. To have a common interface to both the SDK

methods and the newly implemented ones, wrapper classes were implemented for GitHub and GitLab respectively.

SVN doesn't have a REST API as the other data sources do, but can only be accessed through a SVN client that prints data to a console. Thus the SVN requestor was written from ground up and is able to parse such a log from disk.

The requestors (**E**TL) all generate output in the form of JSON objects, which are then passed to a transformer (E**T**L).

### 6.2.2 Transformer

The task of the transformer is the cleaning of data objects and unifying semantically similar data objects from different data sources.

The cleaning of data is a process of "(...) removing errors and inconsistencies from [it]" (Rahm, Do et al., 2000). Following the article of Rahm, the term "cleaning data" mainly focuses on logical errors in the data, which to discover would mean defining data constraints and issuing in-detail inspections of data objects.

After perusing the data objects the requestors return, this definition of "cleaning data" mentioning "removing errors" goes further than needed, as there are no real data errors to be fixed. Instead there are only data fields to be removed and naming inconsistencies to be resolved. Nevertheless "cleaning" will stay the term for this, even though its not meant as strictly as previously defined.

The fields to be removed are mostly Unique Resource Identifier (URI)s that link to related resources of the data objects, but are not necessary for analysis. These URIs do predominantly have keys with the suffix *_url* which enables a recursive function that traverses said data objects to be very effective. While traversing, this function identifies all *_url* fields and removes them.

```
1 def remove_keys_from_dict(d: dict[str, typing.Any], suf: str):
2     remove_keys = find_all_keys_to_be_removed(d, suf)
3
4     for k in remove_keys:
5         del d[k]
6
7     for k, v in d.items():
8         if type(v) == dict:
9             remove_keys_from_dict(v, suf)
```

**Listing 6.1:** Recursive Removal of Unwanted Data Fields

Unifying data proved to be very time consuming - both gathering the information and implementing the unification code. The main task lay in reading over the data objects from one data source and comparing them with semantically similar

ones from another. After gathering information on overlapping data fields, new unified objects had to be created from them. When a pipeline is executed, these unified objects are filled by designated parsing methods, unique for each data object of each data source.

```
class IssueEventUnificator:
  @staticmethod
  def from_github(iss_ev: dict[str, object]) -> UIssueEvent:
      remove_keys_from_dict(iss_ev, "_url")

      return UIssueEvent(
          id=to_int(getFieldFromDict("id", iss_ev)),
          ...
          source="GitHub",
          data=iss_ev
      )
```

**Listing 6.2:** Parsing Method for the Unified IssueEvent Class

The transformers (E**T**L) parsing methods return Python dataclass objects, that are passed along to a writer (ET**L**).

### 6.2.3 Writer

Coming up last in an ETL pipeline is the task of loading the transformed data. To load data in this context means saving it to any kind of storage. This could mean a database, some other service or simply the file system. In the scope of this thesis saving to the file system was sufficient as the focus lay on the requesting and cleaning of data.

As the originally requested data was provided in the JSON format, the transformed data should also be saved as JSON. Building upon the features of Python dataclasses, a generically applicable *Writer* class could be implemented for this task.

Python dataclasses by default can not be output as JSON by the Python runtime library. What the library supports however are custom JSON encoders, which can be passed to a *json.dump* function. Using a custom encoder one can specify how certain objects should be marshalled. This is were one of the dataclasses interface methods *asdict*, comes into play. It is generally available for all dataclasses and converts the contents of one into a Python dict.

A custom JSON encoder that supports the marshalling of any Python dataclass can be implemented by extending the *json.JSONEncoder* class and used as shown in the following listing.

```
class DataclassEncoder(json.JSONEncoder):
    def default(self, o):
        if dataclasses.is_dataclass(o):
            return dataclasses.asdict(o)

        return super().default(o)

    ...

    json.dump(content, file_pointer, cls=DataclassEncoder)
```

**Listing 6.3:** A Custom Dataclass JSON Encoder

Wrapping the call to *json.dump* into a *Writer* class, accompanied by a file pointer, is for now sufficient to save every data object, as all of them are either a Python dict or dataclass.

## 6.3 The Metric Generator Pipeline

The metric generator pipeline should compute a meaningful metric, based on the data the data gatherer pipelines produced. After some brainstorming and evaluation of different metric ideas, I settled upon a Continuous Integration Continuous Deployment (CICD) pipeline jobs metric (workflow run jobs on GitHub).

