Analysis of Git Repositories for Software Visualization

Julian Pleines

Bachelor's Thesis
March 30, 2023

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Alexander Krause-Glau, M.Sc.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Software monitoring and analysis solutions are increasingly important throughout the development process. Continuously verifying software designs and reviewing in-production code is part of keeping a software's code base maintainable. To detect potential weaknesses in the software design or overly complex source code, static program analysis is often used to detect these obstacles and maintain serviceable code. Complex source code makes it particularly difficult for the developer to clarify the facts of the program. Thus, an overview of the software in terms of complexity and interrelationships is always beneficial for program comprehension.

Since source code is often managed in Git repositories, enhancing statical analysis data with Git history metrics aids in the detection of design flaws by revealing frequently accessed and modified files.

This thesis shows the implementation of the first stage to provide ExplorViz with robust, extendable, and easy-to-set-up statical analysis capabilities, which are able to calculate and factor in Git metrics. The analysis service is built to be easily integrable into continuous integration pipelines to provide up-to-date project overviews triggered by Git actions. We will show how such a static analysis is designed and implemented, how Git repositories are handled, and how to design the service with extendability in mind. Additionally, we demonstrate how this service can analyze arbitrary Java projects in a locally running Docker container or integrated into continuous integration based on GitLab. As an accompanying visualization for the generated data is missing at the time of this thesis, the usefulness of the generated data will be examined in upcoming works.

Contents

Chapter 1

Introduction

1.1 Motivation

Git is one of the common version control systems and, therefore, widely used. Its structure of continuously changing source code with the ability to retrace past changes can show the insides of the development process such as the source code's evolution.

Developers spend nearly 50 to 60% of the time for maintenance tasks to comprehend the programs, showing the importance of better tools to help developers to fulfill this task more effectively [20].

As the historical versioning provided by Git can give insights into the evolution of software parts, structural data extraction must be done from the source code directly. Other static information and metrics, such as the complexity of classes, can also be analyzed or retrieved from the source code. To be as beneficial for the user as possible, we have to find meaningful, worthwhile information to be extracted.

ExplorViz is a live trace visualization tool that researches how and if software visualization can facilitate program comprehension. ExplorViz's focus as a runtime behavior analysis tool is particularly suitable to monitor large software landscapes and tends to concentrate on the visualization of dynamic data. Extending it by integrating a static analysis tool would improve its effectiveness, enabling a fast peek into the project's code to clarify some architectural problems.

Even though there may exist more static analysis tools than one would think, these tools often only get used if they are easy to integrate. Developers tend to be more interested in the source code than calculated metrics due to the inconvenient access and generation of static analysis data. Thus, providing the analysis as an easy-to-integrate solution without much configuration would provide this valuable data to the developer hassle-free. Regarding the listed limitations of other approaches, in this thesis, we develop an easy-to-integrate, CI-ready static analysis tool for Java projects and its Git repositories to serve data for a subsequent visualization.

1.2 Document Structure

Starting with Chapter 2, the goals to be achieved are listed and explained. Chapter 3 gives an overview of foundations and technologies used in the thesis and Chapter 4 discusses the project design and underlying principles used, followed by Chapter 5 where the actual implementation of the project is explained. After evaluating sample applications in Chapter 6, we will look at similar projects and finish with the conclusion and an outlook on possible future work.

Chapter 2

Goals

This thesis aims to implement a source code analysis service. It will analyze Java code from Git repositories, collect structural data and code metrics, and provides a remote data endpoint where the analysis data will be send to.

2.1 G1: Valuable Data for Program Comprehension

Large programs with many files and a deep folder structure can easily impede the comprehension task. Thus, maintaining and extending can be hurdled by the time a software engineer or developer needs to find the involved parts of the system. Metrics and data obtained from the source code could help understand interclass communication. In contrast, historical data from Git-commits, on the other hand, can hint at packages under heavy development. The goal is finding meaningful data from within the source code and Git and preparing the data to be visualizable later on.

2.2 G2: Source Code Analysis

The implementation of the source code analysis is the next step. Each source file within a Java project will be analyzed for data and metrics chosen in G1. The goal is to implement a fundamental static source code analysis that can run on any project given. Special attention is paid to the correct resolution of types due to the immense contextual significance within programs.

2.3 G3: Repository Handling

As many projects use Git as their source code management system and the managed repositories tend to grow in complexity and size, the needed computational effort to analyze the software entirely for every commit would increase similarly. A *commit-aware* analysis approach is the solution to decrease the computation time. Only changes from one commit to its successor will be included in the analysis process. The updated implementation of G2 fulfills G3 if a new commit gets added to the monitored repository, and only the new commit needs to be analyzed to get a full representation.

2.4 G4: Networking

As for now, the generated data is still only available locally and needs to be sent to an remote endpoint. The endpoint works as a simple gateway that could be added to ExplorViz. As the case may be, the analysis could not run for some commits. Thus, analysis data is missing. Once the analysis gets invoked, it will request the state of the endpoint's data, meaning the id of the latest transmitted commit, and can send the missing data, analyzed as a batch job, up to the latest commit.

2.5 G5: Docker Container for CI/CD Integration

The last development step is to finalize the analysis tool by migrating it to a self-contained Docker image that can be integrated into a CI/CD pipeline. The analyzer will be invoked by new commits in the Git repository, thus making sure the visualization always has the latest data available. As the analysis is stateless, the needed Docker container is lightweight and fully disposable after a run, resulting in easier migration and less occupied hardware resources.

2.6 G6: Evaluation

Lastly, we need to demonstrate the service's executability in a concrete application, verify its robustness for larger projects, and confirm the ease of integration. We will assess the operability of the container approach and the generated data's usefulness.

Chapter 3

Foundations and Technologies

3.1 Program Comprehension

Program comprehension is a research field of the cognitive process of understanding software, or better, the source code. As software is developed, maintained, and extended, the comprehensibility of the code itself must constantly be monitored, as incomprehensible code is not serviceable [16].

Two classical theories were formulated:

Top-Down The top-down approach heavily benefits from the developer's knowledge within the software domain. While a developer tries to comprehend a program, some hypotheses of the functionings are made. These are often high-level assumptions that get approved or rejected by looking deeper into the code. This procedure of looking deeper proceeds until the ground level, and therefore the complete program, is understood.

Bottom-Up The developer tries to comprehend the program in chunks from the ground up. As it is impossible to draw on knowledge, the ground level needs to be comprehended first. Slowly working upwards and forming abstract higher-level tasks, until the whole software is comprehended.

As most developers have some knowledge from other projects, they often use both strategies simultaneously to comprehend [3].

To help in the overall program comprehension, especially in the higher level abstractions, visual aids in the form of visualizations can ease and facilitate the workload developers do to comprehend the software. As the data from the analysis should help the developer comprehend the software more efficiently, we must first understand what data facilitates comprehensibility.

3.2 Static Analysis

While the analysis of *running* code is called dynamic analysis, the complement,i.e. analysis of the software without running it, is referred to as static analysis. Static analysis can be applied directly to the source code, but even compiled binaries can be analyzed to find flaws after the compilation [8]. Call graphs can be created when applied to the source code, giving analysts or developers an overview of what functions might be invoked and what dependencies exist [6].

As nearly any file can be analyzed statically, in the context of the thesis, only the analysis of Java source code is worth considering. Java-typical tools for static analysis include IDEs (e.g., Eclipse, IntelliJ IDEA) with its functions to aid while developing (e.g., spell checking, providing code suggestions) and external tools like *Checkstyle* to monitor the source code quality. Since Git repositories are analyzed in particular, historical events in code modification besides the source code can be included as material to analyze. Based on the thesis' context, we will analyze the repository's data statically.

3.3 Continuous Integration

CI describes the automation of software projects' build and validation processes. As its goal is to detect build errors early in the development process, it is well-adopted within the industry and speeding up the development process [18, 5]. CD, short for Continuous Delivery, is the next stage of automation, focusing on creating reliable software versions with a high frequency. A sophisticated CI/CD pipeline can increase development speed and productivity. Unfortunately, a poorly designed CI/CD pipeline can do more harm than good [21].

Tools should provide information when needed the most to support the developer during the development. The information is just a click away by preparing the data automatically in advance (in our case, the moment the data is on the Git repository).

GitLab

GitLab[1] is a leading source code hosting facility featuring version control, via Git-scm[2], and DevOps. Since its initial release in 2011, it positioned itself as an open-source alternative to its competitor GitHub by providing the MIT-licensed *GitLab Community Edition (CE)*. GitLab can be used self-hosted or via the Software as a service solution offered at *gitlab.com*. It offers a feature-rich CI/CD pipeline ecosystem with built-in and easy-to-configure runners.

Created jobs within the CI/CD pipeline can use pre-build Docker containers (see Section 3.8), thus adding validation, verification, or analysis functionalities as post-build stages to monitor the current quality of the project.

The *basic* package of GitLab's SaaS solution is free of charge, including the use of the integrated CI/CD pipeline and its runners, making it ideal for academic, open-source, or private projects that require sophisticated pipeline solutions featuring quality assurance functionalities.

[1]https://gitlab.com/
[2]https://git-scm.com/

We use GitLab as our CI runner and tailor the container specifically for it to provide easy integration.

3.4 JGit

Eclipse JGit[3] is an open-source implementation of the Git version control system as a Java library, allowing users to walk through repositories programmatically. Even if JGit provides a small CLI similar to but not as feature-rich as the standard Git CLI, it is not intended to replace it. Some Git porcelain-style commands are available within the API to easily access high-level features, callable directly without needing a CLI. However, the main focus lies in managing internal Git object identifiers. To do this, a so-called RevWalk is used to walk along the tree structure of Git commits [13].

We will use JGit to analyze the local Git repository and check out different branches to retrieve the needed data from the remote repository.

3.5 JavaParser

JavaParser[4] is a library written in Java, allowing developers to interact with Java code as a Java object representation. The Java code is brought to Abstract Syntax Tree (AST) representation. An AST is built out of different nodes arranged hierarchically, each of them containing information about its structural properties. For example, some properties attached to the node representing Java's main methods are *PublicAccessSpecifier*, *MethodDefiniton*, and *VoidDataType*, giving a comprehensive view of every building block of the program analyzed. JavaParser is not only meant for analyzing the source code itself; it provides convenient ways to manipulate it. While complex ASTs can be hard to comprehend, JavaParser provides mechanisms allowing users to search and modify the AST more intuitively. Therefore the user can be more focused on identifying patterns of interest [19]. Extending the JavaParser, the project also habits JavaSymbolSolver, a package that detects and retrieves type information from the AST. It features functionalities to detect Java-bundled types as well as project-specific ones.

JavaParser and its symbol solver are the libraries used to perform the static analysis and type resolving.

3.6 gRPC

Google developed gRPC[5] as an open-source cross-platform Remote Procedure Call (RPC) framework to communicate between different distributed applications. gRPC supports uni-

[3]https://www.eclipse.org/jgit/
[4]https://javaparser.org/
[5]https://grpc.io/

and bidirectional, blocking, and non-blocking communication between client and server and data streaming. It uses HTTP/2 as the transport medium, and the data is, contrary to HTTP commonly using JSON, more strictly specified. gRPC uses Protocol Buffers (see Section 3.7) as the chosen data serializing framework [12].

Figure 3.1 shows the working principle of gRPC. It should be noticed that the service and clients are all written in different languages. By providing our analysis tool as a gRPC client, the gRPC server can be reimplemented in any other language to be able to integrate the analysis into projects using other languages than Java.

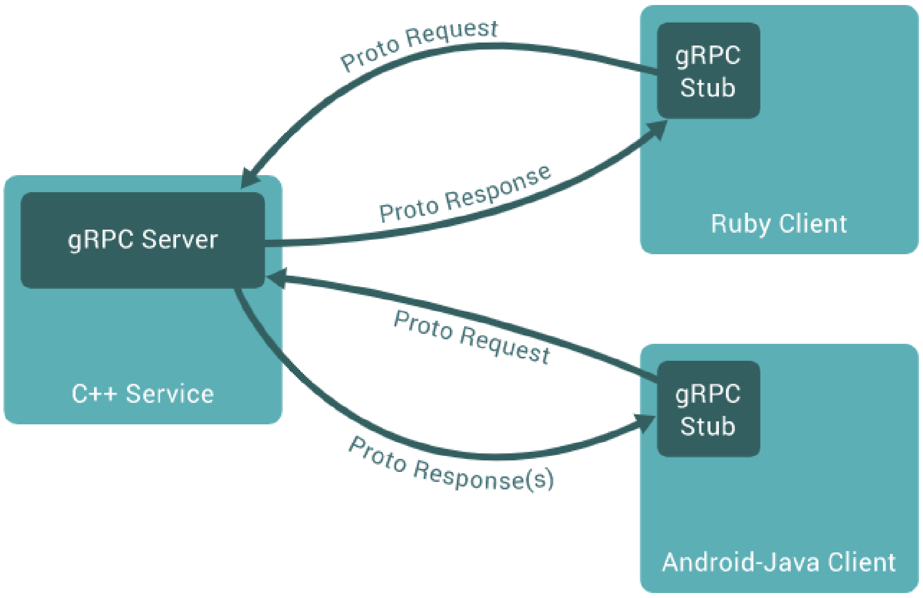


Figure 3.1. gRPC communication overview (Source: grpc[6])

3.7 Protocol Buffers

Protocol Buffers[7], abbreviated as *protobuf*, is a data format and bundled serialization toolchain. Protobuf's *.proto* files define the contained data structure, similar to fields in Java classes, where primitive types belong to a *message* object. The *protoc* - protobuf's compiler and code generator - creates all needed methods to handle the data access, thus making getting and setting of values straightforward and fully deterministic independent of the

[6]https://grpc.io/docs/what-is-grpc/introduction/
[7]https://protobuf.dev/

used language or system. Protobuf messages are both forwards and backward-compatible, small in size due to binary representation, but lack the self-describing characteristic found in file formats like JSON.

Protobuf is used as the file format for the gRPC communication but also for the definition of the internally used data format. Shown in Listing 3.1 is an exerpt of the message definition of the *MethodData*, which is directly used as the data holding object. Protobuf supports nested message definitions as seen in line 4, where the *ParameterData* message is used as a type.

```
1  message  MethodData {
2    string returnType = 1;
3    repeated string modifier = 2;
4    repeated ParameterData parameter = 3;
5    repeated string outgoingMethodCalls = 4;
6    bool isConstructor = 5;
7    map<string, string> metric = 6;
8  }
9
10 message ParameterData {
11   string name = 1;
12   string type = 2;
13   repeated string modifier = 3;
14 }
```

Listing 3.1. Excerpt from a nested protobuf message definition taken from the *fileDataEvent.proto* used in this project.

3.8 Docker

Docker[8] is a software platform that allows developers to package and deploy applications and their dependencies in isolated environments. *Docker containers* are lightweight, portable, and self-contained environments that can run independently of hardware and operating system restrictions. Developers can create containers that contain all the necessary components and then deploy these containers on any machine with Docker. This makes it easier to develop and deploy applications across different environments. It also allows developers to share and distribute their containers through public or private repositories, making it easy to provide functionalities already set up and ready to use.

[8]https://www.docker.com/

3.9 ExplorViz

ExplorViz[9] is a runtime behavior analysis tool for Java applications and can analyze large software landscapes that are more frequent in the industry nowadays as software systems grow. By providing insights into the software, such as the communication between programs and control flow in applications, it is meant to help software engineers to comprehend large-scale software systems better. Architectural problems are easier to address when the root cause is known. By providing a tailored visualization for said information, even in a collaborative context, enabling multiple clients to observe the same visualized software landscape, ExplorViz aids in faster software enhancement [9]. Even scalability is assured by design [10, 14].

As a breif overview, Figure 3.2 shows the architecture of ExplorViz. Our Service will fit in the *Monitoring* environment on the left, responsible for data collection, and provide a service similar to the *Adapter Service* seen in the *Analysis* environment. At a later stage, the *Analysis* and *Visualization* will be extended to accept the newly collected structural data.

