

BENG INDIVIDUAL PROJECT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Mapping Your Codebase

Author:

Pawel Kroll

Supervisor:

Dr Robert Chatley

June 20, 2022

Abstract

In this project, we present Metrico, a tool that allows users to explore and investigate code metrics and their relation to risk identification and analysis. Metrico obtains the data about the codebase from GitHub repositories, stores the metrics in the database and presents visualisations of code metrics to the user.

Metrico contributes to risk management in software engineering by providing a platform to explore unique visualisations of code metrics and investigate their relations to risk identification. Risk management in software engineering aims to identify and minimise potential risks in a project. As risks emerge from changes to the codebase, we can use code metrics that represent these changes to help identify the potential problems in a project.

In evaluation, we validate that Metrico can create meaningful visualisations of code metrics. We discuss the results of a small study conducted among software developers that used our tool to complete a set of tasks related to risk management. On top of that, we measure the performance of our tool to validate that it will not disrupt the user's workflow.

Acknowledgements

I would like to thank my supervisor, Dr Robert Chatley, for his continuous support and guidance throughout the year. His enthusiasm during our regular meetings have been a huge motivation to work on this project. I would also like to thank Dr Holger Pirk for his constructive feedback during the interim report.

I am grateful to my friends who supported and motivated me throughout these difficult three years.

I would like to thank my parents, who sacrificed a lot to make it possible for me to study in the United Kingdom. Their continuous support have been very important to me during these three years and I would not be able to make it without them.

Contents

1	Introduction	6
1.1	Motivation	6
1.2	Objectives	6
1.3	Contributions	7
2	Background	9
2.1	Risk management in software development	9
2.1.1	Defining Risk	9
2.1.2	Risk identification and analysis	9
2.2	Types of risks	10
2.2.1	Technical debt	10
2.2.2	Technologies used in a project	11
2.2.3	Code duplication	11
2.3	Code Metrics	11
2.3.1	Code Churn	11
2.3.2	Contributors metrics	12
2.3.3	Cyclomatic complexity	12
2.3.4	Class coupling	12
2.3.5	Defect density	12
2.3.6	Test coverage	12
2.4	Visualising codebase	13
2.4.1	Gource	13
2.4.2	CodeCity	14
2.4.3	GoCity	14
2.4.4	GitHub Next	14
2.5	Mining Git repositories	16
2.5.1	PyDriller	16
2.5.2	GitHub RESTAPI	17
3	Design and implementation	18
3.1	Architecture overview	18
3.1.1	Technologies used	19
3.2	Getting version control data	21
3.3	Metrics	21
3.3.1	Picking metrics	21
3.3.2	Calculation of metrics	22

3.4	Database models	23
3.5	Serving the data to frontend	24
3.6	User Flow	25
3.7	Visualisation	26
3.7.1	Considered visualisations	26
3.7.2	Features of visualisation	28
3.8	Extending Metrico	28
4	Results and evaluation	30
4.1	General evaluation	30
4.2	Survey	31
4.2.1	First group - survey results	32
4.2.2	Second group - survey results	36
4.2.3	Comparison	40
4.2.4	General questions	40
4.3	Performance	43
4.3.1	Initialisation	43
4.3.2	Metrics call	44
4.3.3	Summary	44
5	Ethical discussion	45
6	Conclusion	46
6.1	Contributions and findings	46
6.2	Future work	47
6.2.1	Solving GitHub API dependency	47
6.2.2	Pull requests	47
6.2.3	More visualisations	47
6.2.4	Add filters	47
6.2.5	Further studies	48
A	Metrico pages screenshots	49
B	Complete results of a survey	50

List of Figures

2.1	Final visualisation of a simple codebase using Gource (29).	13
2.2	visualisation of a codebase using CodeCity (28).	14
2.3	visualisation of a codebase using GoCity (30).	15
2.4	visualisation of a codebase created by GitHub Next visualisation tool (35).	15
3.1	High level architecture diagram of Metrico	18
3.2	Metrico's database models and relationships between them.	23
3.3	Example of a visualisation using bubble chart	27
3.4	Example of a visualisation using clustered bubble chart	27
4.1	visualisation generated using Added lines count and Code churn	30
4.2	visualisation generated in different time period	31
4.3	visualisation generated using Code churn and Pull requests per file	31
4.4	Answers of developers from the first group for task 1	33
4.5	Answers of developers from the first group for task 2	34
4.6	Answers of developers from the first group for task 3	35
4.7	Answers of developers from the first group for task 4	35
4.8	Answers of developers from the second group for task 1	36
4.9	Answers of developers from the second group for task 2	37
4.10	Answers of developers from the second group for task 3	38
4.11	Answers of developers from the second group for task 4	39
4.12	Which metrics have developers found useful	41
4.13	Most useful metrics according to the developers	41
4.14	Least useful metrics according to the developers	42
4.15	"Usefulness" score given by developers	42
A.1	Starting page of Metrico	49
A.2	Help page of Metrico	49

List of Tables

4.1	Metrics used by developers from the first group in task 1	32
4.2	Metrics used by developers from the first group in task 2	33
4.3	Metrics used by developers from the first group in task 3	34
4.4	Metrics used by developers from the first group in task 4	36
4.5	Metrics used by developers from the second group in task 1	37
4.6	Metrics used by developers from the second group in task 2	37
4.7	Metrics used by developers from the second group in task 3	38
4.8	Metrics used by developers from the second group in task 4	39
4.9	Results of a twbs/bootstrap repository	43
4.10	Results of a MaturaIT/maturait_vue repository	43

Chapter 1

Introduction

1.1 Motivation

Within Google, there is a saying that “Software engineering is programming integrated over time.”(36). It signifies the importance of the time dimension in the development process, as software grows and evolves constantly. It indicates that the whole software development process can be thought of as a collection of programming changes made to the specific codebase over time. However, as the codebase changes, some problems may arise.

In software engineering, risks are potential problems that may endanger a project in the future. For example, sometimes, a software engineer may decide to write code that follows bad practices and design in hopes of saving some time. This decision might harm the project when this code must be refactored or rewritten because it can not sustain new requirements or extensions.

Accurate risk identification is a crucial part of a successful risk management process. By helping programmers identify risks in a given codebase correctly, we can provide software developers and project managers with an important tool to make appropriate decisions about the future of a given code.

Code metrics are measurements that can give developers much information and insight about the state of a codebase. They can also represent changes made to the codebase. Therefore, a new framework enabling researchers to investigate code metrics and their visualisations could heavily contribute to finding new and unique approaches to risk management in software development.

1.2 Objectives

This project aims to create software that researchers and software engineers can use to create unique visualisations of code metrics. By creating such visualisations, researchers can then study the relationship between the areas of interest in the visualisation and emerging risks in a codebase. The goal is to facilitate a research tool

that can be used in future investigative work about code metrics and how they can predict and minimise risks in a software engineering process. To achieve this, our project will focus on three key objectives.

1. **Create a platform to research the usage of code metrics for risk management in software projects.** By developing our tool, we want to enable researchers to test specific code metrics' usefulness. The goal is to provide researchers with a framework that will make it easy and intuitive to calculate code metrics from codebases.
2. **Develop a tool that enables the investigation of code metrics through visualisations.** Ultimately, our project aims to develop software that will create meaningful visualisations of codebases. To achieve this goal, this project will focus on producing visual maps of code metrics that can be further experimented with by researchers and software developers.
3. **Be able to analyse software repositories in a reasonable amount of time.** By reasonable time we mean a time in which the tool can be run and does not impact the workflow of software developers or researchers.

By combining these three objectives, we aim to produce a comprehensive and valuable tool that can contribute to improving and advancing risk management in software engineering.

1.3 Contributions

1. This project introduces *Metrico*, a tool that collects data from GitHub about repositories, calculates metrics and creates meaningful visualisations. It allows users to pick the metrics they want to see and enables filtering across periods of time such as weeks or months. As a result, the tool can produce several visualisations representing metrics showing changes in a repository.
2. The tool stores metrics in a local database after pre-processing and initializing any new repository. That makes recalculating and creating new visualisations on an already initialised repository extremely fast (see [4.3](#)).
3. In a small study, several participants were presented with a number of tasks to perform on a visualisation. The study showed how they interact with a tool and what metrics they use to find areas of interest in a codebase. The participants also ranked the metrics in terms of usefulness. The difference between informed and uninformed repository users has been explored during the study. (described in [4.2](#)).

4. Metrico is open source and can be extended to support more metrics and data from repository.

Chapter 2

Background

2.1 Risk management in software development

2.1.1 Defining Risk

By the definition included in PMBOK (1) "risk is an uncertain event or condition that can have a positive or a negative effect on a project". Every risk comes with a possibility of unwanted behavior which may endanger our project. To prevent it we have to identify them before they affect our codebase and manage them accordingly.

2.1.2 Risk identification and analysis

Efficient risk identification is a crucial part of successful risk management in the software development process. There have been many tools developed that make identifying risks in code easier.

- **SonarQube** (12) is a tool that analyzes a codebase and detects code smells, vulnerabilities, bugs, and technical debt. It is widely used in the industry and has a number of features that can be beneficial to software developers in day-to-day work. It works with 29 programming languages and can be integrated into CI/CD pipeline.
- **Designite** (14) is an architecture and design analysis tool. It leverages tree-maps and charts to visualise calculated code smells, metrics and dependencies. Designite is primarily used for a codebase written in C# and Java.
- **DV8** (13) is a suite of tools that helps software developers in identifying risks. It focuses on architectural analysis, technical debt, and software modularity (13). It also measures how maintainable a source code is.
- **Structure101** (15) is a suite of tools that focus on visualising and modeling a codebase into a hierarchical and structural diagram. It is great at identifying cyclic dependencies and code coupling.

2.2 Types of risks

In this section, we will discuss background on some of the risks that can be derived from the git repository and have the potential to be useful in this project.

2.2.1 Technical debt

Technical debt is a concept described as a "phenomenon that impacts software projects and makes them difficult to manage"(3). Recent studies found that around 25% of software engineering effort is wasted on managing Technical Debt(4). In short, technical debt represents the shortcuts made in a software development process that will make future maintenance and development more costly. It can be described as a programmer "acquiring debt" when implementing a quick but not proper solution which they will have to "pay back" later in a development process when they need to interact with that piece of code. Moreover, Technical debt can be classified in two ways according to McConnell (5). It can be either:

- **intentional** - occurs when an organization makes a conscious decision to optimize for the present rather than for the future. It is often time pressure that makes it necessary for organizations to sacrifice the quality of a product knowingly.
- **unintentional** - occurs when it is acquired unknowingly. It can happen when an implementation turns error-prone or becomes harder to maintain than anticipated. Moreover, it is not always a case that an unintentional technical debt results from bad programming inside an organization. Often it may be the case that an organization will inherit a code that already has undocumented technical debt.

There has been extensive research focusing on detecting technical debt. A number of tools have been released, both commercial and research-oriented, that focus on identifying technical debt. Frameworks such as SonarQube (12), DV8 (13) or Structure101 (15) employ different rules and definitions to detect technical debt from source code. For example, Structure101 identifies possible technical debt by simple file size, excessive complexity, and a number of edges in dependency graph (6). At the same time, SonarQube has hundreds of rules to identify code smells, bugs, and vulnerabilities (6). Such different definitions of technical debt cause these tools to behave completely different from each other and produce unrelated results (6).

The more reliable way to process technical debt is by focusing on intentional debt. The self-admitted technical debt is identified by a software developer who is acquiring it. It can be indicated in the code comments, or issue tracker (7; 8). There are different ways in which we detect the technical debt in the issue tracker. We can do it through machine learning methods that take data obtained from issues into account (8). The other approach is to derive the self-admitted technical debt from the tags of an issue which is much easier and straightforward (7).

2.2.2 Technologies used in a project

There are a number of risks that can come from the types and number of technologies used in a project. For example, a high number of technologies used in a codebase may pose a risk regarding the range of knowledge needed to maintain the source code. The more varied an organization's project in terms of programming languages is, the more experts and software developers need to be hired. This increases costs and makes projects high maintenance. Recent studies have found that an average project utilizes five programming languages (9).

It is also common for projects to use outdated libraries and frameworks. A recent study (10) has shown that almost 37% of websites evaluated during the research, depend on the version of libraries with a known vulnerability. This has been found to cause a 'lag' in tested websites.

2.2.3 Code duplication

Code duplication is an effect of software developers copying and pasting parts of code back into the source code, creating a "code clone". Programmers usually do it to save time or simply because they do not care about the quality of produced code. However, this approach is very shortsighted as duplicated code is harder to maintain and can affect the quality and correctness of produced software. Moreover, if we copy a part of code with already existing risk, we may further endanger our project.