On GitLab and GitHub, pipeline jobs are parts of larger CICD pipelines. These can be triggered when e.g. code gets uploaded and can be used for automatic building, testing and deploying of programs. For a developer working on a coding task, it may prove tedious to wait on a CICD pipeline that takes long to finish as he may have to act on its output. Besides that, CICD pipelines might take up more resources - in the form of memory, bandwidth or CPU-time - than need be. This said, long running pipelines might waste both time (of developers) and money (of a company).

Closely analyzing pipeline jobs might prove beneficial as it may help to reveal bottle necks in a pipeline or unexpected behavior.

A pipeline job is commonly configured in the Yet Another Markup Language (YAML) format and features bash scripts and other information, like the name of the job. When a pipeline is run, job configurations get loaded and executed. The data that can be extracted post-execution is a list of job objects. The *duration* attribute of the unified job model acts as a good starting point to compute further metrics.

```
...
},
{
    "duration": 449.784755,
    "finished_at": "2023-08-24T15:41:09.819+02:00",
    "name": "go-build",
    "status": "success"
    ...
},
{
...
```

**Listing 6.4:** Unified JSON Object for a Pipeline Job

The metric generator pipeline traverses this list of job objects and aggregates the absolute *duration* for all objects with the same name, e.g. *go-build*. Along side this information, it also computes its average, min and max execution duration. On top of this, it also calculates the duration median and its 25%- and 75% percentile.

```
{
    "abs": 19347.362822000003,
    "avg": 351.7702331272728,
    "max": 990.994203,
    "median": 307.472652,
    "min": 179.316715,
    "name": "go-build",
    "q1": 277.96117300000003,
    "q3": 414.3075705
}
```

**Listing 6.5:** A Pipeline Job Metric Object

The metric generator produces such objects for all pipeline jobs with distinct names. These can be fed into a plotting engine and e.g. displayed as box plots.

## 6.4 Visualizing the Metric

As determined in the analysis section, Vega Lite was used for plotting a graph from the metrics data. Displaying the data in form of box plots was a relatively straight forward process, as the Vega Lite documentation already had an example for one.

The data for this plot was computed by a metrics generator pipeline and originally extracted from a running Inner Source project. The names of the pipeline jobs were altered in order to anonymize said project.

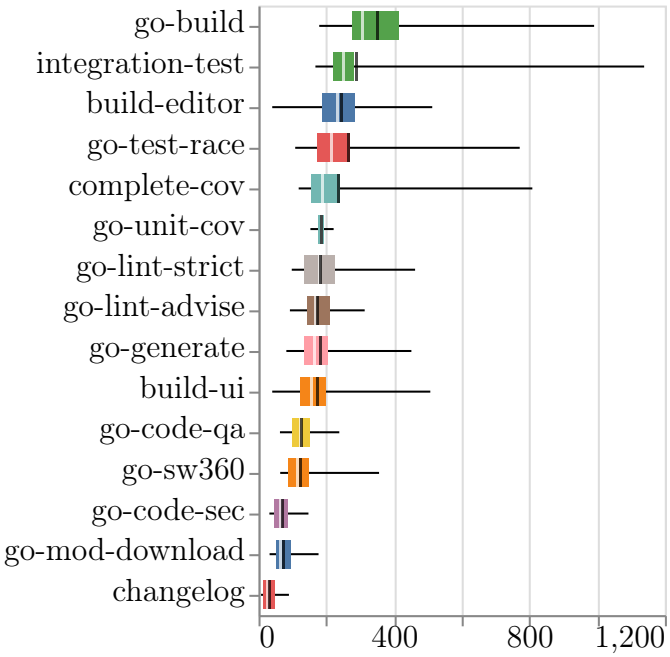


**Figure 6.3:** Box Plot of Pipeline Execution Time In Seconds

This graph shows a box plot for each pipeline job, identifiable by its name on the y-axis. The duration of each pipeline job is displayed in seconds along the x-axis. Every box plot has, additional to the 25%- and 75% percentile indicated by a colorful box, two ticks that are colored white and black. The white tick resembles the duration median, the black one the duration average. The start and end of the black rule of every box plot indicate the minimum and maximum of seconds reached.

Judging by the box plots, the *changelog* job seems to be the fastest whereas the *go-build* and *integration-test* seem to take the longest time to execute. Additional to that, the medians of both of these jobs are located on the far left of the rule, which might suggest that the execution duration of these jobs spikes from time to time.