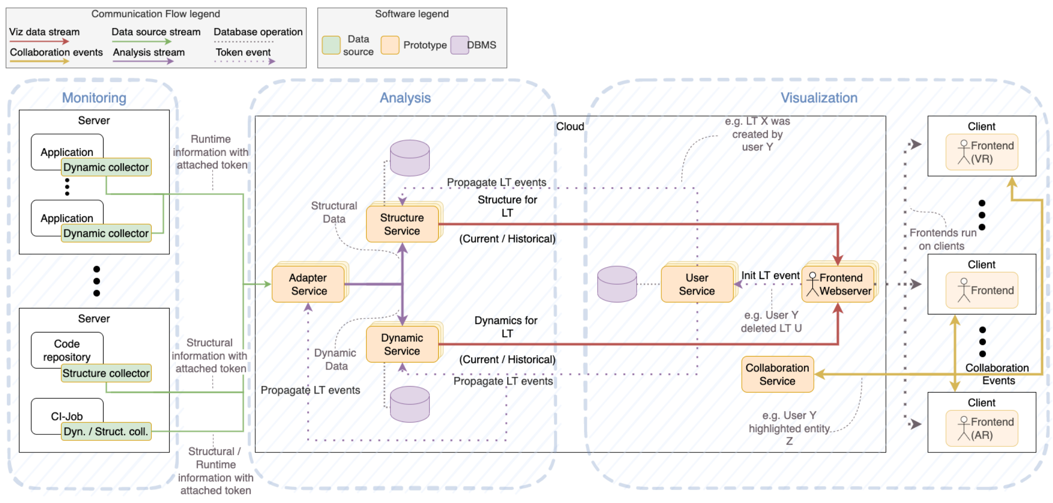


Figure 3.2. ExplorViz's conceptual design.

Shown in Figure 3.3 is a visualization of a runtime behavior in ExplorViz. Even though the data visualized is dynamic data, the visualization should give a good starting point of what data we could need as we want the structural data be visualized similar. As the user of ExplorViz is able to view specific dynamic data *snapshots*, the user interface on the bottom could be used to select the Git commit for structural data, if implemented that way.

[9]https://explorviz.dev/

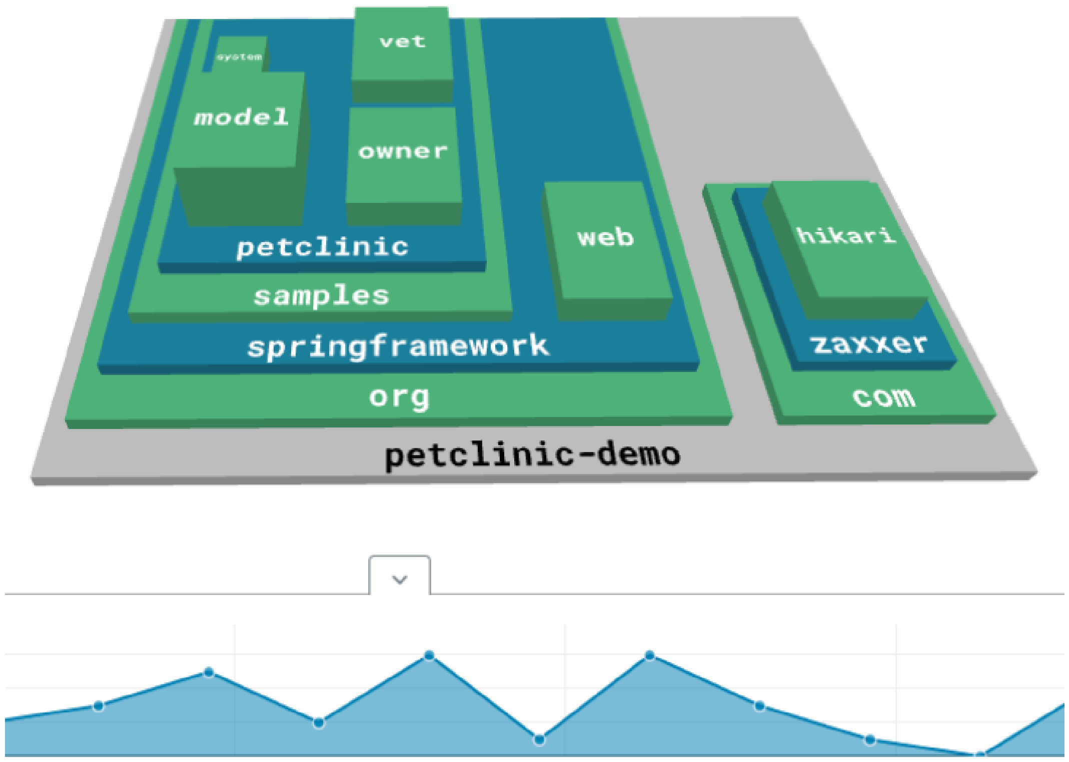


Figure 3.3. An example of runtime data visualized by ExplorViz; used as a reference as later structural data should be visualized similar.

3.10 Quarkus

As large-scale containerized software deployment increases in popularity, Quarkus[10] offers simple deployment of Java applications on Kubernetes. The framework follows a *container first* design, optimized for low memory usage and fast startup times, particularly tailored for reactive distributed applications.

Quarkus is used as the framework for the analysis service we have implemented.

[10]https://quarkus.io/

Chapter 4

Approach

This chapter discusses how a static source code analysis for Java code is done and what data we can collect. Further, we will look into handling Java projects managed in Git repositories, how the analysis can be designed as a service, and shed some light on potential limitations and why it is designed as a stateless, pre-built Docker container. As developing an ExplorViz extension to visualize the generated analysis data is not part of this thesis, we will design against a virtual, to-be-implemented version, similar to the current dynamic data visualization (seen in Figure 3.3).

4.1 Source Code Analysis

In software development, the written source code is only used to express and write down the intended actions the computer has to perform. Thus, to understand and analyze the software, the exact representation of the source code itself is unnecessary and makes the latter more complex. To analyze program code efficiently, the conversion to an *abstract syntax tree* (abbreviated AST) prior to analysis is often used to simplify the following static analysis. Compilers often use an AST representation to check for errors before generating executable code, as the AST is created based on a syntax analysis performed on the source code. The AST representation of program code allows a partly syntax-independent representation where some simplifications to the syntax can be made, making later analysis' of the AST easier as the focus is on semantic meaning and not syntactical correctness. A generated AST contains all the same semantic information as the source code it is based on. Furthermore, a developer is mostly not interested in the exact program representation. It is only used as an information transportation medium, such as every natural language. Thus, we will focus our analysis on the AST representation of the program, so we do not have to deal with different syntactical representations with the same semantics.

As generating and analyzing ASTs heavily depends on the languages and parsers used, due to ExplorViz's focus on Java, we will only discuss problems and solutions for the Java programming language. Even though some concepts may be transferable to similar languages, other language-specific problems may exist.

Several Java projects to generate ASTs from Java code exist, but we focus on the JavaParser project as this will enable us to use the JavaSymbolSolver for type resolving afterward. The reason for the need for the JavaSymbolSolver will be explained in Section 4.1.2.

A motivation to perform a *source code analysis* in the context of program comprehension is to provide insights about a program without looking at the source code. The analysis will collect information about the program that should help get an overview of some software building blocks. Once the analysis data is visualized to the developer, it can help them find their way into the program more quickly.

4.1.1 Structure Data

Now that we looked into the reasons and advantages for source code analysis and AST generation, we can turn to the types of data we can collect. As the collected data dictates the meaningfulness of the visualization later in ExplorViz, we have to be careful about what data to keep and what to omit. The most trivial approach would be to keep all data we get from the generated AST, as this data resembles the whole Java file. However, this would defeat the purpose of this analysis service to pre-structure and preselect data for the following visualization.

File Data

Representing class dependencies is only possible if all used classes are listed and available to the visualization. The language conveniently provides this as Java requires specifying all used classes, static methods, and enums in the file as imports. Thus, the file data part of the analysis consists of a list of imports, the package the file is in, and a list of all classes contained in this file.

Class Data

The main content of files consists of class declarations. We must remember that Java has four class-like object types: Classes, Abstract-Classes, Interfaces, and Enums. We can handle these in almost the same way as each consisting of a body potentially containing methods and fields (in Enums, the fields are called *enum constants*).

Classes can have modifiers such as *private, public* or *protected*. modifiers are important to be present in the visualization as private and protected keywords mark almost always only classes essential to look into if the internal workings of the class needs to be considered. Each class can be addressed by its fully qualified name (abbreviated FQN), consisting of the concatenated package and class name. This is easy to determine for *top-level classes*, but nested classes require a hierarchical FQN containing the parent's FQN combined with its name. Saving the FQN will enable the visualization to assign each type a corresponding class. Class data holds, therefore, a list of contained methods, fields, access modifiers, inner classes, implemented interfaces, and the superclass.

Method Data

The program's logic is contained chiefly within methods, and so is the complexity. To understand the task a method fulfills, its name and signature are the best sources of information. The signature contains the types of the parameters, which we will look at in more detail in Section 4.1.2. The name should be descriptive enough to hint at the task. Constructors are just a *special form* of methods, so they can be handled similarly. Besides the name and parameters, the return type and modifiers are added to the method's data, even though the data entries are mostly a container to add specially calculated metrics. These metrics can summarize some characteristics of the method's inner workings. We will discuss the code metrics in Section 4.1.3.

Data Model

We designed the data model shown in Figure 4.1 based on the above needs. The model has been expanded by `FieldData` and `ParameterData` to keep them out of the `ClassData` and `MethodData` as they hold multiple entries that specify only themselves, not the parent `Class` or `Method`. The model is designed to be *class-flat*. Thus, all classes are stored at the same level. This allows us to easily list and loop over all available classes, regardless of their hierarchy depth. The hierarchically accurate position is represented by its FQN.

4.1.2 Type Resolving

As outlined above, the type resolution is an essential part of the data collection. Knowing a method has parameters of an arbitrary name does the programmer no good in comprehending the fulfilling tasks of the observed portion of the program. The programmer needs some information about the used type itself. We can split Java types into five different resolve categories, which all have different complexities to resolve them: *primitives*, *built-ins*, *package-types*, *project-types*, and *external-types*.

Primitives

Primitives are the most basic types in Java. These are integrated into the language and are *no objects*. Primitives are the most fundamental variable types every Java programmer knows. Thus, we do not need to process these further to help understand these types. Primitives consist of `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`. Besides these eight primitive types, the advanced data structure *array*, denoted by a pair of brackets following the type, can be combined with any data type. Any Java programmer should recognize arrays and primitives; hence we will not change the representation of the type.

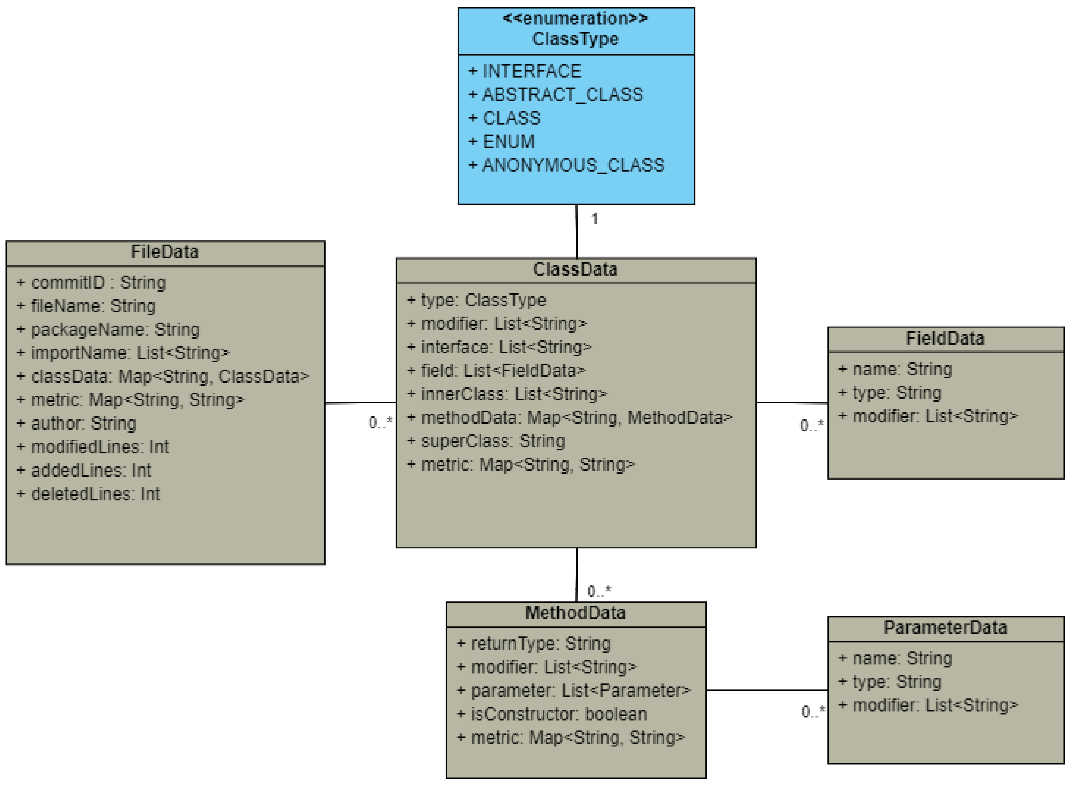


Figure 4.1. The structural data model with metric fields.

Built-Ins

Built-In data types are often used in many projects and are part of the Java language. These types are defined in the `java.lang` package and get implicitly included when used. Thus no import statement is needed. Well-known built-ins are `String`, `Class`, and `Object`, the objectified primitive type wrapper such as `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float`, and `Double`, further the interfaces `Throwable`, `Iterable`, and `Comparable` as well as the annotations `Deprecated` and `Override`. The package also contains definitions of errors (`StackOverflowError`) and exceptions (`NullPointerException`). Stated above are only some of the most used or known types contained in the package.

As some of these types may not be recognized by the programmer, to make it unambiguous, we will append the package name to the built-in types as well, thus representing the type `ThreadGroup` with its fully qualified name `java.lang.ThreadGroup`.

Package-Types

A significant design decision of the Java language is the `package` concept. Within the same package, *package-private* classes can be accessed by classes, methods, or fields from other Files. All of this is possible without explicitly importing the classes directly. A Java file can contain types that are neither *primitives* nor *built-ins* but are not imported either. To resolve types of this kind, we have to look into every file within the current package, as we have no further information which file contains the type definition. While both of the categories above could be handled by a simple look-up, resolving types of this category requires a more advanced technique. Thus we will use the *JavaSymbolSolver*, as hinted above, and its integrated *JavaParserSymbolSover*. The type solver provides us with the needed type FQN, so we know where the type is coming from.

Project-Types

Project-Types are types defined within the project but not the same package. These types are defined somewhere in the available source code; therefore, we can collect data from them. Since the Java compiler also needs to find the types, the fully qualified names are found in the list of imports. If we need more data about these types, we can simply search the path given by the FQN to find the corresponding source code. Sometimes types may be relative to imports, only taking the import as a starting point, but the FQNs are easily determined as the last import identifier must match the first of the type.

External-Types

These are the vaguest types. As a project becomes more extensive and depends on libraries, external libraries in the form of *jars* are used. These jars usually are unavailable within the project's repository but are loaded by the build tool on demand. While some projects depend on explicit versions of libraries, others just use the current latest version of a library. This can lead to code-breaking changes when the latest library version is used together with an old project state. As we only do a static analysis and do not have to build the project, this will not necessarily break the analysis but may produce some unresolvable types. Also, this would demand that we have to get every library used. This may not seem like a significant performance impact, but this has to be done for every Git commit while finding the matching library version that should have been used in the building process. This overhead can proliferate by the number of libraries used and commits to being analyzed. Omitting the inclusion of external libraries in the type resolving to a degree where we do not need to look into the jar itself should not be very limited in the sense of comprehension as the developer should be more interested in the analyzed project's code as in the library's code. Thus, we only provide the FQN for the type without considering the definition. As external library types are always stated in the *imports* of a file, we only need to match these import names to the found type name. This comes with all the drawbacks similar to undefined types but compromises performance, simplicity, and completeness.