Studies have found that reducing code duplicates positively impacts code quality metrics (11). By refactoring code clones, researchers were able to decrease LOC and improve code coupling, stability of the system, complexity, and cohesion metrics.

2.3 Code Metrics

Several code metrics can be used in our project. In this section, we will describe some non-trivial metrics and look at what risks can be derived from them.

2.3.1 Code Churn

Code Churn helps to measure a change to the examined part of code in the project. There is a number of different code churn metrics that can be calculated for a given codebase. The absolute measures of code churn values are the most basic metrics and include: lines of code (LOC) added, LOC removed, or files modified. On the other hand, there are relative code churn measures that are weighted by other values such as LOC or file count (17). Both of these types of code churn metrics can be used to predict system defects and faults. However, studies have shown that relative code churn values are far more accurate than absolute code churn measures (17; 18).

2.3.2 Contributors metrics

Several metrics can be mined from source code contributors' data. The most basic one is contributors count, which represents the number of contributors that made a commit to the specific file. We can also track "contributors experience," a metric that measures the percentage of lines in a given file authored by a specific contributor.

2.3.3 Cyclomatic complexity

Cyclomatic complexity (CC) is a metric proposed by Tom McCabe in 1976 (19). It can be described as several linearly independent control flow paths. Therefore, the higher a cyclomatic complexity of a given piece of code, the higher the number of test cases needed. Research has also shown that often high CC corresponds with high maintenance cost and defect density (20).

CC is still popular and widely used in industry, mainly because it is easy to understand (20). However, a lot of research has doubted the effectiveness and validity of this code metric (21; 22). Moreover, Herraiz and Hassan argued that this code metric correlates with LOC; hence it is not needed (23). Despite this criticism, exploring the usage of cyclomatic complexity in our tool may be beneficial.

2.3.4 Class coupling

Class coupling (or Coupling Between Objects (CBO) (24)) indicates how many other classes are used in a given class. Usually, sound software development principles imply low-class coupling. Therefore we could postulate that the high value of the class coupling metric can indicate risk in a codebase. Research shows that the average limit on class coupling should be around nine (25). It means every class with a class coupling metric higher than nine should be considered a vulnerability and a bad practice. A high-class coupling value could also indicate a technical debt. This limit will not be optimal for all organizations, so it would be wise to make this value editable.

2.3.5 Defect density

Defect density is calculated as the number of defects divided by a unit size of code (16). It is a measure that helps to determine the effectiveness of software processes. It is often used as a metric to identify components that should be candidates for refactoring and further testing. Defect density can be an excellent measure to track the progress and quality of source code. For a healthy project development process, defect density should decrease over time.

2.3.6 Test coverage

Test coverage is a code metric that describes what percentage of source code is covered by tests. Basic test coverage criteria are (26; 27):

- **Statement coverage** - a percentage of statements from source code that has been executed by a test suite
- **Branches coverage** - a percentage of control flow branches from source code that has been executed by a test suite

High test coverage is necessary for risk management in software development. Without it, the organizations can miss failures and errors that can severely impact their project.

2.4 Visualising codebase

In this section, we will explore some visualisation tools that present data obtained from codebases in a meaningful way. In our project, we are more interested in exploring visualising code metrics rather than focusing on more popular dependency and structural diagrams.

2.4.1 Gource

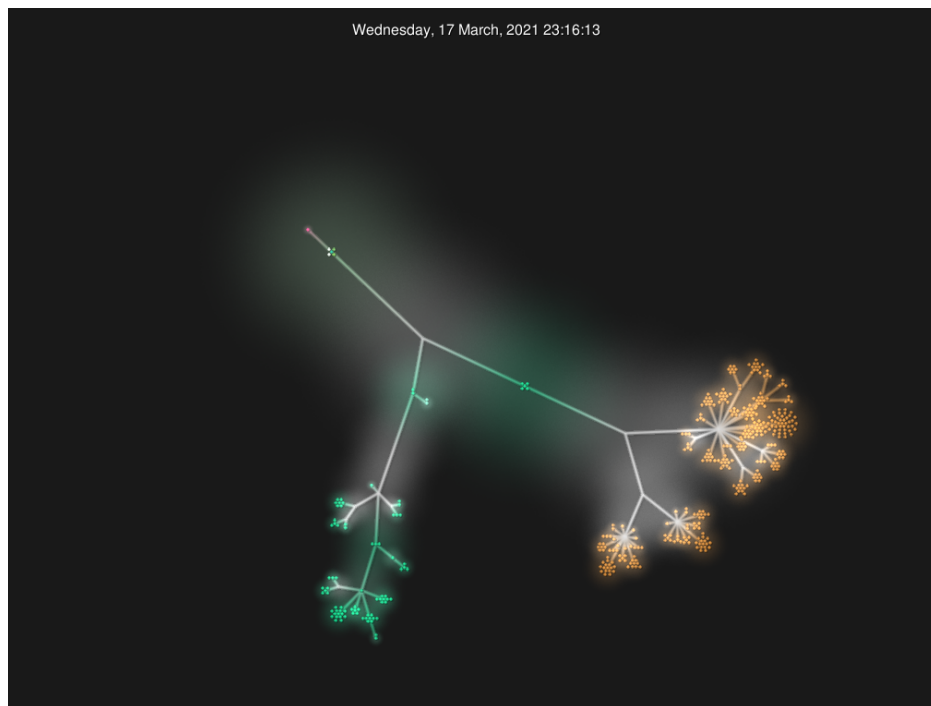


Figure 2.1: Final visualisation of a simple codebase using Gource (29).

Gource (29) is a tool that creates a video visualisation of changes made over time to the git repository for a given software engineering project. One of the features of this tool allows us to pick a starting point for the video visualisation. It displays version control data as a tree in which branches represent directories and leaves represent files. Moreover, each leaf has a colour depending on its file format.

Source is an exciting tool; however, it does not provide much data about the project being analysed.

2.4.2 CodeCity

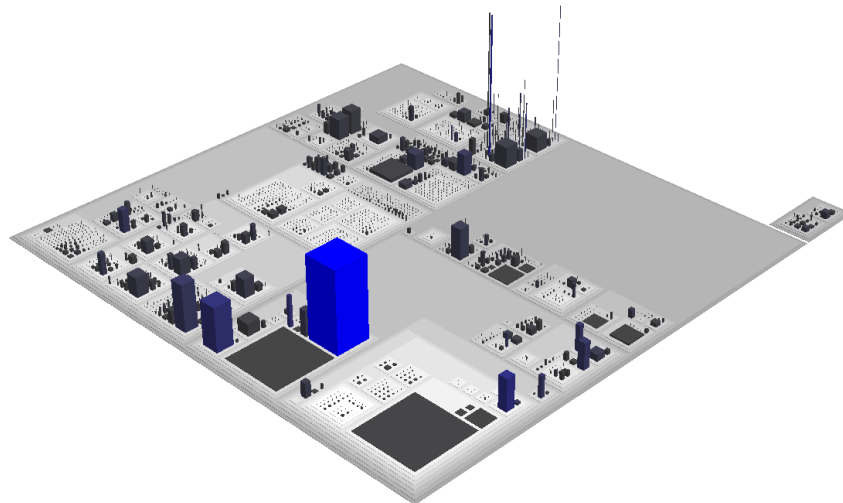


Figure 2.2: visualisation of a codebase using CodeCity (28).

CodeCity (28) is a tool that visualises a software engineering project as a city map. The classes are represented as buildings and packages as districts. The most notable code metrics that are visualised in CodeCity are lines of code and the number of methods. This tool has an interesting and unique approach; however, there are still not many measures displayed in the final result.

2.4.3 GoCity

GoCity (30) is another variation of software that uses a "city metaphor" to visualise a codebase. However, it only works with projects written in Go. The way it constructs its map is similar to CodeCity (28). The significant difference is that in GoCity, there is an object that represents structs visualised on top of "buildings".

2.4.4 GitHub Next

GitHub Next(35) has developed a visualisation tool that aims to help developers get familiar with a codebase. It uses a circle packing chart to visualise data about files. Currently, it generates a codebase diagram that shows the structure of a repository and the size and extension of all files in the repository. The visualisation is clear and

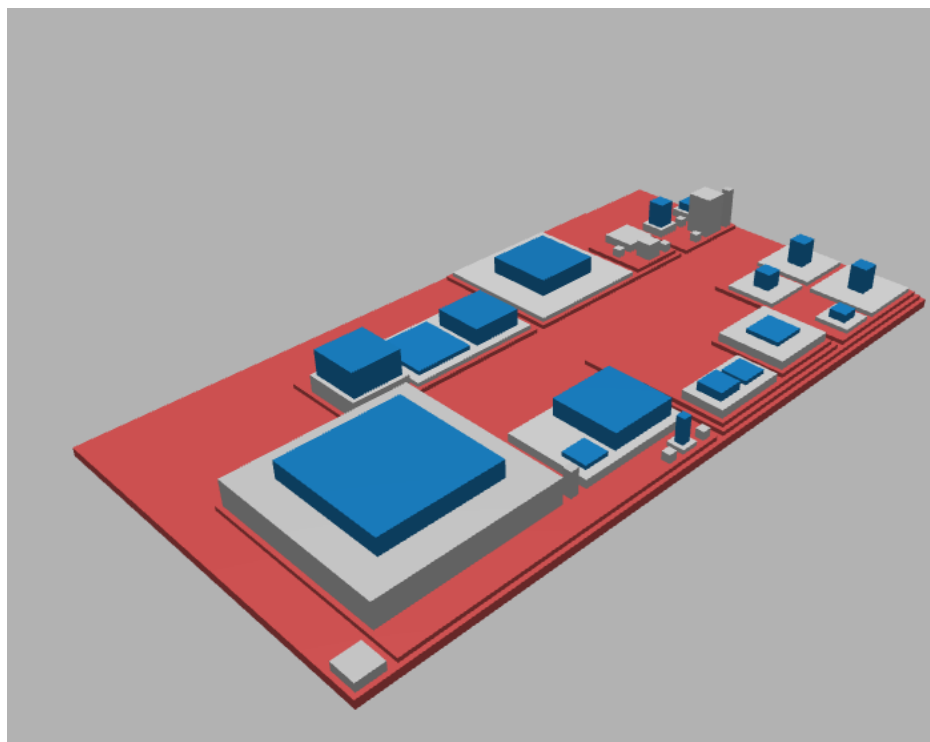


Figure 2.3: visualisation of a codebase using GoCity (30).

deepmind/alphafold

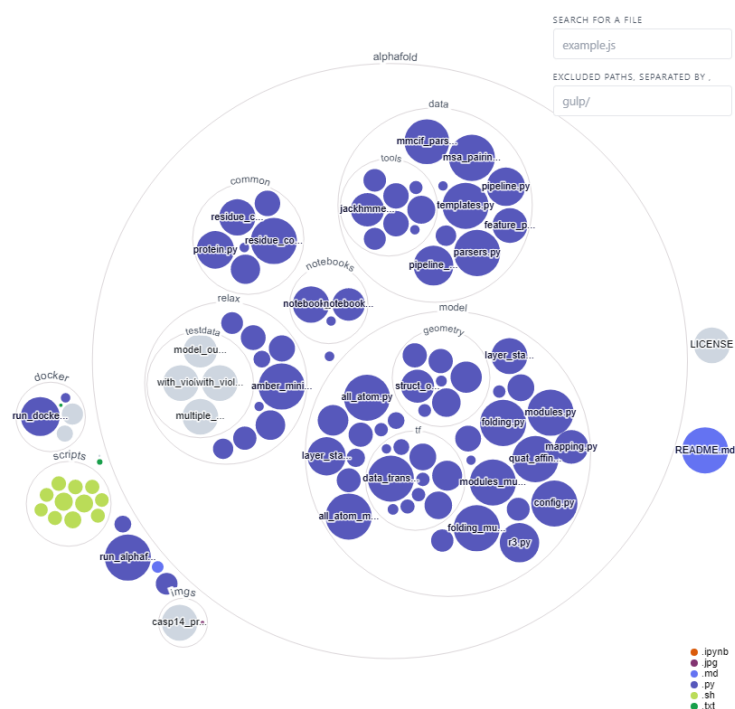


Figure 2.4: visualisation of a codebase created by GitHub Next visualisation tool (35).

accurately represents the structure of a repository. However, there are some issues with it. It seems not to visualise specific files and represents some files as nearly invisible circles. To use this tool, developers can add it to their GitHub actions, and the diagram will be re-created every time there is a new change to a codebase.