Further taking the purpose of the individual pipeline jobs into consideration, one could also be able to argue whether or not the execution time of a job is justified or if there's room for optimizations.

# 7 Evaluation

This chapter recalls the requirements that were set earlier and revisits them, evaluating if they were met by the implementation efforts.

## 7.1 Evaluation of Functional Requirements

Concerning functional requirements, four were defined, which present the most crucial features the implementation should cover.

**Complete Data Extraction of Chosen Data Sources:** The data sources that were integrated were GitHub, Gitlab and SVN. For the two Git hosting platform it was possible to peruse an elaborate API documentation, filter out any development specific data and implement requestors for their endpoints accordingly. For SVN the most pressing task was to integrate the parsing of commits, which was also implemented. Both of the above conclusions result in this requirement being met.

**Cleaning of Data:** Data cleaning consisted of two parts. First, removing all attributes irrelevant to analysis and second, unifying any data object that semantically appeared to be served by more than one data source. The first task is implemented by a generic function that is able to remove said attributes from any JSON object. The latter one is carried out by individually designed Python dataclass objects which in turn leads this requirement to be met.

**API for Calling the Functionality:** The implemented functionality is able to be called from any third party not involved in the development process using the Python import syntax. Specific classes with a static *run* method are provided for executing the pipelines.

**Implement a New Data Metric:** A basic data metric was thought of and also implemented. This was addressed in detail in the design and implementation section of this thesis.

## 7.2 Evaluation of Non-Functional Requirements

In contrast to the functional requirements which define the specific workings of the implementation, the non-functional ones define constraints and quality concerning properties. These requirements focused on simple extension and understanding of the implementation.

**Easy Onboarding of New Developers:** The architecture follows the ETL pipeline architecture which means that every piece of code, ranging from requesting to saving data, has its concrete place. The design in which the pipelines were implemented also focuses on integrating as less abstraction as possible. Additionally there is developer documentation that explains the workings of the implementation and should make for a good understanding of the used concepts.

**Seamless Integration of Additional Data Sources:** Adding additional data sources follows a simple process, which can also be explored by taking a look at the existing pipelines. There is code in place for easy accessible data cleaning and saving which means that the main task is mostly implementing a requestor.

**Lightweight and Intuitive API Design:** The API is held simplistic and rather uncomplex. Individual types of data are accessible through static methods. These static methods are always named the same - *run* - and also mostly take the same arguments.

**Comprehensible User- and Developer Documentation:** There is documentation for both users and developers. User documentation is provided in form of a *getting started* inside the working repository. It explains what the implementation does and through an example how it can be used. The developer documentation is mainly located in a *readme file* that contains information about the design, architecture and thoughts that went into the implementations making. This should give each user and developer good starting points for further working on - and with - the project.

**Make Use of the Provided Pipeline Framework:** While designing and working on the implementation, the pipeline framework that was introduced prior to the start of this thesis was always considered and used.

# 8 Conclusions

The aim of this thesis was to implement data requestors for data sources such as GitHub and Gitlab. Based on extracted data from these sources, a metric had to be computed, expressive enough to give new insights on a software project.

For this purpose data gatherer- and a metric generator pipeline(s) were implemented following an ETL pipeline architecture. The implementation of these pipelines had to be compliant with a few functional- and non-functional requirements, which have all been met.

The way in which the different tasks of analyzing, designing and implementing were tackled, proved to work quite well. The extensive analysis that was conducted prior to the programs implementation, covered most of the open questions and lead to a mostly smooth working process.

Open issues this thesis did not cover further are first, (very) large data requests. With the current ETL architecture, the entire requested data has to be stored in working memory before it is written to disk. For large data requests it may prove invaluable to constantly write batches of said data to disk, to not overwhelm the memory.

The second issue is that the access to many APIs is authorized and regulated through access tokens. These tokens often have a request rate limit, which could prove to be a problem for large projects. Maybe multiple alternating tokens could be used for requesting data to circumvent this problem or a completely different approach may succeed in doing so.

Conclusively it was possible to integrate more development data sources to complement the efforts GrimoireLab undertook up until now. This broadened the foundation for future analysis on software projects in the Inner Source domain.

# References


Alamer, G., & Alyahya, S. (2017). Open source software hosting platforms: A collaborative perspective's review. *J. Softw.*, *12*(4), 274–291.