Limitations

While almost all types should be resolved by now, we need to define what happens when the resolution fails. We can not simply omit the type, as that would result in analysis data that is missing data. As we have no fallback for this case to get the resolved type, we can only add the type name from the source code we have found but could not resolve and add that type name as the data entry.

Similar problems occur if wildcard imports from libraries are used. We could safely assume that if a single wildcard import is present, the type in question is part of the package imported by the wildcard if *every other type resolving failed* and the type resolver all worked correctly. Then the type in question *must* be defined within the wildcard's package. This always fails if more than one wildcard import is present.

4.1.3 Metrics

Even though the above-discussed basic structural data and attached types are essential to the overall comprehension of the project's interactions, it is often wanted to get a sense of the code's *smell*. Be it to estimate the maintainability of specific project segments or to appraise the risk of errors due to high complexity. We will now look at some metrics that we will implement later. Some metrics are more valuable than others in complexity estimation, but they all add new information to the analysis data.

Lines of Code

As the name suggests, this metric is calculated based on the lines of the present code. Only lines of source code are counted, and line-, block-, and JavaDoc comments are omitted. Thus the overall lines of code are always less than or equal to the lines in a source file. The lines of code get calculated for the entire file, containing classes and methods. This lets us quickly see if a class or file is potentially prone to errors. Even though no clear values are defined to differentiate between *good* and *too large* classes and files, it can give us a rough idea. A file containing 10,000 lines of code is objectively worse than a file with 100 lines. On the other hand, a file containing 1000 lines of code equally distributed over some methods seems big but might be better than a file consisting of 500 lines, all contained within a single method.

Nested Block Depth

Program building blocks are not only simple statements; a large portion of program code consists of control structures. While the complexity of a section of program code, consisting of simple statements only, is determined by the number of statements, the complexity of program code containing control structures can be harder to understand for the same number of lines. The *nested block depth* metric searches for the deepest nested statement and returns its depth. This metric aims to show the increased context awareness developers need. Every control structure creates a new context for executing the following statements. The developer has to correctly remember all applicable requirements for the current context, thus increasing the cognitive load relative to the nested depth. A high *nested block depth* value can hint at a complex method that could benefit from outsourcing some code.

Cyclomatic Complexity

The cyclomatic complexity metric[1] is another complexity measure for control structures. Unlike the *nested block depth* metric, the cyclomatic complexity sums up all control structure statements. Thus, the complexity increases with every control structure used. Nested structures are not counted differently than top-level structures, making this metric more valuable along with the *nested block depth*. A low nested block depth and a high cyclomatic complexity value are potentially more straightforward to comprehend than the other way around. Another known drawback in calculating this metric is the valuation of *swtich-case enties* as every entry in the switch block will increase the complexity by one. The problem can be seen in Listing 4.1 where a straightforward `swtich-case` is presented. The resulting value is 14 indicating a higher complexity than the doubly nested, labeled `for` loop with a low value of 5 (seen in Listing 4.2).

```
switch (month) {
    case 0: return "January";
    case 1: return "February";
    case 2: return "March";
    case 3: return "April";
    case 4: return "May";
    case 5: return "June";
    case 6: return "July";
    case 7: return "August";
    case 8: return "September";
    case 9: return "October";
    case 10: return "November";
    case 11: return "December";
    default: return "NONE";
}
```

Listing 4.1. Complexity value of 14

```
int sum = 0;
OUTER: for (int i = 0; i < limit; ++i) {
   if (i <= 2) {
      continue;
   }
   for (int j = 2; j < i; ++j) {
      if (i % j == 0) {
         continue OUTER;
         }
   }
   sum += i;
}
return sum;
```

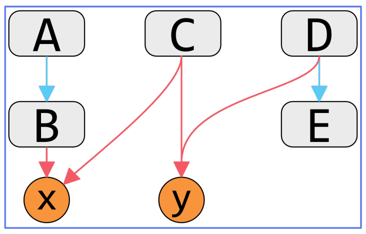
Listing 4.2. Complexity value of 5

Lack of Cohesion in Methods

The last metric we want to look into is the *lack of cohesion in methods* metric, precisely the variant number 4, abbreviated LCOM4. The findings based on this metric indicate if a class

[1]also McCabe metric after its inventor Thomas J. McCabe

```java
class A {
    private int x;
    private int y;
    void methodA(){ methodB(); }
    void methodB(){ x++; }
    void methodC(){ y++; x++; }
    void methodD(){ y++; methodE();}
    void methodE(){}
}
```



```java
class B {
    private int x;
    private int y;
    void methodA(){ methodB(); }
    void methodB(){ x++; }
    void methodC(){ y++; }
    void methodD(){ y++; methodE();}
    void methodE(){}
}
```

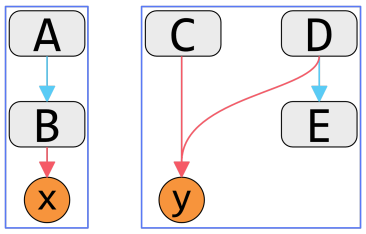


Figure 4.2. Two similar classes with their respective LCOM graphs. The left class has LCOM=1, a *good* class. The right has LCOM=2 and can be divided into two classes. One new class contains `methodA`, `methodB`, and `x`, the other class contains `methodC`, `methodD`, `methodE`, and `y`.

can be split into multiple classes due to *lack of cohesion*. In the sense of the LCOM4 metric, two methods are related if one of the methods calls the other or both access the same field. We analyze all methods within a class and search for not connected methods. A *good* class should result in an LCOM4 value of 1, meaning all contained methods are related. A value of 0 only happens if the class does not have any methods at all. This can be considered a *bad* class, but it depends on the job of the class. We will revisit this later in Section 5.5.4 as the behavior is also implementation dependent. Furthermore, splittable classes result in an LCOM4 value of 2 and above, meaning these classes can be outsourced to their class as they are not cohesive. Shown in Figure 4.2 are two nearly identical class implementations.

4.2 Git Repository Management

Now that we have clarified how to analyze a single state of a Java project, including all its files, we have to think about handling Java project *snapshots*. These are Java projects at a state corresponding to a Git commit. Analyzing every file for every commit is needed to see the changes throughout the development period. The data must be structured so that the visualization can efficiently and quickly retrieve it later.

4.2.1 Handling Changes

Firstly, we must specify what a *change* means in our context. We can define a *change* as any file modification that changed its content from the previous commit to the current. If a file has not changed, the file does not need to be analyzed again. Since this definition broadly matches the principle Git uses, we must handle any file Git marks as *modified*.

As our analysis' smallest data block is the method, we could check if a method has changed. If a change is detected, only the analyzed method's data chunk needs to be saved. This sounds good in theory but has significant drawbacks. Some metrics defined above, namely lines of code and LCOM4, are calculated over the entire class or file. Only updating the method data would result in false values for those metrics and any metric that may be added in the future featuring the same class or file-wide scope. Thus, we have to update these metrics as well.

Additionally, we had to send an *update* analysis data block containing only updates to the last complete analysis data. This would ultimately lead to an increased computational effort. Imagine a file present for 1000 commits, changed every commit in only one method. This would require the visualization to accumulate all 999 updates and add them to the complete initial analysis to get its current state. While this would be a good idea regarding file size, the increased complexity for the data aggregation done by the visualization cancels the advantage. Aside from that, the analysis has to be performed on the entire file due to the mentioned file-wide metrics, keeping the execution time nearly identical with only a slight decrease in storage space but an increase in aggregation complexity. A solution would be to send the entire analysis data packet if the file was changed during the commit. The resulting data packet would contain all valid data for the analyzed commit. The visualization can therefore be more straightforward as the newest analysis packet will contain all data for a file. Further, if a file has no changes, we can ignore it as the *old* state is still valid.

Regarding changes, Git has different *change types* for files. A file's *content* can be *changed*, the file can be *renamed*, *deleted*, or *copied*. We could preserve these differences in changes, but we are only interested in the data within the file. If a file is no longer used in the project, we are not interested in the fact that it is deleted. If a file is renamed, we are not interested in the old name as the new name counts and is used for analysis. Same for copying. We can briefly break down file modifications to the *changed* type. This simplification combined with the above-mentioned *only entire file* strategy creates one problem: We cannot find out which files are part of the commit as we do not know if a file has been deleted or was merely not modified in this commit. To solve this problem, we introduce a *commit report* that contains a list of all files that belong to the current commit. Accumulating any state is as simple as getting its matching commit report and searching for all the newest analysis data packets for each listed file.

4.2.2 Handling Branches

As Git supports branches, the analysis also needs a way to support this feature. Maybe the user wants to analyze the *main* and *develop* branches at the same time to compare them. In practice, Git branches are only ordered lists of connected Git commits; thus, handling branches is almost trivial. Figure 4.3 shows a basic Git project containing two branches; the node labels symbolize the commit's hash identifier values. As we can see, the branches share commits 1 and 2, and the analysis data is the same. Due to design restrictions (see Section 4.3), we will send the identical analysis data packets twice for commit 1 and 2, once for the *main* branch, and once for the *develop* branch. This can be solved by a lookup on data insertion into the database, as the combination of filename and commit identifier is a valid unique key and must only be inserted once. We generate further analysis data for commits 3 to 6 only once.

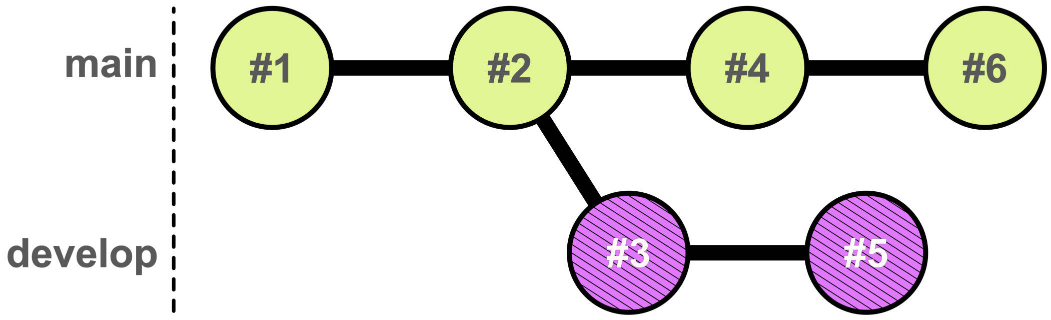


Figure 4.3. A repository commit-graph containing two branches. *develop* branched off from *main* at commit *2*. Both branches contain the commits *1* and *2*.

4.2.3 Git Metrics

Analysis data of source code and its metrics is valuable but static. One main advantage of analyzing the project's repository is to be able to track changes over time. Access to the raw Git data allows the collection of some information about the kind and frequency of file modifications.

Modifications/Code Churn

Our policy of sending new analysis data only on changes would allow the visualization to calculate the frequency of modifications but not the quantity. The raw access to the Git data enables the collection of more specific information. Adding the number of changed, added, and deleted lines makes it possible to give a qualitative assessment of the commit. If a file features many changes during the development, it could be a sign of a potential *god-file* the whole project strictly depends on. Also, as we can compute the *delta* of code changes at the visualization state later on, we can calculate the *code churn* metric [7].

Authors

By coupling the author's name to each analysis data object, it is possible to identify files accessed and modified by many developers. These data can be retrieved by getting all files with the same name and listing the authors in a set. If many different authors modify a file, it can hint at a file without a specific purpose (e.g., *utility classes*) or unclear areas of responsibility. These unclearly defined workflows are not detectable by a standard static code analysis. However, this metric is more suited to human resource or project management, which become increasingly important with a growing software project and increased code base.

4.3 Service Design

Figure 4.4 shows the design of the *analysis service* consisting of the *code-service*, serving the task of beeing a data endpoint inside another project's ecosystem, and the *code-agent*, the source code analysis runner and Git repository handler. Recalling the ExplorViz architecture shown in Figure 3.2, the *code-agent* serves as *Structure Collector* in the *Monitoring* environment by providing the analysis data. The *code-service* has a similar job than the *Adapter Service*; acting as a gateway to forward data towards the database.

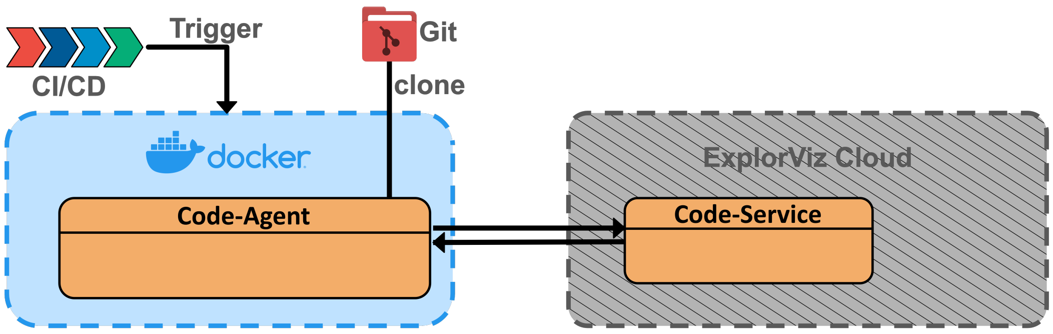


Figure 4.4. Overview of the service design. The ExplorViz cloud is used exemplary and can be exchanged with any project.

As we want to implement the *code-agent* stateless, we need to consider if we want to tolerate the resulting network traffic and increased complexity this type of design brings along [4]. We unquestionably accept the drawback if we would need the entire state all the time. However, as we only would need the entire state to accumulate the Git metrics, which is a performance-wise cheap task, we opted for a more self-sustaining design without the need of external state-input. Thus, we can keep the workload of the environment the *code-service* runs on lower during analysis, as it only has to handle and store the incoming data, not to send anything back.

Therefore, we will focus on a lightweight, almost unidirectional stateless approach where we only request the current remote state once per execution of the analysis, independent of the number of commits we need to analyze or the content already stored in the database. This simplifies the *code-agent* and *code-service* and moves the calculation of *metrics across multiple commits* towards a visualization that has to be implemented downstream in the future. Since the visualization still has to aggregate the required data, the overhead for calculating these metrics is minimal.

Advantages

The stateless system is more lightweight and better suited for integration into the CI pipeline. A Docker container can be used as the execution environment and is fully self-contained, no external information is needed, and no information needs to be kept. The upgrade process is a simple drop-in solution when migrating from one pipeline to another or after an update of the *code-agent*.

Disadvantages

As implied above, the calculation of *metrics across multiple commits* is limited due to the absence of data before the current start commit.

If the future shows the design choice to hand over the responsibility to the visualization led to downstream problems, the current design's extension is not trivial. *Code-agent* and *code-service* need to be extended, as well as the entire analysis loop.

4.4 Continuous Integration

To prepare this analysis service for use as part of the continuous Integration pipeline, we must provide the service as a stand-alone and easy-to-embed software package. While some options exist, the readily available built-in support of Docker containers with Quarkus and the GitLab CI has settled the decision. Settings can be exposed as environment variables, and a pre-built container can be provided. This eases the integration of this service into already existing and newly created projects. Even though the project and its pre-built container are designed explicitly for the GitLab CI, integration to other CI services should be possible by this design, as we will see in Section 6.2.4.

Chapter 5

Implementation

While we now understand how the project is designed, we need to focus on the implementation. We not examine how we can clone repositories, checkout branches, and walk commits. We will see how to detect changes in Java source files and apply file filters. After that, we see how to interact with the AST from JavaParser and collect the needed structural data with the correct types. We look at the communication between *code-agent* and *code-service*, the collection, and calculation of the different metrics. Lastly, we look at how we built the service as a Docker container and how to customize it.

5.1 Project Structure

As discussed in Section 4.3, the project resembles a service that can be integrated into new or existing projects. Even though the analysis is meant to be used with ExplorViz, it is a fully functional service on its own. As the communication between the *code-service* and *code-agent* is handled by gRPC, a third-party endpoint, implementing a custom gRPC server, can be used to integrate it into any other project for utilizing the provided analysis features.