2.5 Mining Git repositories

Mining git repositories is a valuable way of obtaining data about the software development process. By getting data directly from version control instead of code, we will have access to information about contributors, commits, and other features that can be derived only from Git. That gives us unique data that would not be accessible via a simple code scan. There have been several tools created to allow programmers to mine Git repositories.

2.5.1 PyDriller

PyDriller ([31](#)) is an open-source framework for Python that provides software developers with a neat and straightforward way to extract data from Git repositories. By using this framework, we can obtain information about commits, contributors, modified files, and source code. PyDriller has classes corresponding to Git data, such as Repository, Commit, and Modified File. By utilizing these classes, software developers can interact with data more efficiently.

PyDriller also has a range of basic metrics that it is able to calculate automatically:

- **Change Set** - files committed at the same time.
- **Code Churn** - metric of code churn of a file.
- **Commits Count** - number of commits made to a file.
- **Contributors Count** - number of contributors that made a commit to the file.
- **Contributors Experience** - percentage of lines of code that a main contributor of a file authored.
- **Hunks Count** - number of continuous blocks of changes made to a file.
- **Lines Count** - number of lines included in a commit, for a specific file. Both added and removed lines of code are calculated.

These measures would be extremely helpful in our process of exploring and calculating more advanced metrics.

2.5.2 GitHub RESTAPI

GitHub RESTAPI ([32](#)) is an official API released by GitHub. It provides software developers with a wide range of data that can be mined from Git repositories, such as pull requests, issues, repositories, commits, branches, and more. The GitHub RESTAPI is much more advanced than PyDriller ([31](#)) in terms of the range of information that can be derived from it. It also has several wrappers that make it easier to use in most popular programming languages

Chapter 3

Design and implementation

3.1 Architecture overview

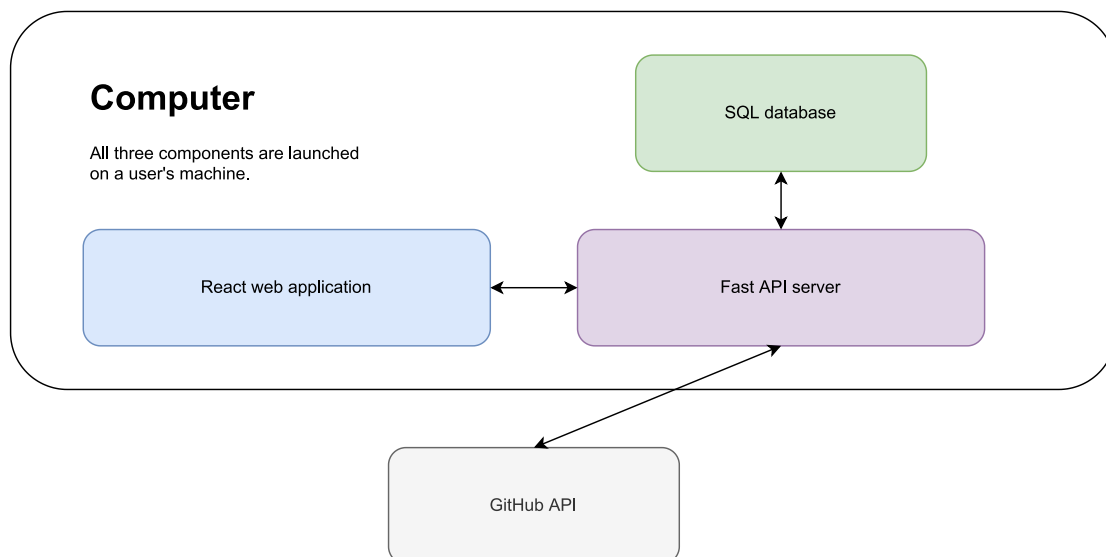


Figure 3.1: High level architecture diagram of Metrico

Metrico is made of three main components:

- **Frontend - React web application:** The frontend of our application has been developed using React web framework. It generates visualisations given data from a server.
- **Backend - FastAPI server:** The backend of our application has been developed in the Python framework FastAPI. The backend handles frontend requests and communicates with GitHub API and the database.
- **Database:** Metrico uses SQL Lite database for storing repository data and metrics

The app has been designed with the premise that the user will launch both backend and frontend on their own machine. However, it's still capable of being hosted on a

server machine through the network.

The backend of our web app is designed to get the repository data from GitHub API, compute the metrics and efficiently serve the metrics to the frontend. Metrico aims to make an analysis of the codebase as smooth as possible. The data should be sent to the frontend immediately, without long loading times. Because of that, systems have been created to cache data and metrics in a database system. That way, the most time-intensive operations will be done in the pre-processing time when the database will be populated, and serving this data later to the frontend of an app will be instant. The downside to this approach is that pre-processing times can get a bit long. However, the trade-off of no loading times and efficient analysis is crucial for the comfortable usage of this app.

3.1.1 Technologies used

FastApi

FastAPI(39) is a modern Python web framework used to build APIs. It handles HTTP requests and responses while being fully compatible with OpenAPI(55) standards. FastAPI is a relatively new framework that focuses on performance and ease of use.

Uvicorn

Uvicorn(44) is an ASGI web server implementation for Python, picked for this project precisely because of its compatibility with FastAPI and its support for asynchronous frameworks.

SQLAlchemy

SQLAlchemy(45) is a Python SQL toolkit and an Object Relational Mapper framework. It provides users with an easy way to create, manage and perform database operations through the SQL abstraction toolkit. SQLAlchemy comes with a data mapper that allows programmers to represent their database models in their Python code.

SQL Lite

SQL Lite(46) is a small, fast and reliable SQL database engine. It is also widely used as one of the most popular database engines in the world. It was used in this project as it is lightweight, easy to use and integrates well with SQLAlchemy.

React

React(47) is a modern, open-source frontend JavaScript library for building user interfaces. According to the Stack Overflow developers survey, it was the most commonly used web framework in 2021(50). React is based on components with their own state and rendering logic. React is a powerful tool that provides a considerable performance boost compared to pure JavaScript web apps. It can achieve this by utilising the Virtual DOM concept, which results in React efficiently updating the browser's DOM by rerendering required components instead of the whole page. React has been picked as this project's web framework mainly because of its popularity, ease of use and performance advantages, although some other choices were considered, such as Vue.js(52).

Redux

Redux(41) is a predictable state container for JavaScript apps, most commonly used with React. Redux can be used to circumvent the top-to-the-bottom approach of passing the data in React, which in some cases avoids additional re-rendering and helps the overall performance of an app. For this project, the React-Redux(43) wrapper has been chosen for easier integration with our frontend. By using this version, we get dedicated hooks that make using this framework really easy.

React Router

React Router(40) is a React framework that handles client-side routing. It allows React to decide what components should be displayed under different URLs and enables developers to build multi-page React applications.

D3

D3(42) is a JavaScript library that allows users to create dynamic and interactive visualisations. It uses HTML, CSS and SVG to create data-driven graphical representations. Other libraries such as Recharts(53) and React Charts(54) were considered for this project. However, they are limited in terms of what can be achieved using them. That is why D3 was picked, as it is a much more flexible and powerful library.

Bootstrap and Blueprint

To speed up the development process, JavaScript frontend toolkits have been picked such as Bootstrap(48) and Blueprint(49). Both are powerful and popular libraries that consist of prebuilt components. They help the developer create compelling and transparent user interfaces.

3.2 Getting version control data

For Metrico to calculate metrics, it has to obtain commit and pull request data from GitHub. We decided to use PyGithub(51) - a Python wrapper for GitHub REST API. At first, PyDriller was considered; however, after experimenting with this framework, we decided to use a more flexible and efficient PyGithub.

The GitHub REST API can provide us with the data about the repository, branches, commits, pull requests and much more. However, to obtain the data, the app needs to get a GitHub access token from a user. GitHub Access Token can be generated in GitHub settings with specified permissions. For the app to work correctly, the user should pass an access token with “repo” permissions that will allow the backend to query the API for both public and private repositories.

When the Metrico wants to make a call to the GitHub API, it always first invokes the `get_repo()` method from the PyGithub framework with the user’s access token and repository name as arguments. If a user does not have permission to access the repository, the API will throw a 403 Error, and the user will receive an error.

When the validation is complete, the PyGithub call will return a repository object. The backend app will then invoke these two methods:

- `get_commits()` - given the branch name and two dates, this method will return the list of commits created between a start date and an end date
- `get_issues()` - given the start date, this method will return every issue modified after the start date

In PyGithub, pull requests are represented as issues. That is why the backend queries the `get_issues()` method to get the pull requests data. After getting the issues, we have to filter out every issue that is not a pull request.

The most significant limitation of the GitHub Rest API is that it has a limit of calls per user per hour of 5000. It means that for a large repository, it is impossible to load the data of all the commits from a more considerable period of time. To combat this, we created a cache database system that will allow loading the data from the API in smaller parts. However, it is still a substantial bottleneck, as larger requests will have to be loaded for hours, depending on the resets of an API limit.

3.3 Metrics

3.3.1 Picking metrics

Several metrics have been considered for this project. However, we wanted to pick only a small number, to test them properly and not overwhelm users. As we explored

different code metrics, we came up with two categories of metrics.

The first category is code metrics calculated from the contents of files, for example, cyclomatic complexity, code duplication or lines of comments in a file. These metrics force us to know file's exact contents to calculate them. They rely on code analysis, which also depends on the programming language used.

The other category of metrics we characterised were metrics calculated from changes to the codebase. We can think of them as metrics calculated on the repository's activity instead of the contents of particular files. These are, for example, added lines in a commit, the number of commits that modified a file or the number of contributors.

The decision was made to choose the metrics describing changes to the codebase. The huge advantage of these metrics was that we take our data from GitHub RESTAPI, and we have easy access to commit and pull request data. Moreover, calculating metrics based on file contents for a potentially large number of commits could be more expensive.

As our metrics, we picked Added lines count, Contributors count, Code churn, Commit count and Pull requests per file.

3.3.2 Calculation of metrics

Metrico currently calculates five metrics for every repository that it analyzes. For a given time period, these metrics represent:

- **Added lines count** - the number of lines added to the given file
- **Contributors count** - the number of contributors that authored a commit that changed a given file.
- **Code churn** - the difference between added lines and removed lines for a given file
- **Commit count** - the number of commits that modified a given file
- **Pull requests per file** - the number of open pull requests that modify a given file. To calculate it, we get a list of every pull request that has been created before the start of a time period and merged after.

The metrics are calculated in the weekly and monthly periods. All of the computations are done in the pre-processing phase if the new commits are loaded to the database.

3.4 Database models

Metrico's database system stores data in seven models. The complete schema is shown in the figure 3.2. All operations that describe interaction with the database are written in the `crud.py` file in the database directory.

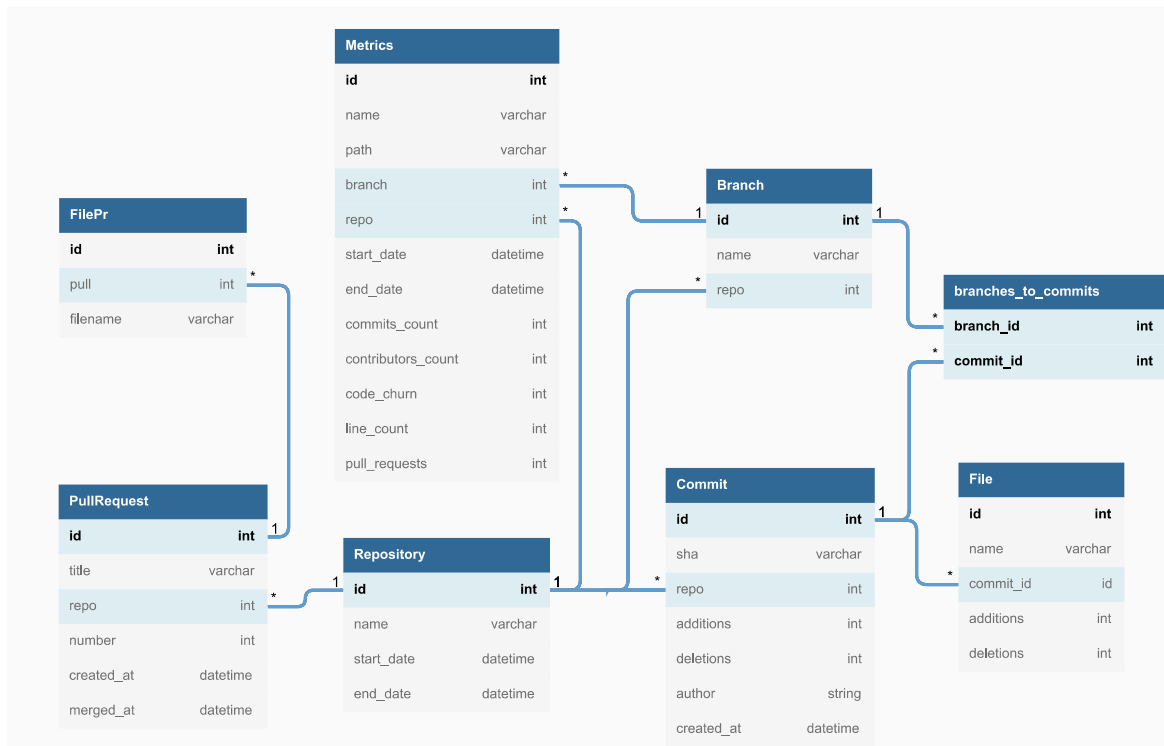


Figure 3.2: Metrico's database models and relationships between them.