Apache. (2022). Kibble. Retrieved September 14, 2023, from https://kibble.apache.org/

Bostock, M., Ogievetsky, V., & Heer, J. (2011). D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*. http://idl.cs.washington.edu/papers/d3

Buse, R. P., & Zimmermann, T. (2012). Information needs for software development analytics. *2012 34th International Conference on Software Engineering (ICSE)*, 987–996.

Capraro, M., & Riehle, D. (2016). Inner source definition, benefits, and challenges. *ACM Computing Surveys (CSUR)*, *49*(4), 1–36.

Cosentino, V., Izquierdo, J. L. C., & Cabot, J. (2018). Gitana: A software project inspector. *Science of Computer Programming*, *153*, 30–33.

Crawford, L., Costello, K., Pollack, J., & Bentley, L. (2003). Managing soft change projects in the public sector. *International Journal of Project Management*, *21*(6), 443–448.

Dohmke, T. (2023). 100 million developers and counting. Retrieved August 27, 2023, from https://github.blog/2023-01-25-100-million-developers-and-counting/

Duenas, S., Cosentino, V., Gonzalez-Barahona, J. M., San Felix, A. d. C., Izquierdo-Cortazar, D., Canas-Diaz, L., & Garcia-Plaza, A. P. (2021). Grimoirelab: A toolset for software development analytics. *PeerJ Computer Science*, *7*, e601.

El-Sappagh, S. H. A., Hendawi, A. M. A., & El Bastawissy, A. H. (2011). A proposed model for data warehouse etl processes. *Journal of King Saud University-Computer and Information Sciences*, *23*(2), 91–104.

Fielding, R. T., Taylor, R. N., Erenkrantz, J. R., Gorlick, M. M., Whitehead, J., Khare, R., & Oreizy, P. (2017). Reflections on the rest architectural style and" principled design of the modern web architecture"(impact paper award). *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 4–14.

g2.com. (2023). Best version control hosting software. Retrieved August 27, 2023, from https://www.g2.com/categories/version-control-hosting

Garlan, D. (2008). Software architecture.

Glinz, M. (2007). On non-functional requirements. *15th IEEE international requirements engineering conference (RE 2007)*, 21–26.

Hartig, O., & Pérez, J. (2018). Semantics and complexity of graphql. *Proceedings of the 2018 World Wide Web Conference*, 1155–1164.

IDL. (2013). Interactive data lab. Retrieved August 7, 2023, from http://idl.cs.washington.edu/

Johnson, I. (2023). Plot and vega-lite. Retrieved August 7, 2023, from https://observablehq.com/@observablehq/plot-vega-lite

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. *29th International Conference on Software Engineering (ICSE'07)*, 344–353.

Lima, A., Rossi, L., & Musolesi, M. (2014). Coding together at scale: Github as a collaborative social network. *Proceedings of the international AAAI conference on web and social media*, *8*(1), 295–304.

Olatunji, S. O., Idrees, S. U., Al-Ghamdi, Y. S., & Al-Ghamdi, J. S. A. (2010). Mining software repositories–a comparative analysis. *International Journal of Computer Science and Network Security*, *10*(8), 161–174.

Peng, D., Cao, L., & Xu, W. (2011). Using json for data exchanging in web service applications. *Journal of Computational Information Systems*, *7*(16), 5883–5890.

Perens, B., et al. (1999). The open source definition. *Open sources: voices from the open source revolution*, *1*, 171–188.

Rahm, E., Do, H. H., et al. (2000). Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, *23*(4), 3–13.

Raj, A., Bosch, J., Olsson, H. H., & Wang, T. J. (2020). Modelling data pipelines. *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*, 13–20.

Richards, M. (2015). *Software architecture patterns*.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K., & Heer, J. (2017). Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)*. https://doi.org/10.1109/tvcg.2016.2599030

Schreiber, A., de Boer, C., & von Kurnatowski, L. (2021). Gitlab2prov-provenance of software projects hosted on {gitlab}. *13th International Workshop on Theory and Practice of Provenance (TaPP 2021)*.

Siddiqui, T., & Ahmad, A. (2018). Data mining tools and techniques for mining software repositories: A systematic review. *Big Data Analytics: Proceedings of CSI 2015*, 717–726.

Weiss, D. M., & Basili, V. R. (1985). Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Transactions on Software Engineering*, (2), 157–168.

Weller, E. F. (1993). Lessons from three years of inspection data (software development). *IEEE software*, *10*(5), 38–45.