5.1.1 Code-Service

The code-service acts as a simple gateway for the analysis data; it is meant to be integrated into ExplorViz and is responsible for handling the saving of the incoming data packets. The code-service is a simple data endpoint without real logic. It provides a gRPC endpoint and prints the data to the standard-out. An potential integration for the code-service into ExplorViz is shown in Figure 5.1. This could be easily adapted to other projects designs as the code-service is only a gateway for the incoming data.

To integrate the code-service into ExplorViz, or any other software system, the provided methods in the `KafkaGateway` class need to be populated. The KafkaGateway, cleaned from logging calls, is shown in Listing 5.1. The method `processCommitReport` needs to handle incoming commit reports. Each is sent after the analysis of a commit is complete. `processFileData` gets called for each incoming FileData packet, containing all the analysis data for a single file. Lastly, `processStateData` gets called if the code-agent requests the current state of the analysis data the database holds. For an explanation regarding the

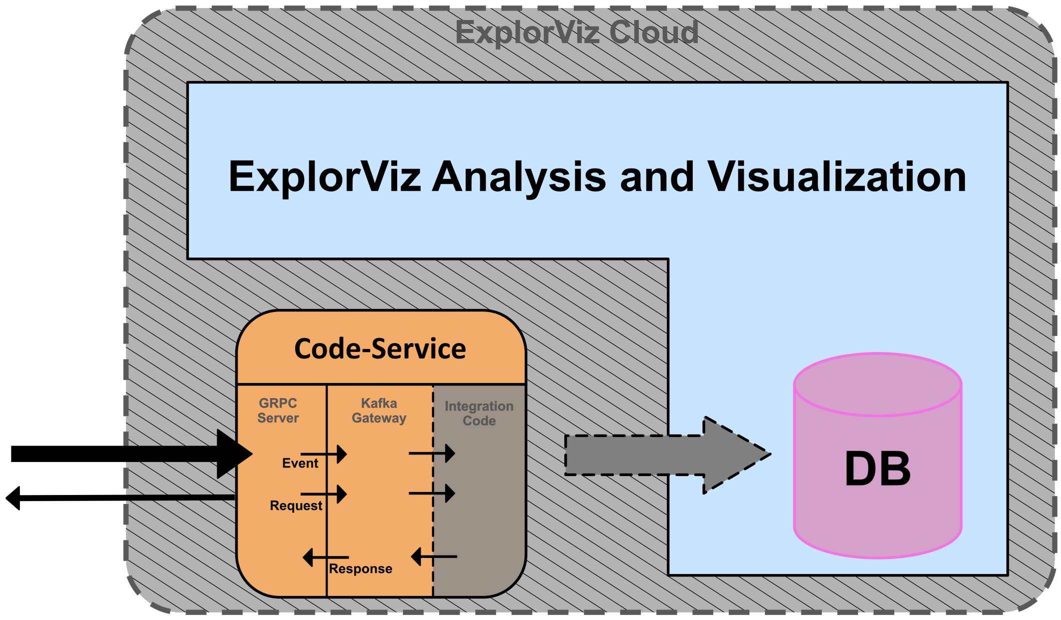


Figure 5.1. Proposed integration of the code-service into ExplorViz. Due to the abstract design an integration into other projects is possible in a similar way.

implementation, consider the commented original file within the code-service repository[1].

```java
public class KafkaGateway {
  public void processCommitReport(final CommitReportData commitReportData) {}
  public void processFileData(final FileData fileData) {}
  public String processStateData(final StateDataRequest stateDataRequest) {
    return "";
  }
}
```

Listing 5.1. Simplified KafkaGateway.

5.1.2 Code-Agent

The *code-agent* is the runner of the analysis. Unlike the code-service, which could be implemented differently as a tailored gRPC endpoint, the code-agent is served and used as a ready-to-run Docker container. All functionality regarding static code analysis and

[1]https://github.com/ExplorViz/code-service/tree/jp-dev

repository management is implemented in it, and therefore the following sections focus primarily on the work done there.

5.2 Git Repository

To be able to analyze the content of Git repositories, we need to interact with repositories in the first place. The code-agent uses JGit to control the repository as it provides an abstraction layer for easy, Git-CLI-like interactions and a feature-rich Java-API for precise repository handling. While the main scope of application for the service is the integration into CI pipelines and will therefore handle the cloning itself, it should also be possible to run the analysis with a local repository provided by the user. After loading the repository, the next step is to gain access to the commits we are interested in. Therefore we have to decide which branch should be analyzed.

5.2.1 Repository Access

As the service also allows the option to analyze local repositories, Listing 5.2 shows the straightforward decision process of whether or not the local repository is used. The variable `localRepositoryPath` gets populated by an environment variable (see Appendix) or is blank if the user does not define the variable. Thus, the presence of this value makes the decision to clone the repositories. On the other hand, the `remoteRepositoryObject` holds all the values needed to clone a repository. As we will see in Section 6.2.1, by proving an URL and a branch name, the analysis enters the `downloadGitRepository` method and begins the cloning.

While the cloning of the repository is mainly handled by JGit (see Listing 5.3), it is vital to perform error checking and sanitizing of user input before we handle the data to JGit. A tiny part can be seen as the call to `convertSshToHttps` to ensure the user provided a valid HTTP URL to a Git repository. Besides this, as we perform network communication, we also have to deal with its standard errors. As seen in the mentioned listing, it is also possible to provide a credentials provider opening the possibility to clone private repositories. After the JGit command, we have access to a complete cloned repository checked out with the given branch ready to work with.

```java
public Repository getGitRepository(final String localRepositoryPath,
      final RemoteRepositoryObject remoteRepositoryObject)
      throws IOException, GitAPIException {
  if (localRepositoryPath.isBlank()) {
    return this.downloadGitRepository(remoteRepositoryObject);
  } else {
    return this.openGitRepository(localRepositoryPath, remoteRepositoryObject.
        getBranchName());
  }
}
```

Listing 5.2. Decision whether to clone or to open an existion repository.

```
final Map.Entry<Boolean, String> checkedRepositoryUrl = convertSshToHttps(
    remoteRepositoryObject.getUrl());
String repoPath = remoteRepositoryObject.getStoragePath();
this.git = Git.cloneRepository().setURI(checkedRepositoryUrl.getValue())
    .setCredentialsProvider(remoteRepositoryObject.getCredentialsProvider())
    .setDirectory(new File(repoPath)).setBranch(remoteRepositoryObject.
        getBranchNameOrNull())
    .call();
return this.git.getRepository();
```

Listing 5.3. Cloning a repository with JGit. Stripped down version of the `downloadGitRepository` method.

5.2.2 Commit Walking

While having the repository available at our fingertips for further processing, we need access to each commit of the branch and process them in the order from the oldest to the most recent. Conveniently, JGit provides the `RevWalk` object to *walk along* the repository's commit tree. As we are only interested in a single branch, we restrict it to include commits from the branch in question. Then we must sort the RevWalk sort based on commit time in ascending order. Now we have an iterator to walk all commits included within the branch in the order we need. While we loop over the commits, we must remember that we do not have access to the repository's state at the commit we are pointing to. We can only interact with the files at the demanded state after a `checkout` of the specific commit. This puts much strain on the storage as we have to restore the state of each commit in the file system. To minimize the storage access as best as possible, it is sensible to check for changes before checkout. Access to the data for the analysis is only needed if the monitored files change.

5.2.3 Change Detection

While we can access the *changes* produced by a commit, we can not run an analysis solely on this data. Nevertheless, we can detect which commits contain changes in files we are interested in. Thus, it is possible to pre-select commits for which a checkout is mandatory. Not only can we skip commits containing no changes in Java files, but it is also possible to detect the files that contain the changes. Even though it would be better to differentiate the *kind of change*, e.g., if it is a comment or a line of code, we decided to treat all changes equally owing to simplicity. A *pre-analysis* would be required to detect the difference, ultimately resulting in a complex project in itself.

To provide a list of changed files for the analysis, the `listDiff` method is implemented in the project. A simplified version to have a closer look at is shown in Listing 5.4

```
 1 public List<FileDescriptor> listDiff(final Repository repository,
 2                                       final RevCommit oldCommit,
 3                                       final RevCommit newCommit,
 4                                       final List<String> pathRestrictions) {
 5     final List<FileDescriptor> objectIdList = new ArrayList<>();
 6     final TreeFilter filter = getJavaFileTreeFilter(pathRestrictions);
 7 
 8     final List<DiffEntry> diffs = this.git.diff()
 9         .setOldTree(prepareTreeParser(repository, oldCommit.get().getTree()))
10         .setNewTree(prepareTreeParser(repository, newCommit.getTree())).
               setPathFilter(filter)
11         .call();
12     for (final DiffEntry diff : diffs) {
13         if (diff.getChangeType().equals(DiffEntry.ChangeType.DELETE)) {
14             continue;
15         }
16         objectIdList.add(new FileDescriptor(diff.getNewId().toObjectId(), diff.
               getNewPath()));
17     }
18     return objectIdList;
19   }
```

Listing 5.4. Simplified version of the `listDiff` method generating a list of changed file descriptors.

The method is provided with a reference to the repository, which does not add much value to our overview as it is only needed for some internal handling. Further, `oldCommit` and `newCommit` are given to point at the commits we want to get the difference from. `pathRestrictions` should not interest us further as it is only needed to restrict the analysis to special directories. In line 6, a filter is created to only check for Java files as these are the only file types the analysis will handle. Starting with line 8, JGit generates a list of changes between the old and the new commit. The `for` loop spanning lines 12 to 18 creates a `FileDescriptor`, a simple container object holding data for the later analysis; data to find the changed files easily as well as information about the modifications. The `DELETE` case is handled specially as - even though it is a change where all the content is erased from a file, and therefore the file itself - no meaningful data can be gathered from an empty file. Therefore, we can skip this file in the analysis. Lastly, the list of changes gets returned. By checking if the list is empty, we can detect if a checkout of the current commit can be skipped, reducing unnecessary I/O operations. After all this repository handling, the service can clone a repository, walk over all available commits that belong to the branch, and serve a list of actually changed files for each commit that can be analyzed.

5.3 File Analysis

Prior to any collection of structural data or calculation of metrics, every changed source file needs to be converted into its corresponding AST representation. By using JavaParser, we can rely on a very capable and mature parsing project. Feeding in the source code we want to analyze, JavaParser returns an AST for the files as well as the ability to handle its building blocks by providing access to the data using the *visitor pattern*.

5.3.1 Structural Data Collection

The structural data collection's main task is filling the data model previously shown in Figure 4.1. All entries, except for the metric data, need to be filled before we can compute any of them, as we need the backbone first, filled with class and method names. The task seems simple, but we must factor in some Java peculiarities. Every Java file has a public class inside (counting `enum` and `interface` as well), but can hold *infinitely many* non-public classes as well. Thus, we need to keep track of the class we currently analyze. This is needed as the provided visitor for the AST automatically visits every node in the proper order but does not provide easy access to parent nodes. It is, therefore, unknown which class a method belongs to. Handling the tracking of classes directly in the visitors does not seem like a good design decision as the *current class* is more data-specific information and not relevant to the current visitor; thus, we will discuss it in Section 5.3.3.

While the file can hold multiple classes, it is also possible that they are nested within another class as already described in Section 4.1.1; the solution to this is to push the class name to a *class stack* and generate the FQN based on the package name and entire class stack. More complex is the handling of *anonymous classes*; an example is shown in Listing 5.5. The interface's FQN is easy to determine, `code.analysis.MyInterface`, just like the public class', `code.analysis.MyClass`. But what about the anonymous class held by the `clazz` field? Using the nested class approach and using `code.analysis.MyClass.MyInterface` would ultimately fail upon the second instantiation of an anonymous `MyInterface` class. Using the name of the field would result in an FQN that sounds arbitrary. Further, the AST treats the name as a parent object of the object creation expression. A simple approach to solve this is adding an index to the type that counts up on every encounter of another anonymous class. The resulting FQN, therefore, is `code.analysis.MyClass.MyInterface#1`.

```
package code.analysis;
interface MyInterface {
  public void doSomething();
}
public class MyClass implements MyInterface{
  MyInterface clazz = new MyInterface() {
    public void doSomething() {}
  };
  public void doSomething() {}
  public void doSomething(int a) {}
}
```

Listing 5.5. Example class containing an anonymous class.

The snippet also shows another problem we must deal with: Overloaded methods. Both methods of `MyClass` currently result in `code.analysis.MyClass.doSomething`. We could solve it like we already did for the anonymous classes. However, a cleaner solution is to hash the parameter types, as they have to be unique. Otherwise, the Java code would not be valid. Thus, the parameterless method gets the FQN `code.analysis.MyClass.doSomething#1` and the method expecting an `int` gets `code.analysis.MyClass.doSomething#1980e`. Once the FQN generating is implemented, the adding of remaining data entries such as *super class*, *interfaces*, or *modifiers* is only a matter of interacting with the AST to get the data.

5.3.2 Type Resolving

At the same time collecting the structural data, we must keep in mind that there is also a need to get the types of methods, parameters, or fields. More precisely, we need the FQN of the types to show where they are defined. Finding out the type is a non-trivial task, as we have to look into every file contained within the current package and every file stated in the imports. Within these files, the type has to be searched. The JavaSymbolSolver, contained in the JavaParser project, can be used to solve many types. The `ReflectionTypeSolver` is one of the solvers available. It is used to solve *primitive* and *built-in* types (see Section 4.1.2) on its own. The symbol solver gets attached to the JavaParser and is therefore available during the *tree-walking* within the *visitor*. As AST parameter node objects, as well as other nodes containing types, feature a field to get an unresolved type object, this type can quickly be resolved by calling the *resolve()* method of the object. Once called, the type solver tries to solve the type within its limits. As the ReflectionTypeSolver can not resolve any *package-types* nor *project-types*, we need to add an `JavaParserTypeSolver`. For smaller projects, it might be ok to provide the solver with the entire source folder. For larger projects, the environment variable `ANALYSIS_SOURCE_DIRS` (see Appendix) might be used to restrict the type resolving to a particular directory.

Even though many types can be resolved this way, by using these two type solvers, we can not resolve types defined within external jars, as explained in Section 4.1.2. Thus, we can use the simple import lookup as a workaround. It also serves as a simple fallback if an exception occurs during the resolution. The lookup gets more complicated as we must pay attention to correctly handle *arrays* and *generics*.

5.3.3 Context Handling

As mentioned in Section 5.1, the analysis data has to be sent to the code-service, which will be explained in detail in Section 5.4. While the data format is already set, since we want to use protobuf, we only need to define how the data we want to send can be defined in the protobuf format. To avoid an intermittent storage object to save the structural data during the analysis, providing wrapper objects for the protobuf message objects allows us to use them directly as the storage object while being able to embed logic to the wrapper. This creates an abstraction layer for data adding. The wrappers also provide handy methods to easily keep track of the current context, which facilitates the addition of new structural data. A similar but more abstract wrapper is also provided for the metric visitors (see Section 5.5).

5.4 Communication

As mentioned in Section 5.1, code-agent and code-service potentially run on different machines and therefore need a way to exchange data packets over the network.

As we designed the analysis data objects as protobuf messages, extending both Java projects to transmit and receive the already defined messages is a simple matter of adding the *gRPC service* code to the respective *.proto* files as seen in Listing 5.6. The listed code is all we need to define a communication where the client sends a *FileData* packet to the server and expects *nothing* as the response.

```
service FileDataService {
  rpc sendFileData (FileData) returns (google.protobuf.Empty) {}
}
```

Listing 5.6. gRPC service definition for the FileData message.

The *protoc* will handle all the code generation for the needed client and server code for us. We only have to get the gRPC client for the code-agent side, which is conveniently served by a Quarkus annotation. Listing 5.7 shows all the code needed to use the gRPC client and send the protobuf message.

```
@GrpcClient(GRPC_CLIENT_NAME)
FileDataServiceGrpc.FileDataServiceBlockingStub fileDataGrpcClient;

public void sendFileData(final FileData fileData) {
    fileDataGrpcClient.sendFileData(fileData);
}
```

Listing 5.7. gRPC client implementation in the code-agent.