Repository

The Repository object is created for each Github repository analysed by a user. The `id` field is an auto-populated index. The `start_date` and `end_date` fields hold the information about the time period in which the data for this repository is up to date in a database. Both dates should be updated every time the new data has been loaded into the repository.

Branch

Every branch analysed by a user will be represented as a Branch object. Each Branch model references the Repository object as a many-to-one relationship. Branch also

has a many-to-many relationship with a Commit model that is represented by a `branches_to_commits` table.

Commit

The Commit table holds data about each commit from the analysed branch. It references the Repository and File models illustrated in a figure 3.2. As stated before, the Commit model has a many-to-many relationship represented in a `branches_to_commits` table. The Commit object also holds data about sha, additions, deletions, author and time of creation (`created_at`).

File

The File model does not represent a file from a repository but rather the file statistics for a specific commit. It holds the `name(path)`, `commit_id`, additions and deletions information. The decision was made to not create a model for specific Files as this app deals more with changes to the codebase rather than specific file contents. However, this can be further explored in the future development of a tool.

Metrics

The Metrics model represents the metrics for a specific file in a specific time frame. The time frame is represented by the `start_date` and `end_date` fields. The file is characterized by a `path` and `name` fields. The Metrics model has relationships to the Repository and Branch models, which are many-to-one relationships.

PullRequest

The PullRequest model contains data about the pull requests taken from an analysed repository. It has a reference to the appropriate repository object. It also holds information about when the pull request has been created (`created_at`) and merged (`merged_at`). For open pull requests, the `merged_at` field will be null. Every pull request has a one-to-many relationship with the FilePr model.

FilePr

The FilePr model represents changes to a file made by a `pull_request`. Similarly to the File model, it shows changes made to a file rather than a file itself.

3.5 Serving the data to frontend

The main aim of the backend of Metrico is to send the relevant data to the frontend. The frontend interacts with the backend in two ways.

/POST/init

This call sends the data of the repository that should be initialised by a backend. The data sent through this request has a structure of:

- `repName` - name of the repository in a specific format - `owner/repository_name`. For example the React repository would have to be put in as `facebook/react`.
- `accessToken` - personal access token generated from the users GitHub account
- `branch` - the name of the main branch that the user wants to calculate metrics for
- `startTime` - the date which represents the beginning of the time period that a user wants to analyse
- `endTime` - the date which represents the ending of the time period that a user wants to analyse

After receiving the data, the backend will have to check if a given repository has already been initialised. If it has not, the backend will issue a call to the GitHub REST API and perform pre-processing, creating all necessary objects and calculating all the metrics. However, if the given repository has already been found in a database, then the backend will have to check if a given time frame is already included in a database. If not, it will issue necessary calls to the GitHub API, load the data into the database and update the metrics.

After all of the metrics are calculated, the app divides the time period picked by a user into weeks and months. It takes the dates that represent them and put them into the list that will be returned to the frontend with metrics from last week.

/POST/metrics/

The `metrics` call receives the same data from frontend as an `init` call. However, in this case, `startTime` and `endTime` represent the dates of the week or month for which the frontend wants to know metrics. Upon receiving the data, the backend queries the database for a metrics from the given time frame and returns the metrics object to the frontend.

3.6 User Flow

Starting page

Upon starting Metrico, the user will see a webpage with a form asking them about necessary information to initialise the repository visualisation (shown in [A.1](#)). To start the initialisation, the user will have to type in the data outlined in [3.5](#).

After filling out the form and clicking the “send” button, the `init` call will be received on the backend, and after some time, a user should be taken to the visualisation page.

Visualisation page

After first entering the visualisation page (shown later in 4.1), the user will see the navbar. On the navbar, the user can pick the metrics that the visualisation should represent and the time period. After clicking the “apply metrics” button, the visualisation should appear with the applied metrics that the user picked. After changing the time period or metrics, the frontend will issue another `metrics` call to get the metrics for a newly selected configuration and rerender the visualisation.

Help page

A help page (shown in A.2) is a simple page with a description of all the metrics used and a quick guide on creating a visualisation. It is accessible through the button that is located on a navbar on a visualisation page.

3.7 Visualisation

3.7.1 Considered visualisations

Throughout the project’s development, several different visualisations were considered. We looked for charts that would represent several metrics in a clear and meaningful way. We also wanted to preserve the locality of files in a codebase. It means that we were looking for a visualisation that would clearly represent that files are in the same directory. We also wanted to show as many relations between metrics as possible without overwhelming the user.

Bubble Chart

The bubble chart (shown in 3.3) was our initial idea. In this type of visualisation, bubbles would represent the files. Each bubble would represent one metric with its colour and another with its size. To decide on the placement of the bubbles in the chart, we had to assign some meaning to the axes. The decision was made to add metrics to the axes, one representing the x-axis and the other representing y-axis. In that case, we could represent relations between 4 metrics using this visualisation method. However, the bubble chart, in this case, does not preserve the locality of our files. On top of that, we found out that most of the time during our analysis, we would just look at one of the corners, disregarding most of the chart entirely.

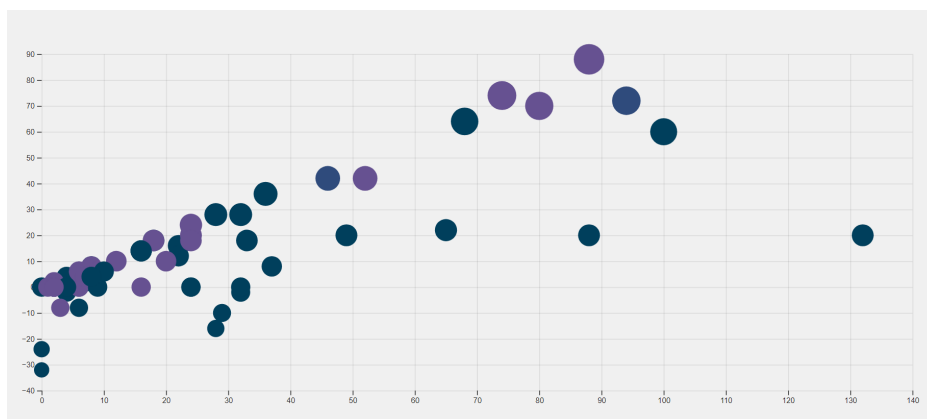


Figure 3.3: Example of a visualisation using bubble chart

Clustered Buuble Chart



Figure 3.4: Example of a visualisation using clustered bubble chart

The clustered bubble (shown in [3.4](#)) chart inherits many ideas from a standard bubble chart. It can represent metrics both as bubble size and colour. However, it differs from a standard bubble chart because it groups the bubbles into clusters, and it does not have meaningful x and y axes. The clusters can be determined by the value of a specific metric. There was a lot of experimentation with different metrics. However, the most effective grouping was by parent directory of a file. It looked promising, although it did not accurately represent the project hierarchy, since all of the clusters appeared on the same level.

Circle Packing Chart

The circle packed chart (shown later in 4.1) is a variation of a clustered bubble chart, inspired by observable blog post(37). It provides a way to represent two metrics, again represented by bubble size and colour. The main advantage of the Circle Packed Chart is that it preserves the file structure hierarchy in a repository. That way, the users get a more accurate representation of a codebase, and they can find a particular file easier.

Ultimately, we decided to pick Circle Packing Chart for our visualisation.

3.7.2 Features of visualisation

As Metrico focuses on changes made to the codebase, it is worth noting that the files, which are represented by bubbles, will only appear in the visualization if a change has been made to them in a given time period. It means that we will not show the entire structure of a project in every visualization but rather only the part that was interesting (changed) in a given time frame.

The chart is zoomable, so if a user wants to zoom to a certain level, they should click on a circle that interests them, and the whole chart will be transformed. The user should click the same circle a second time to zoom out. There are also labels in the chart for every circle. However, only those on the same level as the current zoom will be displayed.

To get a numerical value of metrics for a specific file, a user should click on the circle. The data should get updated on the sidebar on the left side of the screen, and the user should get access to all the metrics calculated for a specific file.

3.8 Extending Metrico

As Metrico is supposed to be used for experimentation, we have to ensure that it can be extended by researchers or software developers.

Extending the models

Currently, the models do not represent the complete data that can be extracted from GitHub REST API, so there is some potential to expand on the information stored in the database. It is possible to add other fields to the database with relatively low overhead. To do that, the developer would have to add a new field to the appropriate model and scheme in `models.py` and `scheme.py`. On top of that, some changes to the related functions in the `crud.py` file may be required.

Extending metrics

The process of adding a new metric to the app is not much harder than adding a new field to the database. To add a new metric to the app a developer has to:

- Add a new function in `metrics.py` that will calculate a dictionary that represents a mapping between a file path and a new metric. This function should return a said dictionary, given appropriate data (either commit, pull requests or both)
- Add call to this function in the `init_metrics` method that will calculate this metric for a given time frame
- Follow the steps described in Extending the models paragraph (see [3.8](#)). Every metric should have its field in the database Metric model. A developer should add a field to a Metric model in `model.py` file and MetricsBase schema in the `schema.py`.
- It can also be essential to change some functions in the `crud.py` file to account for a new metric.

Chapter 4

Results and evaluation

4.1 General evaluation

The main objectives of this project (as specified in 1.2) were to produce a tool that can help researchers and software developers calculate the code metrics and produce their visualisations. We want to show that Metrico can produce various visualisations depending on the metrics and time period.

We will use a twbs/bootstrap repository to showcase the visualisations. It is a repository of a JavaScript framework, Bootstrap, that we described in 3.1.1. It contains around 21000 commits and 13000 pull requests, and the repository is still active and frequently updated.

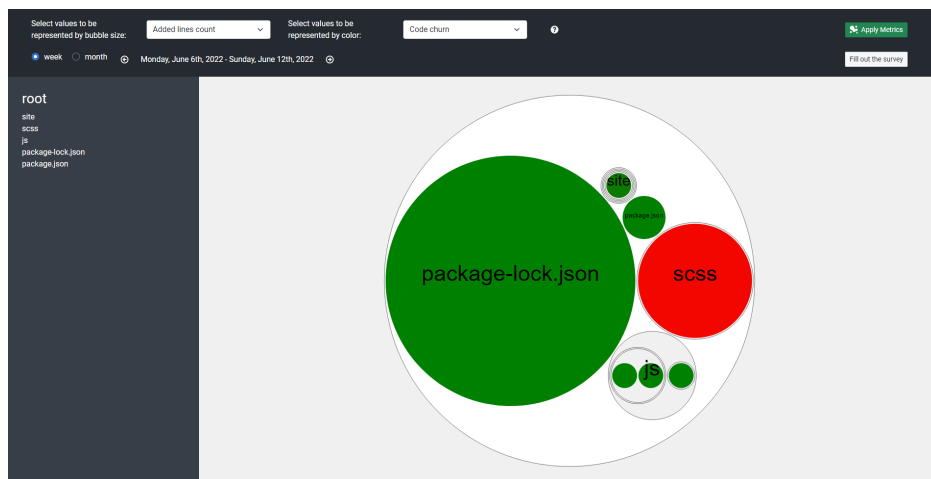


Figure 4.1: visualisation generated using Added lines count and Code churn

To showcase the visualisations we initialised Metrico with a month of repository data from twbs/bootstrap. By selecting Added lines count as a metric representing size and Code Churn as a metric representing colour, we were shown a visualisation as in figure 4.1.

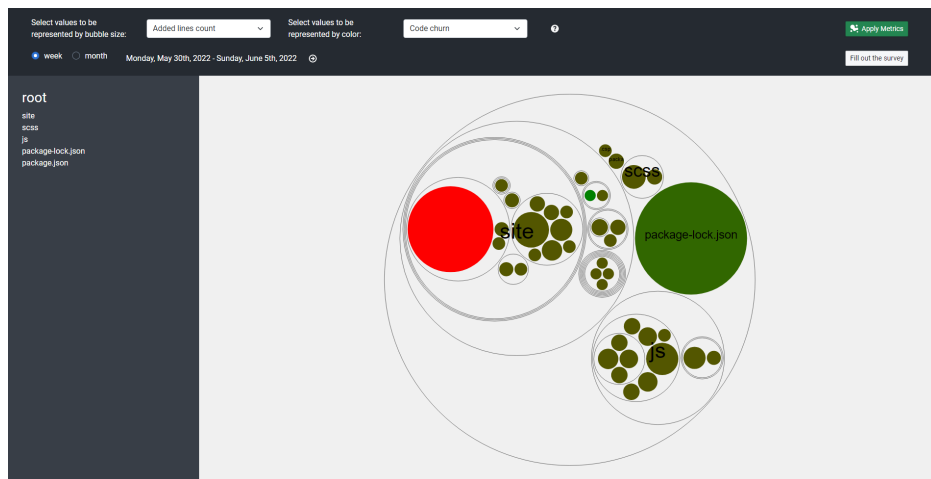


Figure 4.2: visualisation generated in different time period

By changing a time period (shown in figure 4.2) we obtain a completely new visualisation that shows that code metrics for files changed in the new time frame.

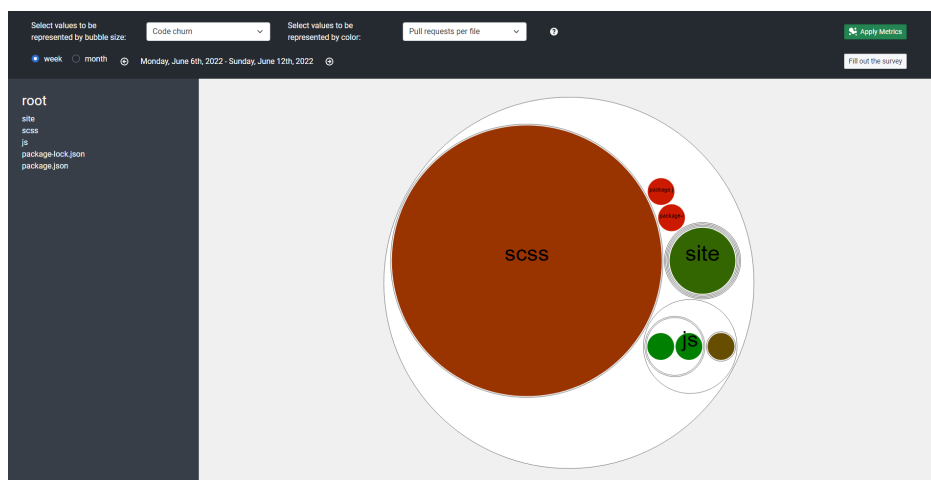


Figure 4.3: visualisation generated using Code churn and Pull requests per file

We can also see that changing metrics from our initial choices will produce new results (see figure 4.3).

4.2 Survey

We conducted a small study among 20 software developers to investigate our visualisation and gain feedback on the metrics we used. The main reason for conducting this survey is to prove that Metrico can be used to research code metrics, visualisations and their relationship with risk management. We also wanted to see if there would be any difference in using the tool if developers have previous knowledge of the codebase. That is why we divided our study into two groups. In the first group,

15 developers answered questions about the codebase that they were not familiar with. For the second group, five software developers from a particular company have been picked. The second group had to answer questions about the codebase they have been working on for the past couple of months.

Both groups have received four tasks to complete on the visualisations of their respective codebases. They also answered some general questions about metrics and our tool. With this survey, we wanted to check how meaningful the visualisations of our tools may be to the software developers using them. As there are no objectively right answers to any of the questions in the tasks, we wanted to explore what metrics setup the users would use and if any patterns would emerge.

The first group had to work with a visualisation of a twbs/bootstrap repository (described in 4.1). However, the second group was working on the visualisation of the private repository of their platform called “MaturaIT”(38). It was a web application repository consisting of around 2200 commits and 300 pull requests. The full repository name is MaturaIT/maturait_vue.

Both groups received the same four tasks, and had to answer these questions:

1. What file may cause the most problems when merging the pull requests in the coming weeks?
2. What file was the most “popular” in the month of April, 2022?
3. Which file in your opinion is at most risk?
4. Which file in your opinion should be targeted for refactoring?

4.2.1 First group - survey results

Task 1

In the first task, the file picked as the answer the most often was `.bundlewatch.config.json` with five answers (see Figure 4.4). The other often picked options were `borders.md` with three answers and `webpack.md` with two answers. Other files have been picked at most once.

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	6	1	7
Code churn	0	2	2
Commits count	4	1	5
Contributors count	1	3	4
Pull requests per file	4	8	12

Table 4.1: Metrics used by developers from the first group in task 1

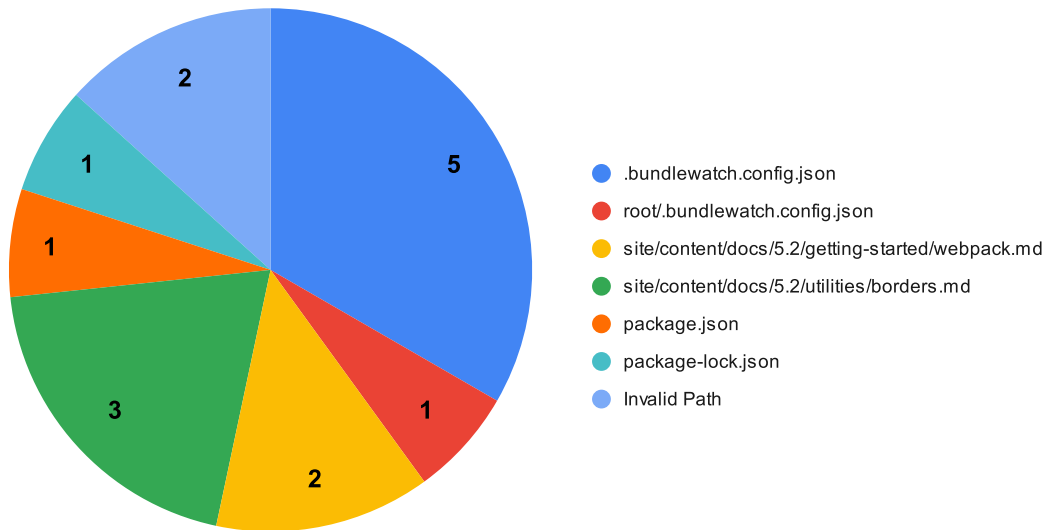


Figure 4.4: Answers of developers from the first group for task 1

However, the metrics picked for this task (see Table 4.1) has been similar as 12 developers picked Pull Requests per file as their metric for this task (4 as size metric and 8 as color metric). Other often picked metrics were Added Lines Count (7 times), Contributors Count (5 times) and Commits Count (4 times). What's interesting is that the Code Churn metric has only been picked once for this task.

Task 2

For the second task, the most popular answer was `package-lock.json` which was picked by 7 developers (see Figure 4.5). The second most often picked file was `scss/_variables.scss` and was picked by two developers. Surprisingly, all other developers picked a unique file as an answer.

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	3	1	4
Code churn	2	5	7
Commits count	1	4	5
Contributors count	9	2	11
Pull requests per file	0	3	3

Table 4.2: Metrics used by developers from the first group in task 2

Developers picked various metrics to visualise this task. Unsurprisingly, most of the users picked Contributors count as one of their metrics (73% developers). Other metrics that may represent “popularity” of a file has also been picked often. Code Churn, Commits Count and Added Lines Count has been picked respectively 7, 5 and

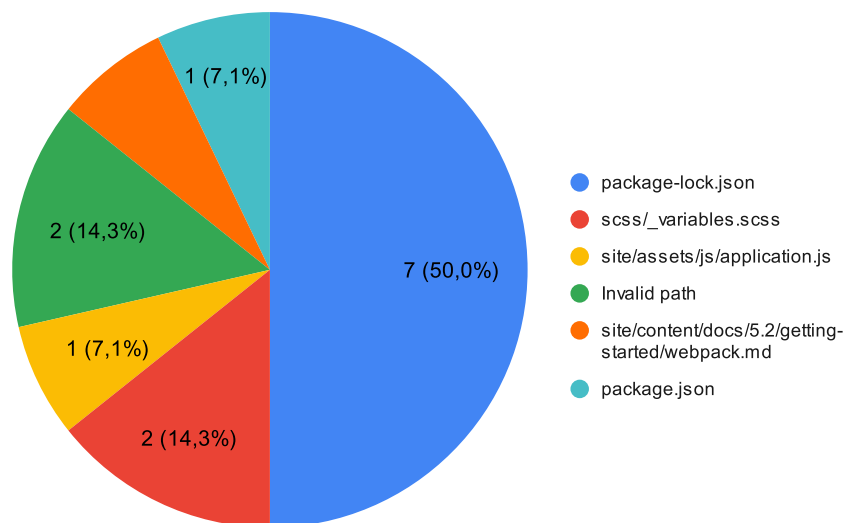


Figure 4.5: Answers of developers from the first group for task 2

4 times (see Table 4.2).

Task 3

For the third task, there was not one most popular answer. Both `application.js` and `webpack.md` has been picked the most often - 3 times. Other files picked more than once were `event-handler.js` and `package-lock.json`. Both were picked 2 times (see Figure 4.6).

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	6	2	8
Code churn	3	3	6
Commits count	2	0	2
Contributors count	2	3	5
Pull requests per file	2	7	9

Table 4.3: Metrics used by developers from the first group in task 3

The choice of metrics for this task also varied (as shown in Table 4.3), with Added Lines Count and Pull Requests per file being the most popular (used 9 and 8 times respectively). Code Churn and Contributors Count has also been used quite often (6 and 5 times respectively).

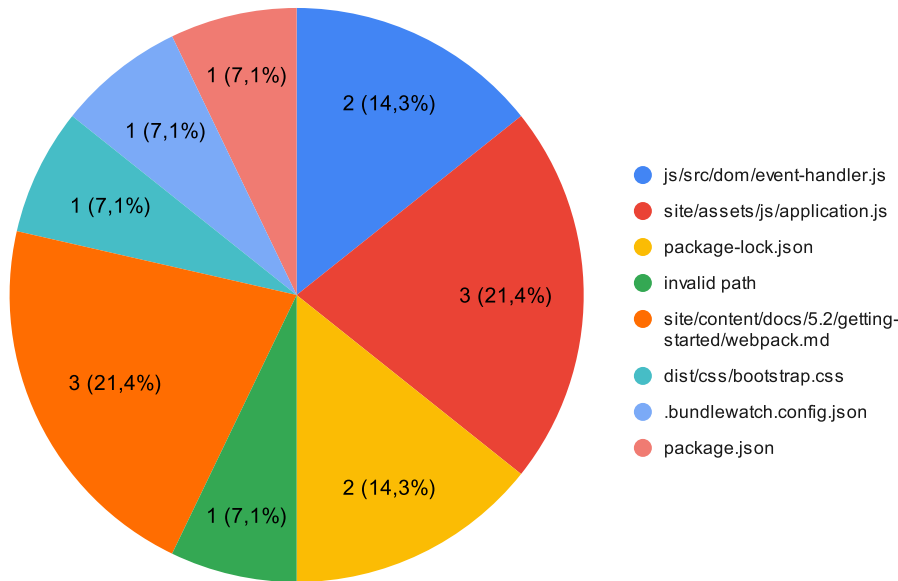


Figure 4.6: Answers of developers from the first group for task 3

Task 4

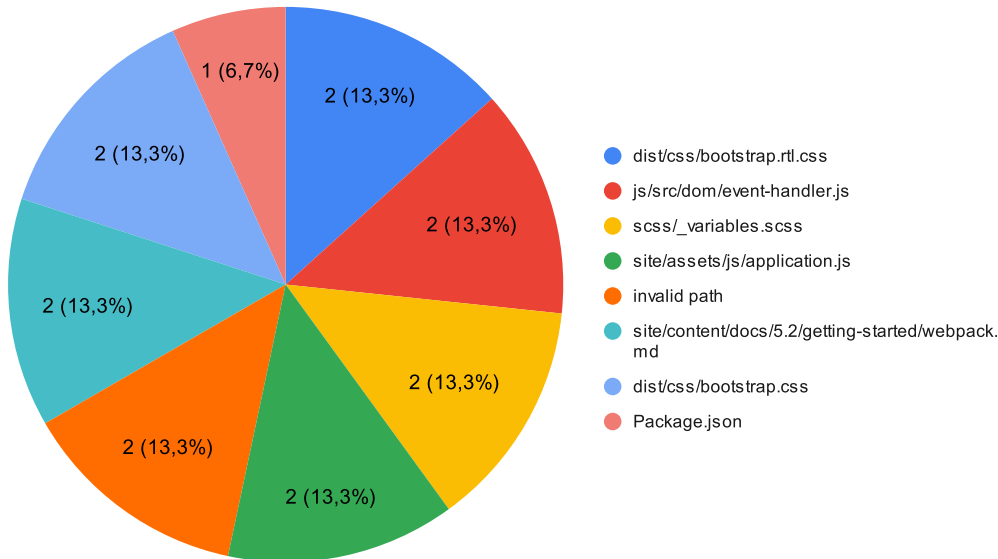


Figure 4.7: Answers of developers from the first group for task 4

For this task, the answers were also varied as 6 files were picked 2 times (see Figure 4.7). However, the metrics picked were much more similar as most developers

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	8	1	9
Code churn	3	8	11
Commits count	1	3	4
Contributors count	1	2	3
Pull requests per file	2	1	3

Table 4.4: Metrics used by developers from the first group in task 4

picked Added Lines Count and Code Churn metrics (used 9 and 11 times, respectively, see Table 4.4).