On the server (*code-service*) side, the protobuf file is the same as the communication definition is unaltered. The server code is similarly straightforward as shown in Listing 5.8.

```java
@GrpcService
public class FileDataServiceImpl implements FileDataService {
  @Override
  public Uni<Empty> sendFileData(final FileData request) {
    // do something with the data, e.g., forward to ExplorViz
    return Uni.createFrom().item(() -> Empty.newBuilder().build());
  }
}
```

Listing 5.8. gRPC server implementation in the code-service.

To conclude the implemented design, Figure 5.2 shows the resulting communication paths. The *FileData* and *CommitReport* messages are sent when they are ready on the client side and do not request any data in return.

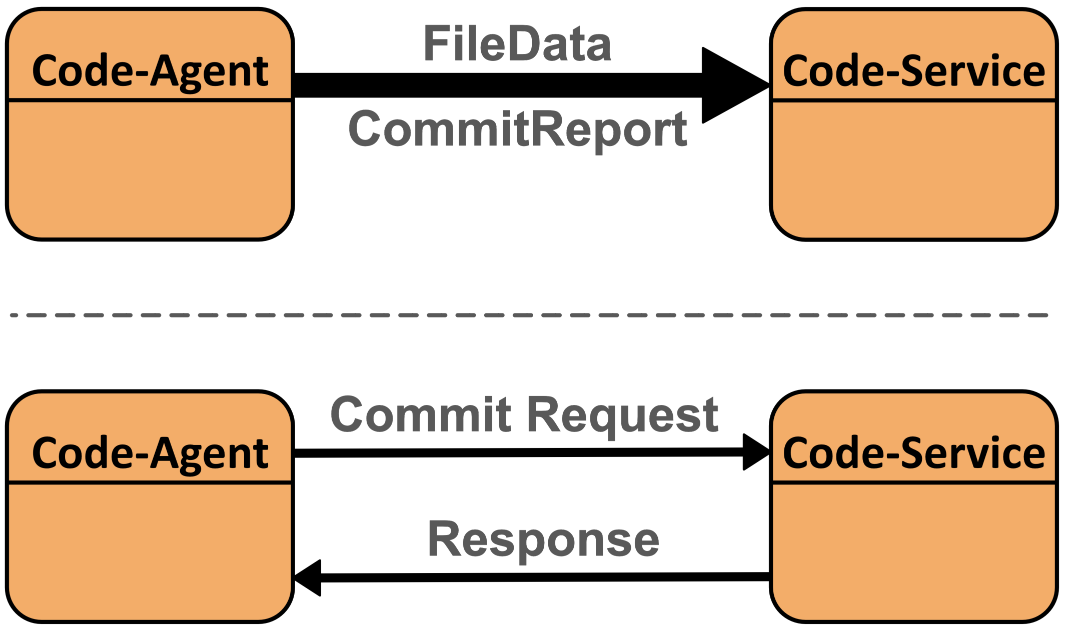


Figure 5.2. Communication between code-agent and code-service via gRPC.

The *StateData*, on the other hand, follows the typical *request-response* exchange pattern. The *code-agent* sends a request containing the current branch to analyze, and the *code-service* responds by sending the newest commit's SHA available in the database.

5.5 Source Code Metrics

To provide an easy-to-extend project structure, all metrics are implemented as independent AST visitors. Thus, the error handling can be kept simple as a failing metric calculation does not result in a complete loss of all the following metrics. All visitors follow the same structure. They use a *MetricAppender* to keep the FQN handling out of the visitor while enabling them to add metrics whenever they want. Thus, the metric visitors are easier to maintain as they only contain little *non-metric* code. The *MetricAppender* basically provides the same functionalities as the *FileDataHandler* used for the structural data construction. However, since the structural data creates entries for methods and classes, the usage is much more heavy-weight and depends on knowledge of the internal representation. Adding more metrics to the analysis is much more likely in the future than restructuring the data representation; the *MetricAppender* features a simplified and easier-to-use interface.

```java
public void visit(final ClassOrInterfaceDeclaration n, final Pair<MetricAppender,
    Object> data) {
  data.a.enterClass(n);
  data.a.putClassMetric("someClassMetric", "classMetricValue");
  super.visit(n, data);
  data.a.leaveClass();
}

public void visit(final MethodDeclaration n, final Pair<MetricAppender, Object>
    data) {
  data.a.enterMethod(n);
  data.a.putMethodMetric("someMethodMetric", "methodMetricValue");
  super.visit(n, data);
  data.a.leaveMethod();
}
```

Listing 5.9. Methods to handle *MethodDeclaration* and *ClassOrInterfaceDeclaration* nodes so that the tracking of the current method or class FQN is done by the MetricAppender and does not pollute the metric visitor. As the MetricAppender is the first entry of the `Pair` it is accessed via `data.a`.

5.5.1 Lines of Code

The *Lines of Code* metric is the only exception to the above rule of *one visitor per metric*, as it is part of the structural data collection. This is because calculating LOC is performance-wise cheap, as JavaParser directly provides the number of lines within a file, class, or method. Further, the generated AST contains the number of comments so that the lines of source code can be calculated based on already available information. The LOC calculation should also be relatively robust as the only error it can encounter is the absence of the needed information from the AST, which ultimately resulted from a failed parsing in the first place.

5.5.2 Nested Block Depth

The *nested block depth* metric code is simple in structure. Besides some complexity regarding the MetricAppender, it only visits all control structure nodes (`for`, `for-each`, `while`, `do-while`, `if`, `try`, `switch` and, `case`), method and constructor nodes, and some special nodes (*lambda expressions* and *synchronized blocks*). For every entered node, a counter increases, and for every left node, it decreases. Listing 5.10 shows the implementation of the `for` node handling, which is virtually the same for all other control structure nodes and special nodes. `currentDepth` is the counter, and `maxDepth` is the variable to hold the maximum depth found in the context.

```java
public void visit(final ForStmt n, final Pair<MetricAppender, Object> arg) {
  currentDepth++;
  maxDepth = Math.max(maxDepth, currentDepth);
  super.visit(n, arg);
  currentDepth--;
}
```

Listing 5.10. Implementation of the depth calculation for a for-loop.

This metric gets calculated for every method or constructor within a file and added to the analysis data.

5.5.3 Cyclomatic Complexity

The *cyclomatic complexity* metric collection works similarly to the above explained *nested block depth*. The major difference is that we count all control structure nodes independent of their depth. The cyclomatic complexity is not only a method-wide metric but is also used for classes and the entire file. Further, the similar *wheighted cyclomatic complexity* needs also be calculated. As data collection is more advanced than before, we use a combination of the MetricAppender's ability to track the current method with a simple `HashMap` to keep the visitor as clean as possible. The Map takes the method's FQN as key and has an `Integer` as value, representing the number of occurring control structures. Listing 5.5.3 shows the exemplary handling of a `for` node. The `addOccurrence` method is a wrapper for the `HashMap` access and increases the stored number by one. In the following lines 3 and 4, we check if the `for` contained a *compare* statement (which is `i<x` in `for(int i=0; i<x;i++)`) to analyze it further. The cyclomatic complexity does not only count the keyworded control flow statements. It also takes the logical and binary expressions `&&`, `||`, `&`, and `|` into account. The calculating was implemented as close to the original as possible [15].

```java
public void visit(final ForStmt statement, final Pair<MetricAppender, Object> data
    ) {
  addOccurrence(data.a.getCurrentMethodName());
  if (statement.getCompare().isPresent()) {
    conditionCheck(statement.getCompare().get().toString(), data);
  }
  super.visit(statement, data);
}
```

After a complete analysis of the entire file, the `HashMap` contains all methods and their cyclomatic complexity values. These can be directly added to the FileData as metric entries. The class metrics are calculated by adding up all metric values of methods belonging to the class. Same for the file metric. The *wheighted cyclomatic complexity* is the average value from all methods, resulting in a trivial computation of the class's cyclomatic complexity value divided by the number of methods the class contains.

5.5.4 Lack of Cohesion in Methods

The LCOM4 metric is the most complex metric to calculate compared to the metrics above. As shown in Figure 4.2, a graph representation of source code is an excellent choice to compute the dependencies between methods and fields, accomplished by adding a vertex for each method and field to an undirected graph. Next, we must analyze the method's body for access to fields. While JavaParser helped us in the past, it does not provide the functionality to detect the context of fields or methods. Thus, `ClassA.valueA` and `this.valueA` are both field accesses, even though the first could access a field from a different class and the second is undoubtedly a local field access. However, even the first is locally accessed if the current class's name is `ClassA`. Consequently, we must check every field access to see if the field's name matches one of the class's field names.

Moreover, if it does, it also needs to be verified if the scope is either `this` or the name of the current class. Only then is the field accessed within the method, and we can add an edge to connect both vertices. Further, as JavaParser does not distinguish between a local variable access and a field access, as they look the same, we need to handle *variable shadowing*. We look into Listing 5.11 to discuss field shadowing.

```java
class A {
    int x = 1;
    void methodA() {
        int x = 4;
        ClassB.methodB();
        x++;
    }
    void methodB() {
```

```
9          x++;
10     }
11 }
```

Listing 5.11. Example of *field shadowing* and scopes for method calls.

In the example, the variable `x` in line 4 shadows the class field `x` as they have the same name. In line 6, the *local* variable `x` is accessed; thus, `methodA` does not access any class fields (belonging to the class `A`). In line 9, no other variable shadows the field `x`; therefore, `methodB` accesses the field. Another similar problem is in line 5, where `methodB` is called but not the one from class `A`. Just like the fields, we also have to check the context for methods calls.

Specially treated are methods with empty bodies and inherited methods as they *do belong* to the super class [11]. Both get skipped in the analysis process.

Once finished, we can examine the resulting graph. If all vertices are transitively connected, the LCOM value is 1 and can be added as a class metric via the MetricAppender. Otherwise, the number of *sub-graphs* dictate the LCOM value. The only reason the resulting value is 0 is that it is an empty class. This also can happen if *all* methods are inherited.

5.5.5 Extending

The above metrics are implemented as a starting point to build an extendable working framework. As the already implemented metrics have different complexities, they should serve as good sources of information for future metrics. The MetricAppender is designed to abstract the internal data structure and handling further. As we have seen, the visitors are implemented with a `Pair` object as the accumulator. The left object is reserved for the MetricAppender, while the right object is free for any metric implementation. The project contains a `VisitorStub` that implements fundamental class and method handling and can be used as a blueprint. Adding a new metric visitor to the project requires setting it up in the `JavaParserService`'s `calculateMetrics` method.

5.6 Git Metrics

The last kind of metric we need to collect is the Git metrics. As already described in Section 4.2.3, the metrics are only meaningful after the visualization has aggregated them. As further data processing is impossible due to the limitations of the lightweight stateless service design, we can only collect the data for future processing.

Modifications

Even though the data is only needed now, the collection happens in the same methods responsible for the *change detection* described in Section 5.2.3. There, we have access to the differences between the last revision of the file and the current content. The retrieved data, containing the number of modified, added, and deleted lines, is added to the file's *FileData* packet.

Authors

Authors are identifiable by their self-chosen usernames or e-mail addresses for the Git account. As the username is more prone to changes, we selected the e-mail as the author identifier. Thus, the *autorIdent* attached to each commit is used to get the identifying e-mail address and added to each analysis file created for the commit in question.

5.7 Docker Container

As Quarkus follows the *container first* approach, it is straightforward to create a Docker container from a Quarkus-based project using jib[2]. All needed build tools are already integrated or available as extensions. The created Docker container was pushed to DockerHub[3] to provide a ready-to-use analysis service.

5.7.1 GitLab Integration

To manage functionality and adjust runtime behavior, environment variables are used. Quarkus already handle the comfortable integration of these. Listing 5.12 shows the connection of environment variables to internally used properties. The environment variable **`ANALYSIS_REMOTE_URL`** is settable by the user. The second variable, **`CI_REPOSITORY_URL`**, is provided by the GitLab CI. Thus, if the user does not specify the URL but runs the container within the GitLab CI, the URL gets automatically set to the current repository. This is predefined for all mandatory variables. We will see the integration in Section 6.2.3.

```
explorviz.gitanalysis.remote.url=${ANALYSIS_REMOTE_URL:${CI_REPOSITORY_URL:}}
explorviz.gitanalysis.branch=${ANALYSIS_BRANCH:${CI_COMMIT_BRANCH:}}
```

Listing 5.12. Excerpt from the project's `application.properties`.

[2]https://github.com/GoogleContainerTools/jib
[3]https://hub.docker.com/

Chapter 6

Evaluation

As the project implementation is done and we need to verify the outcomings, we will look at some example scenarios where the analysis is used and integrated into different CI runners. After that, we will summarize our findings from the example applications and assess the project in its current state.

6.1 Goals

The main goal of the evaluation is to verify the operability of the Docker container approach and the robustness of the analysis and to get a general understanding of the magnitude of the performance. Further, we want to check the reconfigurability of the analysis and the feasibility of integrating it with other CI providers than the intended. We will evaluate the current state of the analysis in the context of aiding in program comprehension and assess the meaningfulness of the output data. This involves running an instance in an easy-to-monitor, local and controllable environment to verify the produced output. On the other hand, to validate the portability, we include the service into different external runners without the need to capture complex output data.

6.2 Example Scenarios

To show and evaluate the capabilities and the usage of the analysis service, we will look at 4 example applications of the service. In Section 6.2.1, we will see how to set up the service as a standalone application and run it on the *main* branch of the *spring-petclinic*[1] project. Stepping up in project size, we perform an analysis of the mature *plantUML*[2] project to demonstrate some error handling and the robustness of the analysis as well as current limitations in Section 6.2.2.

After that, in Section 6.2.3, we will see how to integrate the service into the GitLab CI running on a mockup project, finishing with a basic integration with GitHub Actions to validate the portability of the container design in Section 6.2.4.

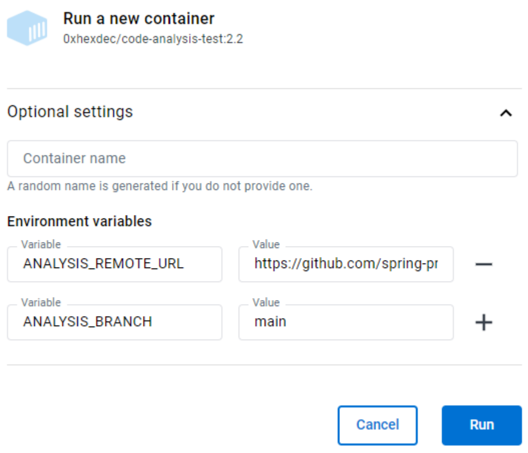
[1]https://github.com/spring-projects/spring-petclinic
[2]https://github.com/plantuml/plantuml

The pre-built Docker container `0xhexdec/code-analysis-test:2.2`[3] is used in all example applications below. The container is built from the code-agent[4] project at commit `#cd704da8`. Standalone example applications run on Docker for Windows, and both CI integrated example applications on the CI's respective operating system. As a reference, all standalone tests were performed on a mediocre Intel Core i7-6700HQ running Windows 10 with 16 gigabytes of memory.

6.2.1 Standalone Analysis: PetClinic

Before diving into the more complex integration of the analysis service, we will run the Docker container as a standalone non-integrated system. This could be an interesting application for the developer who wants to run the analysis locally on the development machine.

We must pull the pre-built Docker image first to analyze the *spring-petclinic* project. Now, we select the pulled image in Docker Desktop and click the run button. The ready-to-run configuration should look similar to the one shown in Figure 6.1, where the *main* branch is selected, and the URL from the GitHub-hosted repository is set. This will instruct the analysis to clone the petclinic repository to a temporary directory in the container's file system as well as run the analysis on all commits of the *main* branch. The analysis data will be saved in the container as *json* files.


Run a new container
0xhexdec/code-analysis-test:2.2

Optional settings

Container name
A random name is generated if you do not provide one.