4.2.2 Second group - survey results

For the second group, the questions were the same as for the first group.

Task 1

Results of task 1

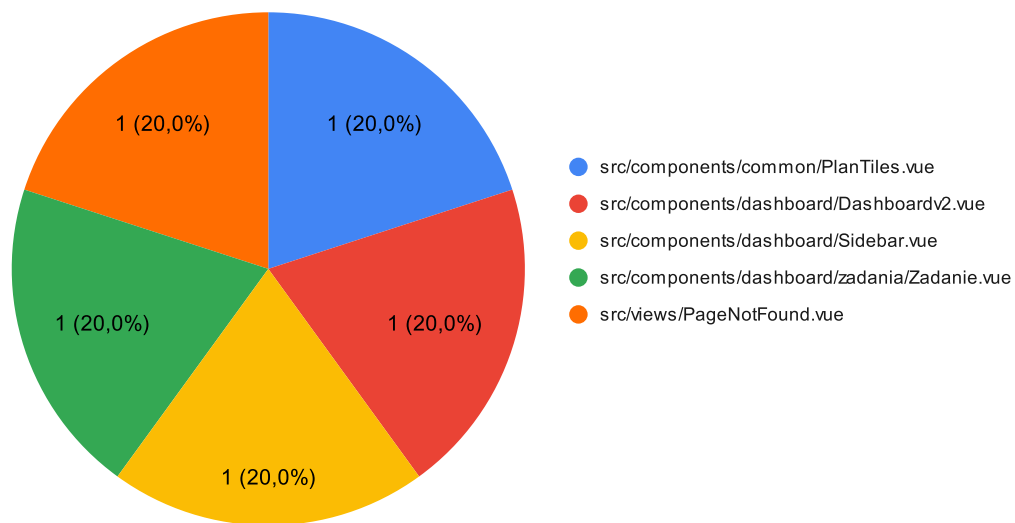


Figure 4.8: Answers of developers from the second group for task 1

In this task, the developers have all picked different answers (see Figure 4.8). It's surprising since they picked similar code metrics. Pull requests per file have been picked 4 times, Added lines count 3 times and Contributors count 2 times (see Table 4.5).

Task 2

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	2	1	3
Code churn	1	0	1
Commits count	0	0	0
Contributors count	0	2	2
Pull requests per file	2	2	4

Table 4.5: Metrics used by developers from the second group in task 1

Results of task 2

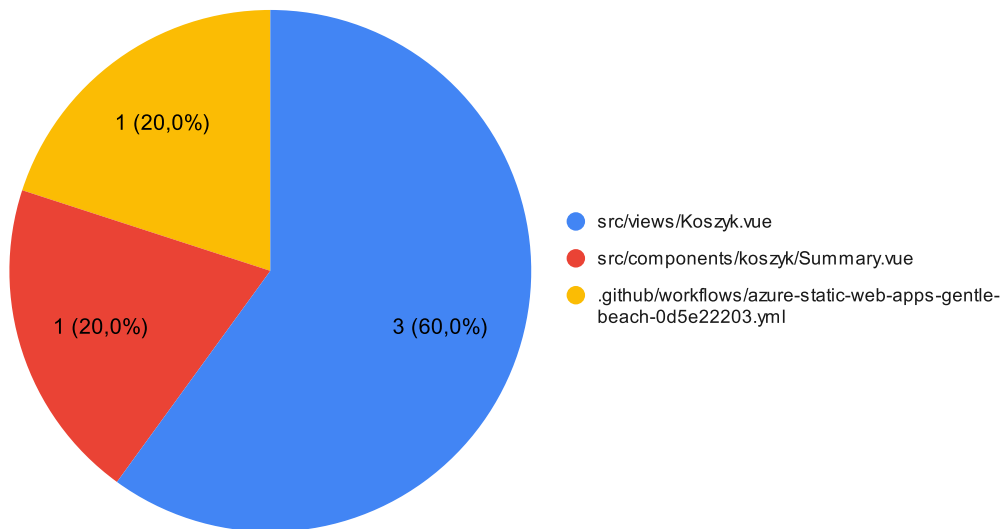


Figure 4.9: Answers of developers from the second group for task 2

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	3	0	3
Code churn	2	0	2
Commits count	0	4	4
Contributors count	0	1	1
Pull requests per file	0	0	0

Table 4.6: Metrics used by developers from the second group in task 2

Most developers from the second group picked `Koszyk.vue` as the most popular file (60% as described in Figure 4.9). The metrics that they used in this task were Commits count, Added lines count, Contributors count and Code churn (respectively 4, 3, 2 and 1 times, see Table 4.6).

Task 3

Results of task 3

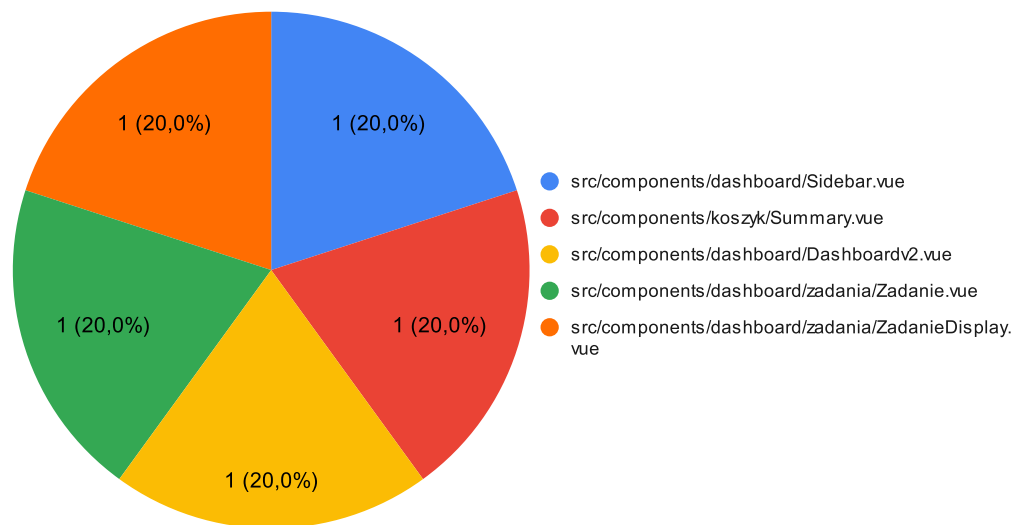


Figure 4.10: Answers of developers from the second group for task 3

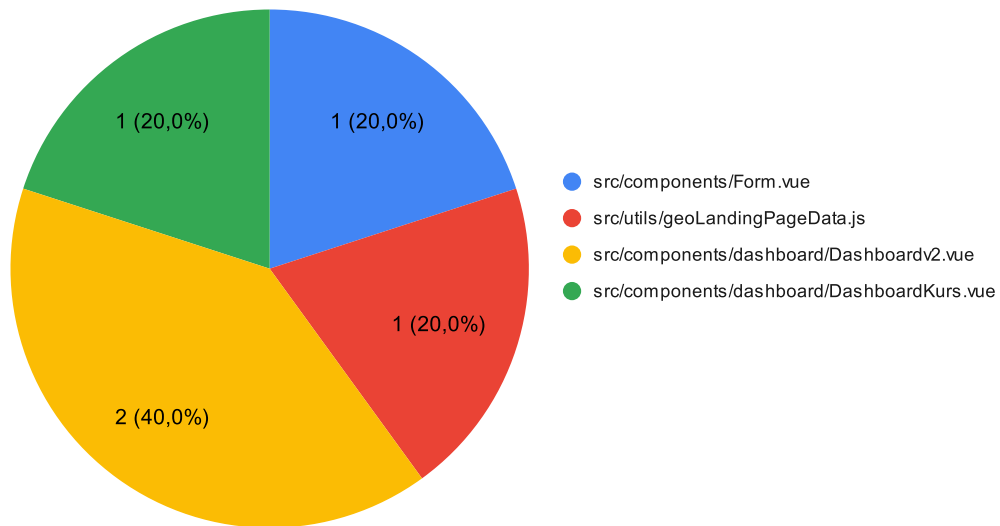
Metrics	Times used as size metric	Times used as color metric	Total
Added line count	3	0	3
Code churn	0	1	1
Commits count	1	1	2
Contributors count	0	2	2
Pull requests per file	1	1	2

Table 4.7: Metrics used by developers from the second group in task 3

In this task, developers from the second group all had unique answers (see Figure 4.10). There was also a lot of variety in the metrics picked, as all metrics have been picked at least once. Added Lines count has been the most popular metric, used 3 times (see Table 4.7).

Task 4

Results of task 4

**Figure 4.11:** Answers of developers from the second group for task 4

Metrics	Times used as size metric	Times used as color metric	Total
Added line count	3	1	4
Code churn	0	1	1
Commits count	1	0	1
Contributors count	0	3	3
Pull requests per file	1	0	1

Table 4.8: Metrics used by developers from the second group in task 4

For task 4 the most popular answer was `Dashboardv2.vue` which was chosen 2 times (see Figure 4.11). On top of that, most of the users picked Added lines count and Contributors count as a metric (see Table 4.8).

4.2.3 Comparison

For task 1, we can see that most of the developers from both groups had the same intuition with picking Added lines count and Pull requests per file as their metrics (see Tables 4.1 and 4.5).

In task 2, there were files in both groups that most of the developers voted for (see Figures 4.5 and 4.9). Most of the developers from the first group have used the Contributors count to analyze this question (see Table 4.2). However, the developers from the second group opted for the Commits count metric (see Table 4.6). The reason might be because the developers from the second group work in a small startup in which they vaguely know who is working on what part of the code. By that logic, this metric could be not very insightful for them. However, for larger projects such as “twbs/bootstrap” Contributors count might be more meaningful.

The results of task 3 for both groups were not conclusive (see Figures 4.6 and 4.10), which may be the result of the abstract nature of the question. In this task, we asked developers to find the most “risky” file in a codebase which is not well defined.

For task 4 there were no overwhelmingly popular answers in both groups (see Figures 4.7 and 4.11). However, developers from both groups have often used Added lines count metric to determine the answer (see Figures 4.4 and 4.8).

4.2.4 General questions

As shown in Figure 4.12 all of the metrics we presented to the developers have been deemed helpful by more than half of the users. Metrics that were found useful by most developers were Pull Requests per file and Code churn with 16 votes. Contributors count has been found useful by the least number of users.

We also asked developers to specify the metric they found the most and least useful. Universally, contributors count has been found to be the least useful metric across both groups (see Figure 4.14). The votes for the most useful metric have been more split, with Code Churn winning slightly with only 33% of the votes (see Figure 4.13).

At the end of the survey, we asked the developers their opinions about the tool (see Figure 4.15). We asked how useful would our tool be when analysing changes made to the codebase and visualising the code metrics. Most users had a positive reception and gave our tool at least a 4 out of 5 score. Overall, Metrico’s average score was a 3.5.

Which of these metrics have you found useful?

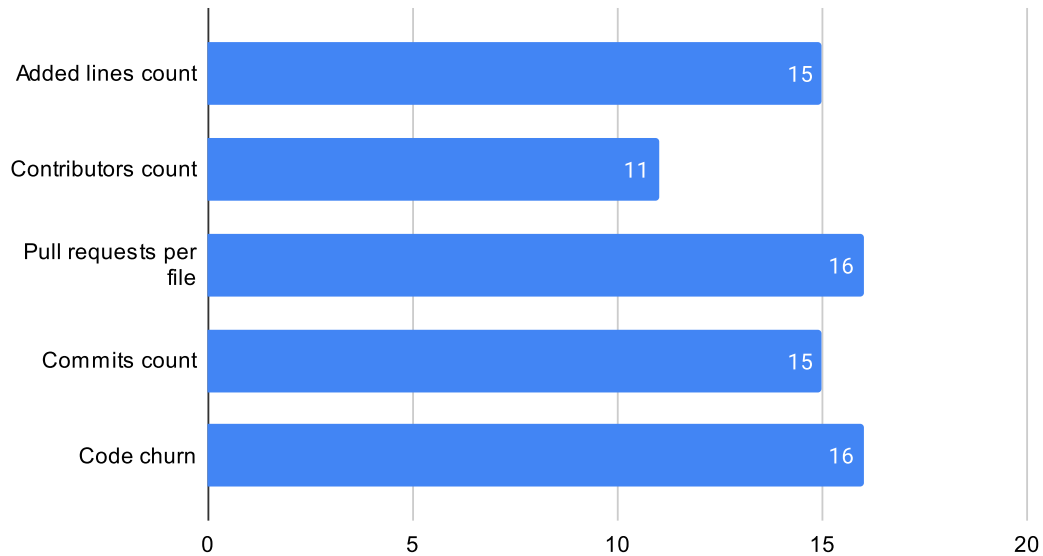


Figure 4.12: Which metrics have developers found useful

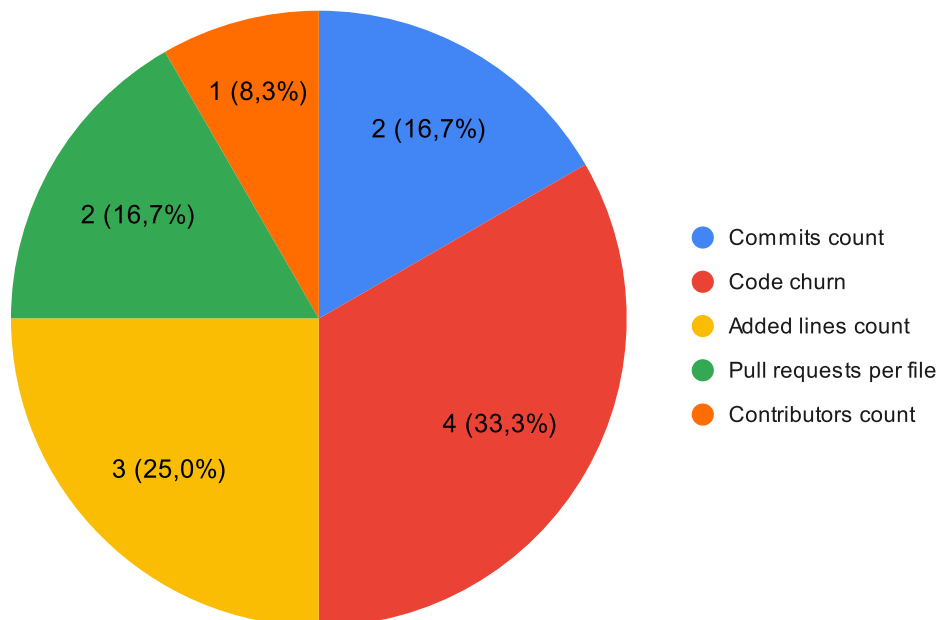


Figure 4.13: Most useful metrics according to the developers

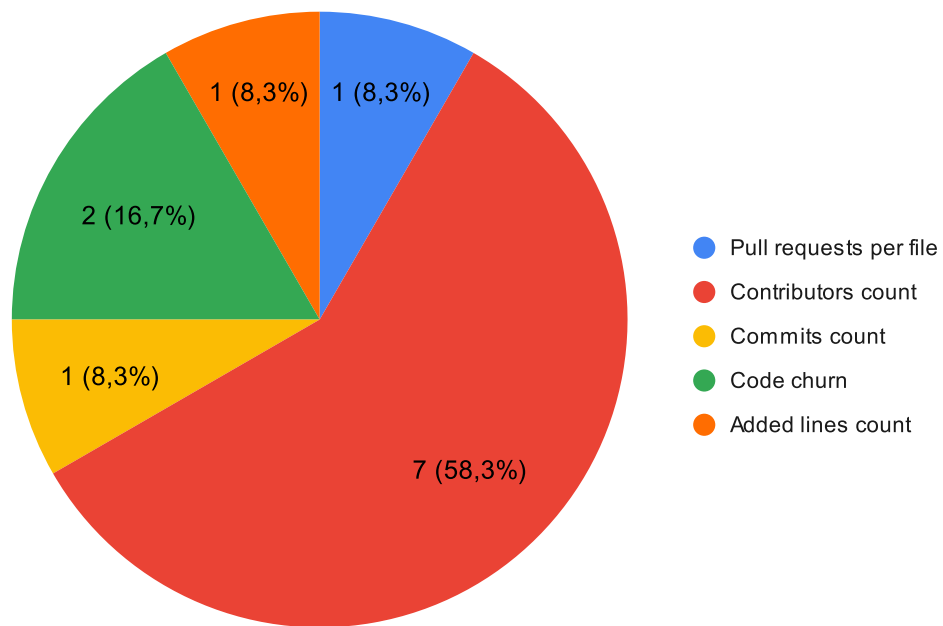


Figure 4.14: Least useful metrics according to the developers

How useful did you found our tool?

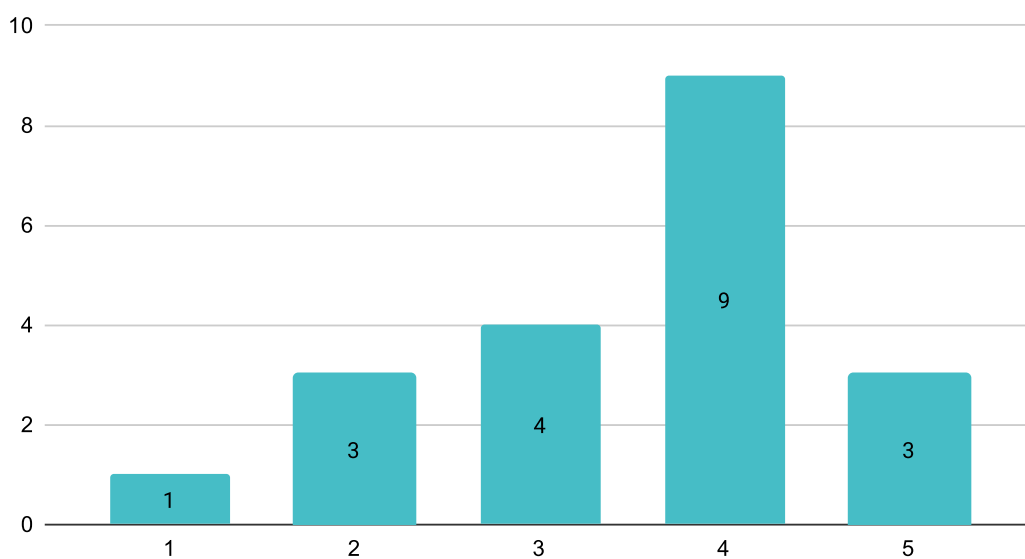


Figure 4.15: "Usefulness" score given by developers

4.3 Performance

In this section, we aim to validate that the execution time of Metrico is reasonable. By reasonable, we mean a time that will not disturb a workflow of a software engineer or a researcher. However, our project is dependent and limited on the GitHub REST API limits described in 3.2. Considering this, we would like to compute data from a period of time of a repository consisting of around 1000 commits and issues in under 30 minutes.

4.3.1 Initialisation

To test the time of execution of Metrico, we did a couple of runs of the initialisation call on two repositories. Once again we will look into the `twbs/bootstrap` and `MaturaIT/maturait_vue` repositories. We want to check the time it takes for Metrico to return data from a `POST/init` call (described here 3.5). We will initialise the data in four different time periods for both repositories. We will describe each run in terms of the number of commits (on the main chosen branch), the number of pull requests and the number of file objects processed during the run.

Time period	Number of commits	Number of pull requests	Number of file objects	Execution time (in seconds)
6 months	417	615	137794	711.32
3 months	204	359	113013	405.28
1 month	61	154	47348	153.61
1 week	17	48	10958	42.72

Table 4.9: Results of a `twbs/bootstrap` repository

As described in 4.2, `twbs/bootstrap` is a repository of a JavaScript framework Bootstrap. We can see in Table 4.9 that it does not have that many commits in the last six months. However, there are a lot of pull requests that we have to process. We can see that the number of file objects is also pretty high. All of this results in a substantial yet expected execution time.

Time period	Number of commits	Number of pull requests	Number of file objects	Execution time (in seconds)
6 months	770	213	73512	504.01
3 months	516	160	66721	342.55
1 month	73	29	2254	39.16
1 week	21	8	177	10.7

Table 4.10: Results of a `MaturaIT/maturait_vue` repository

The `MaturaIT/maturait_vue` repository contains significantly more commits in a given time period than the `twbs/bootstrap` (see Table 4.10 and 4.9). However, the number of pull requests and file objects is smaller. That is why ultimately, all of the execution times in given time periods are smaller than in the previous codebase.

4.3.2 Metrics call

The `POST/metrics/` call serves metrics from a given time period to the frontend of our application. However, because the metrics are stored in a database, the response time is almost instant, averaging around 0.01 seconds. Because of that, we can disregard the execution time of metric calls when evaluating the overall performance of our tool.

4.3.3 Summary

As seen in table [4.9](#) and [4.10](#) all of runs of Metrigo ended in under 15 minutes. It is also worth noting that we analysed reasonably active repositories in substantial time frames, such as six months. Taking all of this into consideration, we can say that Metrigo can produce results in a reasonable time.

Chapter 5

Ethical discussion

In general, Metrico does not involve any significant ethical issues. However, as in the development of any tool, we had to consider some of the data and privacy concerns.

Our tool is designed to be downloaded and launched by a software developer locally. In that case, all of the potential user's data would be stored and processed on their machine. As there is no external server, the possibility of data leaks is minimal and dependent only on the user.

On top of that, users of Metrico do not have to input any personal information besides their GitHub personal access token. However, this is only used to query the GitHub API and is discarded immediately after. Metrico does not store personal tokens in the database. Hence there should not be any possible way to leak the personal access token of a user.

The type of data obtained from GitHub Rest API and stored in the database is data that the user has access to, as we are using their personal access token. That being said, there might be a case when two developers would work on the same machine with Metrico installed, and only one of them had access to a particular project. However, with every call received by the backend of our tool, we check if a user with a given personal access token can access the repository it wants. In that way, there is no possibility of somebody with a token without access to a specific repository data obtaining this data from Metrico.

During the evaluation process, we conducted surveys, which could entail some privacy concerns. However, we did not ask about personal information, and the surveys were submitted anonymously.

Chapter 6

Conclusion

In this project, we developed Metrico, a tool that enables researchers and software developers to explore visualisations of code metrics and their relation to risk management. In this chapter, we will summarise our contributions, findings and discuss some future work that can be undertaken to improve Metrico.

6.1 Contributions and findings

The main contribution of our project is Metrico. As part of this tool we developed:

1. **Visualisation of code metrics:** Metrico is a tool that focuses on generating visualisations of code metrics. The visualisations change depending on picked metrics and time period.
2. **System of calculating code metrics based on repository data:** Metrico uses GitHub REST API to obtain repository data and calculates code metrics based on commits and pull requests data. Right now, Metrico calculates five metrics, but it is fairly easy to extend it to support more.
3. **Database system:** We developed the database that stores our commit and pull request data, as well as calculated code metrics. That way, after the initialisation, we can serve the data to the frontend almost instantly.

During our evaluation, we validated that Metrico can be used to create various visualisations that can represent the code metrics calculated on repository data. We also ensured that researchers can use Metrico to explore code metrics by making our tool relatively easy to extend.