Environment variables

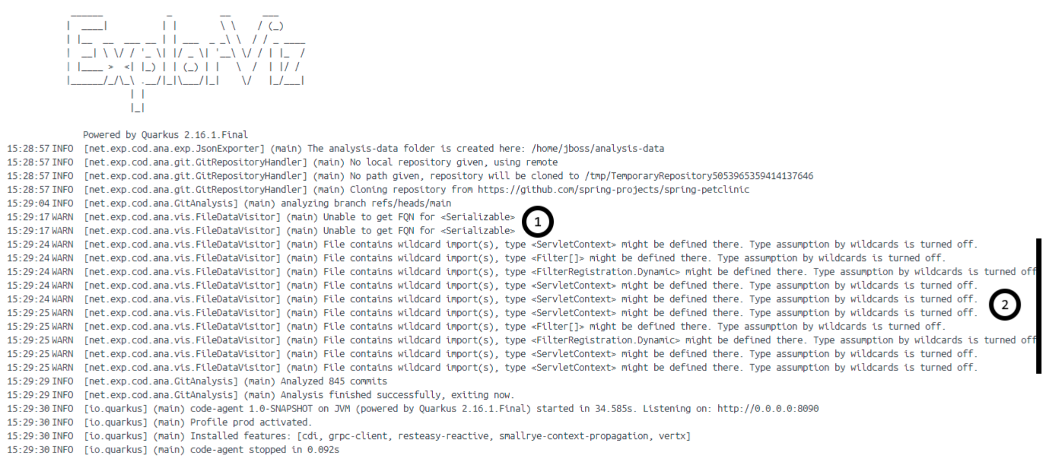
Variable	Value
ANALYSIS_REMOTE_URL	https://github.com/spring-pr
ANALYSIS_BRANCH	main

Figure 6.1. Docker Desktop - Configuration for the PetClinic standalone analysis.

[3]https://hub.docker.com/r/0xhexdec/code-analysis-test
[4]https://github.com/ExplorViz/code-agent/tree/jp-dev

Execution and Output

After the execution, the output will look similar to the one shown in Figure 6.2. The runtime was just shy over 33 seconds for 845 commits and generated 1848 JSON files (5.5 Mbyte of data) saved in the folder `/home/jboss/analysis-data`. As 845 of these files are *commit reports*, we are left with 1003 FileData files (named according to the scheme: <file name>_<commitID>.json), containing all structural data as well as the calculated metrics. A closer inspection of the output log shows two different *warnings* during the execution. The first (①) is a *type warning*, noticing the type `Serializable` could not be resolved. After investigating this error, it was found that this warning was actually not an error by the analysis. The source code file was missing the required import; thus, the file should have created a compiler error as well. The second warning (②) notices the files contain wildcard imports. The *wildcard type assumption* is deactivated by default, and types that wildcard imports might define result in warnings. These warnings disappear by adding the environment variable `ANALYSIS_RESOLVE_WILDCARDS=true`.



```
   _____          _         __     ___
  |  ___|        | |        \ \   / (_)
  | |__  __  ___ | | ___  __\ \  / / _ ___
  |  __| \ \/ / '_ \| |/ _ \| '__\ \/ / | |_  /
  | |____ >  <| |_) | | (_) | |   \  /  | |/ /
  |______/_/\_\ .__/|_|\___/|_|    \/   |_/___|
              | |
              |_|

          Powered by Quarkus 2.16.1.Final
15:28:57 INFO  [net.exp.cod.ana.exp.JsonExporter] (main) The analysis-data folder is created here: /home/jboss/analysis-data
15:28:57 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) No local repository given, using remote
15:28:57 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) No path given, repository will be cloned to /tmp/TemporaryRepository5053965359414137646
15:28:57 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) Cloning repository from https://github.com/spring-projects/spring-petclinic
15:29:04 INFO  [net.exp.cod.ana.GitAnalysis] (main) analyzing branch refs/heads/main
15:29:17 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) Unable to get FQN for <Serializable>
15:29:17 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) Unable to get FQN for <Serializable>
15:29:24 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <ServletContext> might be defined there. Type assumption by wildcards is turned off.
15:29:24 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <Filter[]> might be defined there. Type assumption by wildcards is turned off.
15:29:24 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <FilterRegistration.Dynamic> might be defined there. Type assumption by wildcards is turned off
15:29:24 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <ServletContext> might be defined there. Type assumption by wildcards is turned off.
15:29:24 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <ServletContext> might be defined there. Type assumption by wildcards is turned off.
15:29:25 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <ServletContext> might be defined there. Type assumption by wildcards is turned off.
15:29:25 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <Filter[]> might be defined there. Type assumption by wildcards is turned off.
15:29:25 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <FilterRegistration.Dynamic> might be defined there. Type assumption by wildcards is turned off
15:29:25 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <ServletContext> might be defined there. Type assumption by wildcards is turned off.
15:29:25 WARN  [net.exp.cod.ana.vis.FileDataVisitor] (main) File contains wildcard import(s), type <ServletContext> might be defined there. Type assumption by wildcards is turned off.
15:29:29 INFO  [net.exp.cod.ana.GitAnalysis] (main) Analyzed 845 commits
15:29:29 INFO  [net.exp.cod.ana.GitAnalysis] (main) Analysis finished successfully, exiting now.
15:29:30 INFO  [io.quarkus] (main) code-agent 1.0-SNAPSHOT on JVM (powered by Quarkus 2.16.1.Final) started in 34.585s. Listening on: http://0.0.0.0:8090
15:29:30 INFO  [io.quarkus] (main) Profile prod activated.
15:29:30 INFO  [io.quarkus] (main) Installed features: [cdi, grpc-client, resteasy-reactive, smallrye-context-propagation, vertx]
15:29:30 INFO  [io.quarkus] (main) code-agent stopped in 0.092s
```

Figure 6.2. Output from the petclinic analysis.

Generated Data

Even though the raw generated data is not meant to be humanly readable and needs to be processed and visualized to show its full potential, we look at the small example in Listing 6.1 to understand what we got.

```
"commitID": "bcda93f280e9174f4e5c44fc830a864a963633a5",
"fileName": "AbstractTraceAspect.java",
"packageName": "org.springframework.samples.petclinic.aspects",
"importName": ["org.aspectj.lang.JoinPoint", ...],
"classData": {
  "org.springframework.samples.petclinic.aspects.AbstractTraceAspect": {
    "type": "ABSTRACT_CLASS", "modifier": ["public", "abstract"],
    "field": [{
      "name": "logger", "type": "org.slf4j.Logger",
      "modifier": ["private", "static", "final"]
    }],
    "methodData": {
      "org.springframework./.../.aspects.AbstractTraceAspect.trace#ac4c0c48": {
        "returnType": "void", "modifier": ["public"],
        "parameter": [{
          "name": "jpsp","type": "org.aspectj.lang.JoinPoint.StaticPart"
        }],
        "metric": {
          "loc": "6", ... ,"nestedBlockDepth": "2"
        }
      },
      "org.springframework./.../.aspects.AbstractTraceAspect.traced#1": {
        ... }
    },
    "metric": {
      "loc": "16",
      "cyclomatic_complexity": "1",
      "cyclomatic_complexity_weighted": "1",
      "LCOM4": "1"
    }}},
...
"author": "someone@email.com",
"modifiedLines": 1, "addedLines": 2, "deletedLines": 2
```

Listing 6.1. Shortened analysis data for *AbstractTraceAspect.java* at commit `#bcda93f`.

As we can see in the data excerpt, the file contains one *Abstract Class* that contains two methods, called `trace`. One expects a `org.aspectj.lang.JoinPoint.StaticPart` as a parameter, and the other is parameterless. Metrics were created for methods, the class, and the file. The Git data is added in the form of the author's email address (changed to something arbitrary), and the number of modified lines hints at a minor change. In fact, only the logger library was replaced from *apache.commons* to *slf4j*.

6.2.2 Standalone Analysis: plantUML

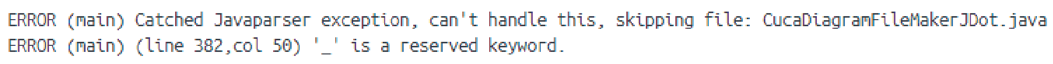
As we want to simulate a remote execution for this repository, we need an endpoint to send the analysis data. For this application, we run the code-service[5] inside IntelliJ IDEA[6] on the same machine as the Docker container. This is the most straightforward setup as it does not require any external port forwarding and can be used as a first project to familiarize oneself with the analysis service. Before anything else, we need to run the `quarkusDev` gradle task to run the code-service. Once the service is running, we can go on and configure the code-agent to analyze the repository. Listing 6.2 specifies the variable values to configure the service to analyze the main source directory of plantUML's code.

```
ANALYSIS_REMOTE_URL=https://github.com/plantuml/plantuml
ANALYSIS_BRANCH=master
ANALYSIS_SEND_TO_REMOTE=true
GRPC_HOST=host.docker.internal
ANALYSIS_SOURCE_DIRS=src
ANALYSIS_RESTRICT_DIRS=src/net/sourceforge/plantuml
```

Listing 6.2. Environment variables used to run the plantUML analysis.

The source folder restricted analysis ran for roughly 43 minutes and analyzed 1130 commits until finished. An unrestricted test run in an earlier project state with fewer metrics resulted in more than 6 hours of runtime. As one of the main goals of the test with plantUML is to check a more evolved code base to find potential instabilities in the analysis procedure, the muti-hour execution may not have generated more meaningful data than we already got. The data was sent to the code-service during the runtime and could be observed as a console output stream.

Multiple errors occurred during the execution, but none prevented successful completion. We want to focus on two types of errors that occurred during the execution. Figure 6.3 shows the first, more severe error. The JavaParser could not create an AST from the source file due to a *reserved keyword* used in the file. The analysis catches the error and handles it by skipping the file. Thus, we lost data for the entire file, but as the analysis depends on the AST, we can not provide an easy workaround.



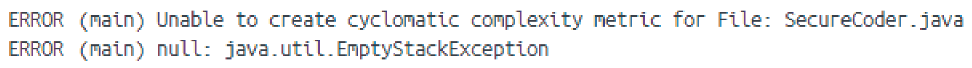
```
ERROR (main) Catched Javaparser exception, can't handle this, skipping file: CucaDiagramFileMakerJDot.java
ERROR (main) (line 382,col 50) '_' is a reserved keyword.
```

Figure 6.3. A severe error: The JavaParser could not generate an AST from the file.

The error in Figure 6.4 is thrown by the *CyclometicComplexityVistor*, indicating an *EmptyStackExpection*. It was found that a missing implementation for a language feature caused the error. Similar kinds of errors happened multiple times during the development

[5]https://github.com/ExplorViz/code-service/tree/jp-dev
[6]https://www.jetbrains.com/idea/

process as Java provides many language features that may not be well-known or often used. The key reason for this error is the expectation of the *CyclometicComplexityVistor* to work in the context of a method when encountering *control structures*. This is a false assumption, as control structures are also allowed in the context of `static` blocks inside classes. This error, fortunately, results only in a missing cyclomatic complexity metric but does not impede the analysis in any other way.



```
ERROR (main) Unable to create cyclomatic complexity metric for File: SecureCoder.java
ERROR (main) null: java.util.EmptyStackException
```

Figure 6.4. A metric calculation failed. This only results in a missing metric entry in the analysis data.

6.2.3 GitLab CI Integration

Integrating the service into the GitLab CI is the most convenient way to use the analysis service as it was designed to run on it. Required environment variables are already pre-set to variables exposed by the GitLab CI; thus, integrating the service is mostly interaction with *gitlab.com*'s user interface. We will show the integration by using the public repository *BusyDoingNothing*[7]. The repository contains a Java project with multiple Java files. Even though it looks like a runnable project, it is not meant to be run. Most classes serve the purpose of testing specific parts of the analysis, implementing language features, or are designed to validate metric calculations. This makes it a perfect candidate to use the project to test and verify the integration of the analysis service into different runners.

Integrating the analysis service to a GitLab-hosted project is only a matter of adding the `.gitlab-ci.yml` file to the project's top-level directory, with the content shown in Listing 6.3, to almost entirely configure the container. The usually mandatory environment variables *ANALYSIS_REMOTE_URL* and *ANALYSIS_BRANCH* get auto-populated by the GitLab CI, meaning, by default, the analysis runs against the current repository. As the current CI job provides the branch's name it runs on, all branches are automatically tracked and analyzed by default. Even if a new branch gets pushed, no changes to the configuration are needed.

```
stages:
  - analysis
analysis-job:
  stage: analysis
  image: 0xhexdec/code-analysis-test:2.2
  variables:
    ANALYSIS_FETCH_REMOTE: "true"
    ANALYSIS_SEND_TO_REMOTE: "true"
    ANALYSIS_CALCULATE_METRICS: "true"
    ANALYSIS_RESOLVE_WILDCARDS: "true"
```

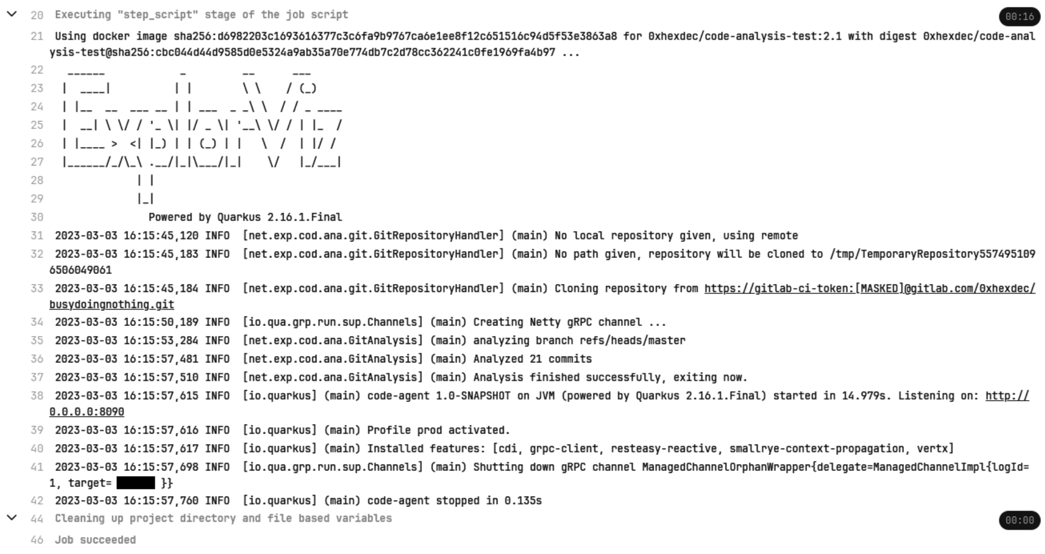
[7]https://gitlab.com/0xhexdec/busydoingnothing

```
  script:
    - echo "Analysis"
```

Listing 6.3. The basic `.gitlab-ci.yml` that is used for this showcase to run the analysis on the code contained in the repository the CI is attached to.

As one might have noticed already, the pipeline is set up so that a connection to a remote endpoint is used; thus, setting the IP and port is necessary. Although it might be possible to define these settings in the `.gitlab-ci.yml` as well, because the project is open source and public, personal data such as the IP and port would be visible to everyone. Under normal circumstances, this is not desirable. To hide private data, it is recommended to define *GRPC_HOST* and *GRPC_PORT* in GitLab as *private variables* found under **Settings->CI/CD->Variables**. The variables defined there are treated the same way as variables defined in the configuration file but are only visible to members.

Once new commits are pushed to the repository, the analysis gets executed as seen in Figure 6.5. Unlike the configuration used in Section 6.2.2, this container was configured to *fetch* the status prior to the execution. This will result in an accelerated execution time for subsequent calls.



```
Executing "step_script" stage of the job script                                                          00:16
Using docker image sha256:d6982203c1693616377c3c6fa9b9767ca6e1ee8f12c651516c94d5f53e3863a8 for 0xhexdec/code-analysis-test:2.1 with digest 0xhexdec/code-anal
ysis-test@sha256:cbc044d44d9585d0e5324a9ab35a70e774db7c2d78cc362241c0fe1969fa4b97 ...
  ______           _          __      ___
 |  ____|         | |         \ \    / (_)
 | |__  __  ___ __| | ___   __ \ \  / / _ ____
 |  __| \ \/ / '_ \| |/ _ \| '__\ \/ / | |_  /
 | |____ >  <| |_) | | (_) | |   \  /  | |/ /
 |______/_/\_\ .__/|_|\___/|_|    \/   |_/___|
             | |
             |_|
               Powered by Quarkus 2.16.1.Final
2023-03-03 16:15:45,120 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) No local repository given, using remote
2023-03-03 16:15:45,183 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) No path given, repository will be cloned to /tmp/TemporaryRepository557495109
6506049061
2023-03-03 16:15:45,184 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) Cloning repository from https://gitlab-ci-token:[MASKED]@gitlab.com/0xhexdec/
busydoingnothing.git
2023-03-03 16:15:50,189 INFO  [io.qua.grp.run.sup.Channels] (main) Creating Netty gRPC channel ...
2023-03-03 16:15:53,284 INFO  [net.exp.cod.ana.GitAnalysis] (main) analyzing branch refs/heads/master
2023-03-03 16:15:57,481 INFO  [net.exp.cod.ana.GitAnalysis] (main) Analyzed 21 commits
2023-03-03 16:15:57,510 INFO  [net.exp.cod.ana.GitAnalysis] (main) Analysis finished successfully, exiting now.
2023-03-03 16:15:57,615 INFO  [io.quarkus] (main) code-agent 1.0-SNAPSHOT on JVM (powered by Quarkus 2.16.1.Final) started in 14.979s. Listening on: http://
0.0.0.0:8090
2023-03-03 16:15:57,616 INFO  [io.quarkus] (main) Profile prod activated.
2023-03-03 16:15:57,617 INFO  [io.quarkus] (main) Installed features: [cdi, grpc-client, resteasy-reactive, smallrye-context-propagation, vertx]
2023-03-03 16:15:57,698 INFO  [io.qua.grp.run.sup.Channels] (main) Shutting down gRPC channel ManagedChannelOrphanWrapper{delegate=ManagedChannelImpl{logId=
1, target= [redacted] }}
2023-03-03 16:15:57,760 INFO  [io.quarkus] (main) code-agent stopped in 0.135s
Cleaning up project directory and file based variables                                                    00:00
Job succeeded
```

Figure 6.5. Output from a successful container execution using the GitLab CI.

6.2.4 GitHub Actions Integration

Only some projects are willing to run the CI as a standalone solution on a local machine or migrate to GitLab. Thus, we will run the analysis with *GitHub Actions* to show the portability of the created analysis service. The integration to GitHub is not baked-in into the analysis service and must be configured by hand.

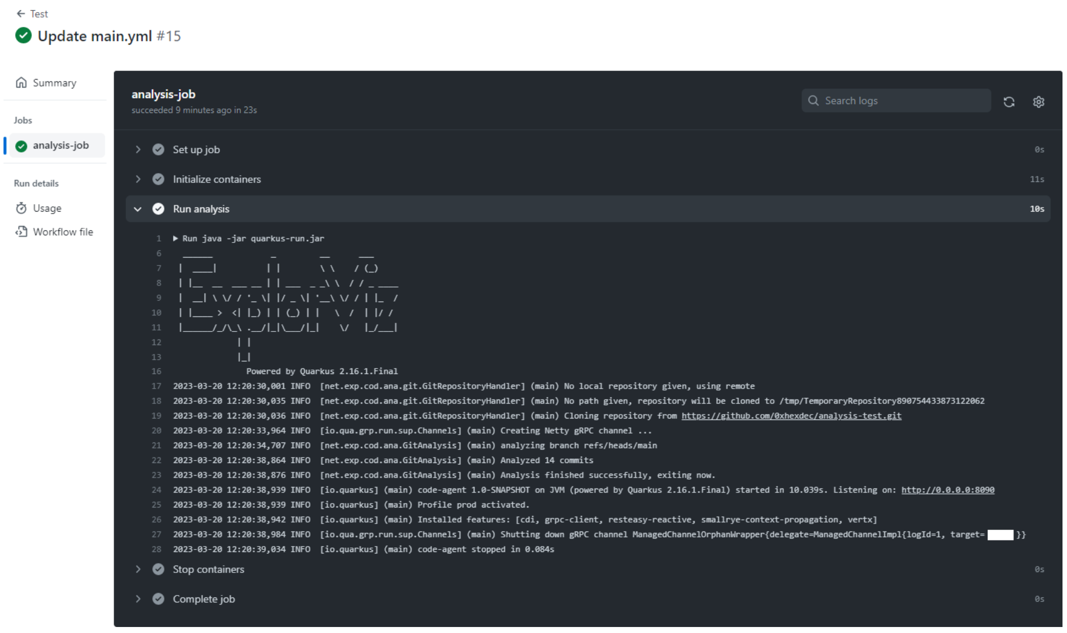
Similar to the above-described GitLab version, it is needed to create a Java project in the first place. The project has to be pushed to a GitHub repository. In this case, we used a public repository (reachable here[8]). The next step is to set up the *Action* by providing the `main.yml` shown in Listing 6.4 to the directory `.github/workflows/` within the project. Once pushed to the *main* branch, the analysis service starts to run but is missing the *GRPC_HOST* and *GRPC_PORT* environment variables. As mentioned in Section 6.2.3, these variables should be defined as secure project environment variables via the UI due to privacy concerns. In GitHub, this setting is found via **Settings->Security->Secrets and variables ->Actions**, select **Variables** and add them as *Repository variables*.

```
name: Analysis
on:
  push:
    branches: [ "main" ]
jobs:
  analysis-job:
    runs-on: ubuntu-latest
    container:
      image: 0xhexdec/code-analysis-test:2.2
      env:
        GRPC_HOST: ${{vars.GRPC_HOST}}
        GRPC:PORT: ${{vars.GRPC_PORT}}
        ANALYSIS_REMOTE_URL: https://github.com/${{github.repository}}.git
        ANALYSIS_BRANCH: ${{ github.ref_name }}
        ANALYSIS_FETCH_REMOTE: "true"
        ANALYSIS_SEND_TO_REMOTE: "true"
        ANALYSIS_CALCULATE_METRICS: "true"
        ANALYSIS_RESOLVE_WILDCARDS: "true"
    steps:
      - name: Run analysis
        working-directory: /home/jboss
        run: java -jar quarkus-run.jar
```

[8]https://github.com/0xhexdec/analysis-test

Listing 6.4. The basic `main.yml` used for *GitHub Actions* to run the analysis service on the current repository. Notice the more complex configuration compared to the GitLab integration file (Listing 6.3), the `quarkus-run.jar` has to be called manually to start the analysis as well as defining the working directory.

The Action is now configured such that the analysis will run for every push on the *main* branch. The GitHub Actions output can be seen in Figure 6.6



```
▶ Run java -jar quarkus-run.jar
  _____            _        __     __
 |  ___|          | |       \ \   / (_)
 | |__  __  ___ __ | | ___  _ _\ \ / / _ ____
 |  __| \ \/ / '_ \| |/ _ \| '__\ \/ / | |_  /
 | |___ >  <| |_) | | (_) | |   \  /  | |/ /
 |_____/_/\_\ .__/|_|\___/|_|    \/   |_/___|
            | |
            |_|
              Powered by Quarkus 2.16.1.Final
2023-03-20 12:20:30,001 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) No local repository given, using remote
2023-03-20 12:20:30,035 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) No path given, repository will be cloned to /tmp/TemporaryRepository890754433873122062
2023-03-20 12:20:30,036 INFO  [net.exp.cod.ana.git.GitRepositoryHandler] (main) Cloning repository from https://github.com/0xhexdec/analysis-test.git
2023-03-20 12:20:33,964 INFO  [io.qua.grp.run.sup.Channels] (main) Creating Netty gRPC channel ...
2023-03-20 12:20:34,707 INFO  [net.exp.cod.ana.GitAnalysis] (main) analyzing branch refs/heads/main
2023-03-20 12:20:38,864 INFO  [net.exp.cod.ana.GitAnalysis] (main) Analyzed 14 commits
2023-03-20 12:20:38,876 INFO  [net.exp.cod.ana.GitAnalysis] (main) Analysis finished successfully, exiting now.
2023-03-20 12:20:38,939 INFO  [io.quarkus] (main) code-agent 1.0-SNAPSHOT on JVM (powered by Quarkus 2.16.1.Final) started in 10.039s. Listening on: http://0.0.0.0:8090
2023-03-20 12:20:38,939 INFO  [io.quarkus] (main) Profile prod activated.
2023-03-20 12:20:38,942 INFO  [io.quarkus] (main) Installed features: [cdi, grpc-client, resteasy-reactive, smallrye-context-propagation, vertx]
2023-03-20 12:20:38,984 INFO  [io.qua.grp.run.sup.Channels] (main) Shutting down gRPC channel ManagedChannelOrphanWrapper{delegate=ManagedChannelImpl{logId=1, target=[redacted]}}
2023-03-20 12:20:39,034 INFO  [io.quarkus] (main) code-agent stopped in 0.084s
```

Figure 6.6. Output from a successful container execution using GitHub Actions.

6.3 Results and Discussions

Now that we showed different scenarios of execution for the analysis, we will summarize the results and discuss some limitations of the project as well as give an outlook of potential problems for future enhancements.

6.3.1 Findings

As shown above, we successfully ran the analysis on different projects without significant problems. We encountered no crashes of the service even though we faced some serious errors during the analysis stages. The analysis was always able to recover fully. Further, we have shown that the service implemented as a Docker container allows easy migration to different runner tools. Even though specifically prepared to run on the GitLab CI, we have shown that the runner can quickly be adjusted by only changing some environment variables to be able to run with GitHub Actions. Integration into other runner tools should also be possible.

Further, it was noticeable that the structural analysis was more stable and reliable than the metric calculation, despite being fed with source code that results in compiler errors. The design of encapsulated visitors for each metric worked out, not only from a maintainability perspective but also from the overall robustness of the analysis, as a failure of computation from one metric does not affect the outcome of another.

A chance find was the missing handling of interrupted connections. While the analysis does not start if the code-service is not reachable on startup, if the connection is interrupted, the analysis continues to send its data to the defined endpoint until it finishes. This may be due to the design decision of setting the reply message of FileData and StateData packets to *emtpy*.

6.3.2 Performance

Even though the performance of *short and clever code* was never a priority, the service seems to be *sufficient* fast for the application it is designed for, as seen in Section 6.2.1, the PetClinic's main branch was analyzed untruncated within approximately 30 seconds for an entire project. This kind of complete project analysis is not the main task of the analysis, so analyzing a dozen commits at a time is. That was roughly the number of commits we encountered in Section 6.2.3 and Section 6.2.4, where the cloning of the project took longer than the actual analysis. However, the entire pipeline runtime was still under a minute which can be considered fast enough to be somewhat futureproof.

6.3.3 Correctness of Metrics

During the development and for some test executions, resulting metric values were sporadically checked against the same metrics calculated by *Checkstyle*[9] and *PMD*[10] for comparison (provided the metric was calculated by the tool, which was not the case for LCOM4). Some tests are also implemented for the metrics, but as these test cases are artificially crafted to provide cases to check against, they may not be a good substitution for actual running code.

While some metrics are easier to implement or apply to a language, others are not. LOC, for example, is relatively easy to comprehend, but the value highly depends on the actual implementation. While this might not be a problem for *lines of code* as the value's

[9]https://checkstyle.org/
[10]https://pmd.github.io/

order of magnitude is more expressive than the value itself, computing the cyclomatic complexity differently could result in a deviated outcome. As the original creator discussed the metric used for Fortran, an evolved language such as Java consists of different features that could be considered in the calculation [15]. The original implementation does not consider language features such as *ternary operators* or *foreach* and *map* calls on `streams`.

6.3.4 Threats to Validity

Even though we performed multiple test executions in Section 6.2 and the analysis service performed as expected, we can not guarantee the analysis is as stable and robust as assumed as we may encounter some more missed language features that result in a crashed service.

As the data model for the generated analysis data is straightforward, the sheer quantity of generated data is not. Thus, the data could only be validated randomly. Major problems would stand out once they are visualized, but since the visualization is not in the scope of this thesis, validating the data is not feasible. As it is not realizable to automatically validate anything, as the data is meant for code *comprehension*, which could only be verified by surveying developers, we cannot give a qualitative statement on the meaningfulness of the collected data.

6.4 Usefulness

As mentioned above, the current state as a *runnable and stable service* makes it unproblematic to include it in an existing CI pipeline. As the generated data needs to be processed and visualized to be meaningful, the service should be described as *technically executable* but not finished to a degree where the collected data is helpful for a developer in the context of program comprehension. It should be considered as the basis for further data collection and to implement the counterpart at ExplorViz's end. The data can be used to develop the visualization against and the thoughts, suggestions, and findings from this thesis can be used as a starting point for improvements as a solid, robust, and extendable service architecture has been created.

Chapter 7

Related Work

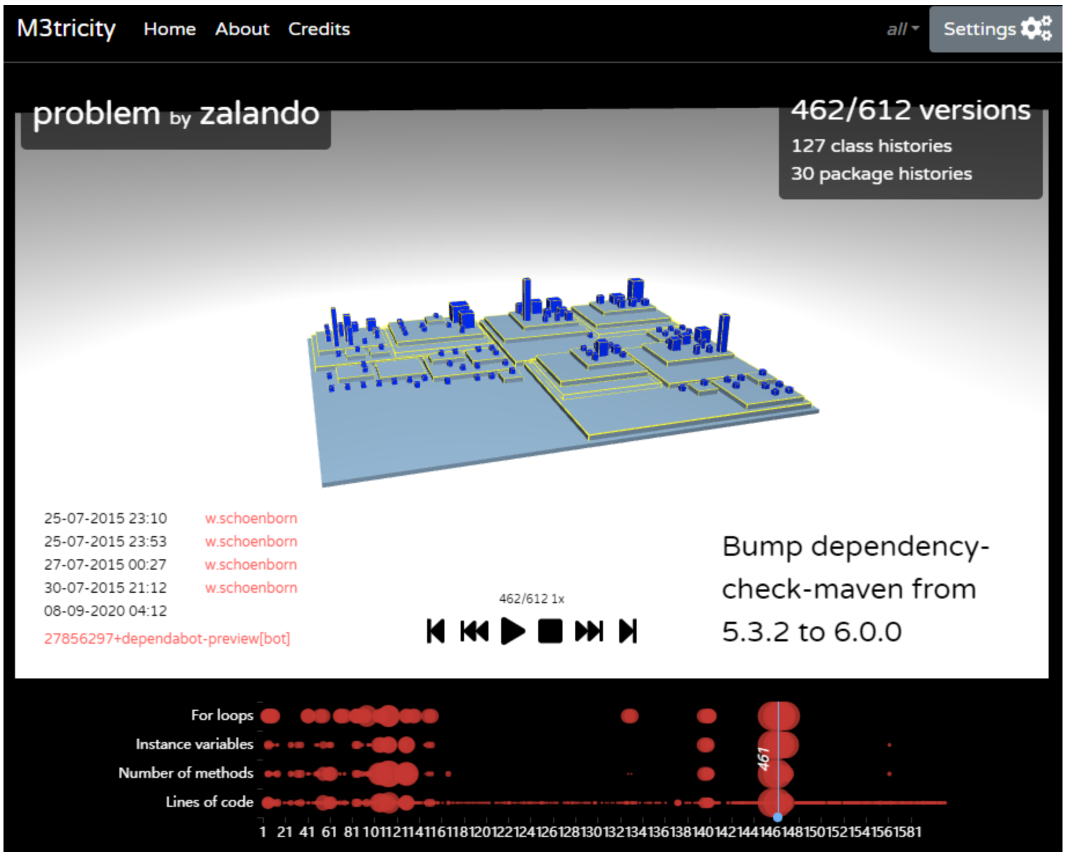
The main focus of this thesis is the implementation of static analysis with Git analyzing capabilities for visualization. As the service is provided as a Docker container to be integrated into a build pipeline, we will focus on related work that worked on similar topics.

7.1 M3triCity

M3triCity[1] is a tool for repository analysis and visualizing evolving software and data cities. It is known for its city-based visualization approach already present at its ancestor, *CodeCity*. The project is split into two parts: the visualization, seen in Figure 7.1, and the backend, responsible for repository handling and source code analysis [1].

This approach is pretty similar to the one provided in this thesis. Frontend and backend are independent, and the analysis is done once to provide metrics to an internal database the frontend retrieves the data from. As seen in Figure 7.1, the calculated metrics consist of *lines of code*, *number of for loops*, *instance variables*, and *number of methods*—the changes in the metrics as highlighted as different sized circles per Git-commit. In contrast to this approach, we try to implement metrics that are more meaningful on their own. Further, we do not model the backend and frontend as a interlinked project but seperate both by implementing the analysis as an external runner service which can be used and integrated into other projects as well.

[1]https://metricity.si.usi.ch/#/



25-07-2015 23:10 w.schoenborn
25-07-2015 23:53 w.schoenborn
27-07-2015 00:27 w.schoenborn
30-07-2015 21:12 w.schoenborn
08-09-2020 04:12 27856297+dependabot-preview[bot]

Bump dependency-check-maven from 5.3.2 to 6.0.0

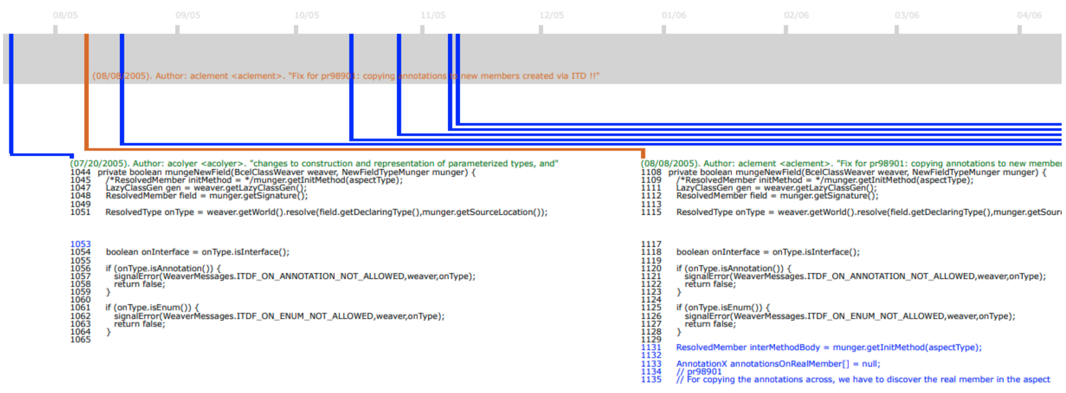
Figure 7.1. *problem* by *zalando* visualized in M3triCity.

7.2 Chronos

Chronos is made to discover and explore historical changes in source code. They provide functions to compare multiple software versions simultaneously, unlike the standard Git implementation, which only provides the *diff* command for two commits. The tool is implemented as an Eclipse[2] plugin and supports CSV, Subversion, and Git as revision-control systems. They used Chronos to reverse engineer and identify design rationale [17]. As seen in Figure 7.2, Chronos focuses on chronical code changes. The blue and orange vertical lines on the gray stripe indicate software changes in a timeline, while the code viewer below shows the source code changes between the two revisions. Code snippets can be marked to search for software changes over the course of the development. While we

[2]https://www.eclipse.org/ide/

also track file revision changes and will provide the data to compare different versions of the software, we do not provide a source code visualization as the focus is more towards a conceptual comprehension of the software and not the syntax level changes represented in Chronos.



(08/08/2005). Author: aclement <aclement>. "Fix for pr98901: copying annotations to new members created via ITD !!"

(07/20/2005). Author: acolyer <acolyer>. "changes to construction and representation of parameterized types, and"

```
private boolean mungeNewField(BcelClassWeaver weaver, NewFieldTypeMunger munger) {
   /*ResolvedMember initMethod = */munger.getInitMethod(aspectType);
   LazyClassGen gen = weaver.getLazyClassGen();
   ResolvedMember field = munger.getSignature();

   ResolvedType onType = weaver.getWorld().resolve(field.getDeclaringType(),munger.getSourceLocation());


   boolean onInterface = onType.isInterface();

   if (onType.isAnnotation()) {
      signalError(WeaverMessages.ITDF_ON_ANNOTATION_NOT_ALLOWED,weaver,onType);
      return false;
   }

   if (onType.isEnum()) {
      signalError(WeaverMessages.ITDF_ON_ENUM_NOT_ALLOWED,weaver,onType);
      return false;
   }

```

(08/08/2005). Author: aclement <aclement>. "Fix for pr98901: copying annotations to new membe

```
private boolean mungeNewField(BcelClassWeaver weaver, NewFieldTypeMunger munger) {
   /*ResolvedMember initMethod = */munger.getInitMethod(aspectType);
   LazyClassGen gen = weaver.getLazyClassGen();
   ResolvedMember field = munger.getSignature();

   ResolvedType onType = weaver.getWorld().resolve(field.getDeclaringType(),munger.getSour


   boolean onInterface = onType.isInterface();

   if (onType.isAnnotation()) {
      signalError(WeaverMessages.ITDF_ON_ANNOTATION_NOT_ALLOWED,weaver,onType);
      return false;
   }

   if (onType.isEnum()) {
      signalError(WeaverMessages.ITDF_ON_ENUM_NOT_ALLOWED,weaver,onType);
      return false;
   }

   ResolvedMember interMethodBody = munger.getInitMethod(aspectType);

   AnnotationX annotationsOnRealMember[] = null;
   // pr98901
   // For copying the annotations across, we have to discover the real member in the aspect
```

Figure 7.2. Chronos

7.3 Static Analysis in CI

Bolduc showed the benefits of integrating the static source code analysis tool *Klocwork* into the continuous integration pipeline. The study showed the benefits of having the latest analysis data ready to consult when a new version was released. It was concluded that the benefit of having access to the most recent analysis data outweighed the potential increase in calculation time due to the frequent analysis execution [2]. Even though *Bolduc* mentioned the increased computation time, our approach does not suffer because we only analyze files that have changed. The main improvement is that the data is always up-to-date and the analysis is triggered automatically.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we designed and implemented a static source code analysis service to generate pre-processed data for visualizations. We implemented complete Git repository handling of arbitrary Java projects and crafted a stable and robust Java code analysis. It analyzes structural data of the underlying Java files and allows for calculating advanced source code metrics. We designed the metrics calculation as an encapsulated add-on to allow an easy and robust extension of the current analysis. We provided functionalities to abstract the AST walking for metric calculations to keep the expense of developing new metrics small. The migration to a self-contained Docker container resulted in an easy-to-integrate service into new and already set-up CI pipelines. The conducted evaluation of the service's output data showed the missing usefulness of the data as it is not designed to be human-readable in a form that would grant access to any information. Even though the service ran fine for all scenarios, the AST generation and metric calculations showed potential for improvement as some language features were incorrectly handled. In conclusion, the created analysis service provides a robust and easy-to-use framework to extend on. It has the potential to be integrated into ExplorViz's ecosystem as its future static analysis component.

8.2 Future Work

As the analysis service implemented in this thesis will only be a component of *ExplorViz's static analysis*, the data visualization is the most critical work to do. Therefore, the *code-service* needs to be integrated into ExplorViz and write the data to the running Kafka instance. Further, ExplorViz's current visualization needs to be extended to handle the new static data as well. After doing so, the generated structural data and metrics can be evaluated by usefulness for the developer to understand the software.

Improvements for the analysis would consist of finding and resolving the occurred errors seen in Section 6.2.2, as well as adding some missing structural data to the current collection step. During the testing phase, it was found that *annotations* were missing completely. The current state misses the ability to list dependencies on a method level, where every method has a list of *external* calls attached to provide call graphs at the visualization stage. Further, as only some metrics are implemented as a proof of concept, more metrics should be added in future improvements. Some good source metrics to implement would be *nPath-complexity*, *acPath-complexity*, or *cognitive-complexity*.

While the current approach of *cloning* the repository each time the analysis runs was the most portable implementation as it always works no matter the environment (requiring internet access, of course), using CI built-in tools or already existing functionality to facilitate and speed up the clonen process would improve the runtime of the service and made for a cleaner solution.

Bibliography

[1] Susanna Ardigò et al. "M3tricity: visualizing evolving software & data cities". In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pages 130–133. DOI: 10.1145/3510454.3516831. URL: https://doi.org/10.1145/3510454.3516831 (cited on page 51).

[2] Claude Bolduc. "Lessons learned: using a static analysis tool within a continuous integration system". In: *IEEE International Symposium on Software Reliability Engineering Workshops ISSREW (2016)*, pages 37–40. DOI: 10.1109/ISSREW.2016.48 (cited on page 53).

[3] Ruven Brooks. "Towards a theory of the comprehension of computer programs". In: *International Journal of Man-Machine Studies* 18.6 (1983), pages 543–554 (cited on page 5).

[4] Wojciech Cellary and Sergiusz Strykowski. "E-government based on cloud computing and service-oriented architecture". In: *Proceedings of the 3rd International Conference on Theory and Practice of Electronic Governance*. ICEGOV '09. Bogota, Colombia: Association for Computing Machinery, 2009, pages 5–10 (cited on page 23).

[5] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007 (cited on page 6).

[6] Manuel Egele et al. "A survey on automated dynamic malware-analysis techniques and tools". In: *ACM Comput. Surv.* 44.2 (2008) (cited on page 6).

[7] S. Elbaum and John Munson. "Code churn: a measure for estimating the impact of code change". In: *Conference on Software Maintenance* (Sept. 2000) (cited on page 23).

[8] Laune C. Harris and Barton P. Miller. "Practical analysis of stripped binary code". In: *SIGARCH Comput. Archit. News* 33.5 (Dec. 2005), pages 63–68 (cited on page 6).

[9] Wilhelm Hasselbring, Alexander Krause, and M. Bader. "Collaborative software visualization for program comprehension". In: *Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 22)*. IEEE. 2022 (cited on page 10).

[10] Wilhelm Hasselbring, Alexander Krause, and Christian Zirkelbach. "Explorviz: research on software visualization, comprehension and collaboration". In: *Software Impacts* 6 (2020), page 100034 (cited on page 10).

[11] Martin Hitz and Behzad Montazeri. "Measuring coupling and cohesion in object-oriented systems". In: Oct. 1995 (cited on page 37).

[12] *Introduction to grpc*. URL: https://grpc.io/docs/what-is-grpc/introduction/ (visited on 11/22/2022) (cited on page 8).

[13] *Jgit/user guide*. URL: https://wiki.eclipse.org/JGit/User_Guide (visited on 11/22/2022) (cited on page 7).

[14] Alexander Krause-Glau, Malte Hansen, and Wilhelm Hasselbring. "Collaborative program comprehension via software visualization in extended reality". In: *Information and Software Technology* 151 (2022), page 107007 (cited on page 10).

[15] T.J. McCabe. "A complexity measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pages 308–320. DOI: 10.1109/TSE.1976.233837 (cited on pages 35, 49).

[16] Václav Rajlich and Norman Wilde. "The role of concepts in program comprehension". In: *Proceedings of the 10th International Workshop on Program Comprehension*. IEEE. 2002, pages 271–278 (cited on page 5).

[17] Francisco Servant and James A. Jones. "Chronos: visualizing slices of source-code history". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013, pages 1–4. DOI: 10.1109/VISSOFT.2013.6650547 (cited on page 52).

[18] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices". In: *IEEE Access* 5 (2017), pages 3909–3943. DOI: 10.1109/ACCESS.2017.2685629 (cited on page 6).

[19] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *Javaparser: visited. Analyse, transform and generate your java code base*. Leanpub, 2019 (cited on page 7).

[20] Xin Xia et al. "Measuring program comprehension: a large-scale field study with professionals". In: *IEEE Transactions on Software Engineering* 44.10 (2018), pages 951–976 (cited on page 1).

[21] Fiorella Zampetti et al. "Ci/cd pipelines evolution and restructuring: a qualitative and quantitative study". In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2021)*. 2021, pages 471–482 (cited on page 6).

Appendix

Environment Variables

The service can be adjusted to work as needed by setting the environment variables. To see what the analysis is capable of, we list each available environment variable and see what it does:

`GRPC_HOST` This is the host IP of the *code-service*. The *code-agent* will send all requests to this IP. It must only be specified if **`ANALYSIS_FETCH_REMOTE`** or **`ANALYSIS_SEND_TO_REMOTE`** are set to true.

`GRPC_PORT` This defines the port gRPC uses for the connection; the default value is 9000. The *code-service* expects the data on this port by default.

`ANALYSIS_LOCAL_PATH` If a local repository is used, the storage directory can be specified with this environment variable. If this is set, it always precedes anything configured for the remote repository. The local repository is used *as-is*, and no updates will be pulled.

`ANALYSIS_REMOTE_URL` This variable is used to set the remote repository's URL. It should be given as an *HTTP* or *HTTPS* URL, *SSH* is not supported by the service and will result in a warning. The service will try to convert it to an *HTTP* URL and then clone the repository with the new URL as a fallback. This variable will be auto-populated when running in a GitLab pipeline by the current repository.

`ANALYSIS_STORAGE_PATH` The repository will be cloned to this directory. If this variable is not present, a *temp directory* will be created, and the repository will be cloned to that.

`ANALYSIS_BRANCH` The branch to be analyzed is given as a short name (without /heads/ref/). Defining the branch that will be analyzed is mandatory. The only exception is inside the GitLab CI, where it gets auto-populated with the current branch the CI is currently running on.

`ANALYSIS_USERNAME` The username is optional and only used to be able to access private repositories.

`ANALYSIS_PASSWORD` The password is optional and only used to be able to access private repositories.

`ANALYSIS_FETCH_REMOTE` This variable expects a boolean; if set to true, the *code-agent* will request the current state (hash of the last commit that was analyzed before) from the *code-service*. Otherwise, the entire branch will be analyzed. (Depending on the value of **`ANALYSIS_START_COMMIT`** and **`ANALYSIS_END_COMMIT`**) The default value is *false*.

`ANALYSIS_SEND_TO_REMOTE` If this is set to true, the analysis data will be sent to the remote endpoint; if set to false, the data will be stored locally as JSON files in a directory stated during the start of the service. The default value is *false*.

`ANALYSIS_SOURCE_DIRS` The source directories to consider for the type resolving are defined here. The directories are defined relative to the repository's base directory. See the *readme* of the project for an in-depth explanation of how to use the *search expressions* supported by this variable. The default value considers all source files found in `/src/main/java` and `/src/test/java` no matter the depth at which the folder structure is found.

`ANALYSIS_RESTRICT_DIRS` Only the directories defined in this variable are analyzed. This can be used to only analyze the source folder and omit the test folder. See the *readme* of the project for an in-depth explanation of how to use the *search expressions* supported by this variable. The default value considers all source files found in `/src/main/java` no matter the depth of the folder structure found.

`ANALYSIS_START_COMMIT` This variable is used to define the commit the analysis starts with. Only commits after this commit is analyzed. This value is ignored if ANALYSIS_FETCH_REMOTE is defined. Expects the complete SHA-1 of a commit. The default is empty, meaning to start with the first commit of the branch.

`ANALYSIS_END_COMMIT` This variable is used to define the last commit. Every commit behind this commit is omitted. Expects the complete SHA-1 of a commit. The default is empty, meaning it ends with the last commit of the branch.

`ANALYSIS_CALCULATE_METRICS` If only the structural data is needed, the metric calculating can be deactivated by setting this to false. The default value is *true*.

`ANALYSIS_RESOLVE_WILDCARDS` The default handling of wildcard imports is to mark them as *unresolvable*; if the attempt described in Section 4.1.2 should be tried, this can be set to true. The default value is *false*.

`EXPLORVIZ_LANDSCAPE_TOKEN` This is a customizable value that gets send together with the *StateRequest* to identify the branch data. This value is not used in the analysis and only forwarded to the *code-service*. The default value is *default-token*.

`EXPLORVIZ_LANDSCAPE_SECRET` This is a customizable value that gets send together with the *StateRequest* to identify the branch data. This value is not used in the analysis and only forwarded to the *code-service*. The default value is *default-secret*.