By conducting a small study, we have shown that Metrico can be used to investigate the visualisations of code metrics. We presented developers with four tasks and asked them to solve them using Metrico initialised with particular repositories. We wanted to see what metrics would the developers use to create the visualisations and if they could identify the areas of interest for given tasks using those metrics. We had some success, for example, we found that most of the developers have preferred to use Added line count and Code Churn to identify areas of the codebase that might

need refactoring (see 4.2).

During the study, we also asked developers about their opinion on the code metrics we used. As a result of that, we validated that Metrico can be used as a platform to test metrics. We also ranked the usefulness of our metrics against the tasks that we gave out in the survey. Contributors count has been identified as the least useful and Code Churn as the most useful (see 4.2).

We made sure that Metrico has a reasonable execution time for substantial time periods in repositories. As shown in 4.3, all of the execution times for our tests were under 15 minutes.

6.2 Future work

6.2.1 Solving GitHub API dependency

Right now, our tool is severely limited by GitHub Rest API call limits (see 3.2). To combat this, we created a database system to allow for downloading data from significant time periods in parts. However, this is not an ideal solution, and some improvements could be made in this area. For example, we could release Metrico as a GitHub App(34) and continuously integrate it with a repository so that it performs calculations on every commit.

6.2.2 Pull requests

As evident in 4.3 pull requests involve a vast number of file objects to analyse, which significantly worsens the performance of Metrico. To resolve this, we should rethink how metrics involving pull requests are calculated and possibly give an option to the user not to calculate them.

6.2.3 More visualisations

Currently, Metrico only uses one type of chart to perform visualisations - circle packing. However, adding more charts for users to choose from could be beneficial.

6.2.4 Add filters

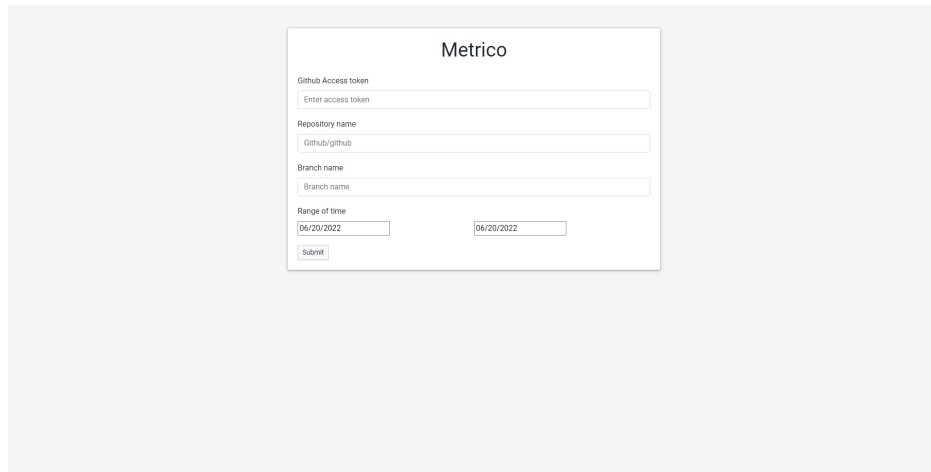
Another feature that could benefit Metrico could be adding filters. For example, we could have filters that enable researchers to check metrics and data for a specific contributor. That way, we could isolate more data and make it easier for our users to extract insights from our visualisations.

6.2.5 Further studies

Finally, it would be beneficial to validate Metrigo with a study on larger scale, as we were able to survey only 20 developers. A study on a larger scale could explore and investigate more relations between code metrics and risk management.

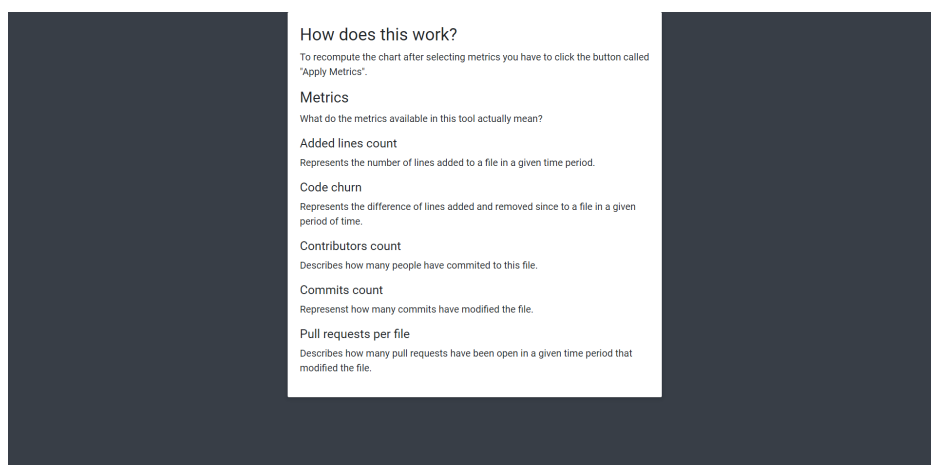
Appendix A

Metrico pages screenshots



The screenshot shows the 'Metrico' starting page. It features a central white box with the title 'Metrico' at the top. Below the title are several input fields: 'Github Access token' with a placeholder 'Enter access token', 'Repository name' with a placeholder 'Github/github', and 'Branch name' with a placeholder 'Branch name'. There is also a 'Range of time' section with two date pickers, one showing '06/20/2022'. At the bottom of the box is a 'Submit' button.

Figure A.1: Starting page of Metrico



The screenshot shows the 'Help' page of Metrico. It has a dark blue background with a white central box containing text. The text is organized into sections: 'How does this work?' with a brief explanation, 'Metrics' with a question 'What do the metrics available in this tool actually mean?', and a list of metrics with their descriptions: 'Added lines count' (Represents the number of lines added to a file in a given time period.), 'Code churn' (Represents the difference of lines added and removed since to a file in a given period of time.), 'Contributors count' (Describes how many people have committed to this file.), 'Commits count' (Represent how many commits have modified the file.), and 'Pull requests per file' (Describes how many pull requests have been open in a given time period that modified the file.).

Figure A.2: Help page of Metrico

Appendix B

Complete results of a survey

Complete results of our study available here [link!](#)

Bibliography

- [1] Project Management Institute. (2017). A guide to the Project Management Body of Knowledge (PMBOK guide) (6th ed.). Project Management Institute. pages 9
- [2] Chawan, P. M., Patil, J., Naik, R. (2013). Software risk management. International Journal of Computer Science and Mobile Computing, 2(5), 60-66. pages
- [3] Ramač, R., Mandić, V., Taušan, N., Rios, N., Freire, S., Pérez, B., ... Spinola, R. (2022). Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry. Journal of Systems and Software, 184, 111114. pages 10
- [4] T. Besker, A. Martini J. Bosch. (2019). Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work, Journal of Systems and Software 156 41–61. pages 10
- [5] McConnell, S. (2008). Managing technical debt. Construx Software Builders, Inc, 1-14. pages 10
- [6] Lefever, J., Cai, Y., Cervantes, H., Kazman, R., Fang, H. (2021, May). On the Lack of Consensus Among Technical Debt Detection Tools. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 121-130). IEEE. pages 10
- [7] Xavier, Laerte Ferreira, Fabio Brito, Rodrigo Valente, Marco. (2020). Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. 10.1145/3379597.3387459. pages 10
- [8] Dai, K., Kruchten, P.B. (2017). Detecting Technical Debt through Issue Trackers. QuASoQ@APSEC. pages 10
- [9] Mayer, P., Bauer, A. (2015, April). An empirical analysis of the utilization of multiple programming languages in open source projects. In Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (pp. 1-10). pages 11
- [10] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., Kirda, E. (2018). Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. arXiv preprint arXiv:1811.00918. pages 11

- [11] Arcelli Fontana, Francesca Zanoni, Marco Ranchetti, Andrea Ranchetti, Davide. (2013). Software Clone Detection and Refactoring. ISRN Software Engineering. 2013. 10.1155/2013/129437. pages 11
- [12] SonarQube, (Accessed 27 January 2022). <https://www.sonarqube.org/> pages 9, 10
- [13] Cai, Y., Kazman, R. (2019, May). DV8: automated architecture analysis tool suites. In 2019 IEEE/ACM international conference on technical debt (TechDebt) (pp. 53-54). IEEE. pages 9, 10
- [14] Sharma, Tushar. (2016). Designite - A Software Design Quality Assessment Tool (2.5.10). Zenodo. <https://doi.org/10.5281/zenodo.2566832> pages 9
- [15] Structure101 (Accessed 27 January 2022) <https://structure101.com/> pages 9, 10
- [16] López-Martín, C. (2019, September). Software Defect Density Analysis. In Proceedings of 28th International Conference (Vol. 64, pp. 139-147). pages 12
- [17] Nagappan, N., Ball, T. (2005, May). Use of relative code churn measures to predict system defect density. In Proceedings of the 27th international conference on Software engineering (pp. 284-292). pages 11
- [18] Omri, S., Sinz, C., Montag, P. (2019). An enhanced fault prediction model for embedded software based on code churn, complexity metrics, and static analysis results. ICSEA 2019, 189. pages 11
- [19] McCabe, T., Meqsure, A. C. (1976). IEEE Tran. On Software Engineering, 2(4), 308-320. pages 12
- [20] Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P. (2016). Cyclomatic complexity. IEEE software, 33(6), 27-29. pages 12
- [21] Vinju, Jurgen Godfrey, Michael. (2012). What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric. 154-163. 10.1109/SCAM.2012.17. pages 12
- [22] Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. Software Engineering Journal, 3(2), 30-36. pages 12
- [23] Herraiz, I., Hassan, A. E. (2010). Beyond lines of code: Do we need more complexity metrics. Making software: what really works, and why we believe it, 125-141. pages 12
- [24] Chidamber, S. R., Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on software engineering, 20(6), 476-493. pages 12
- [25] Shatnawi, R. (2010). A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems (IEEE Transactions on Software Engineering, Vol. 36, No. 2). pages 12

- [26] Malaiya, Y. K., Li, N., Bieman, J., Karcich, R., Skibbe, B. (1994, November). The relationship between test coverage and reliability. In Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering (pp. 186-195). IEEE. pages 12
- [27] Zhu, H., Hall, P. A., May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4), 366-427. pages 12
- [28] Wettel, R. (2008) CodeCity. Retrieved from <https://wettel.github.io/codcity.html> pages 4, 14
- [29] Gource (2019). Retrieved from <https://gource.io/> pages 4, 13
- [30] Brito, Rodrigo Brito, Aline Brito, Gleison Valente, Marco. (2019). GoCity: Code City for Go. 10.1109/SANER.2019.8668008. pages 4, 14, 15
- [31] Spadini, D. (2018). PyDriller [Computer software]. Retrieved from <https://pydriller.readthedocs.io/en/latest/> pages 16, 17
- [32] GitHub RESTAPI (Accessed 18 June 2022) <https://docs.github.com/en/rest> pages 17
- [33] Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P. W. (2019). Scaling static analyses at Facebook. *Communications of the ACM*, 62(8), 62-70. pages
- [34] Github docs. About apps. (Accessed 19 June 2022) <https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps> pages 47
- [35] Github Next (Accessed 19 June 2022) <https://githubnext.com/projects/repo-visualisation/> pages 4, 14, 15
- [36] Winters, T., Manshreck, T., Wright, H. (2020). Software engineering at Google: Lessons learned from programming over time pages 6
- [37] Mike Bostock. Zoomable Circle Packing. (Accessed 18 June 2022) <https://observablehq.com/@d3/zoomable-circle-packing> pages 28
- [38] MaturaIT (Accessed 19 June 2022) <https://www.maturait.pl/> pages 32
- [39] FastAPI (Accessed 19 June 2022) <https://fastapi.tiangolo.com/> pages 19
- [40] React Router (Accessed 19 June 2022) <https://reactrouter.com/docs/en/v6> pages 20
- [41] Redux (Accessed 19 June 2022) <https://redux.js.org/> pages 20
- [42] D3 (Accessed 19 June 2022) <https://d3js.org/> pages 20
- [43] React-Redux (Accessed 19 June 2022) <https://react-redux.js.org/> pages 20
- [44] Unicorn (Accessed 19 June 2022) <https://www.unicorn.org/> pages 19

- [45] SQL Alchemy (Accessed 19 June 2022) <https://www.sqlalchemy.org/> pages 19
- [46] SQL Lite (Accessed 19 June 2022) <https://www.sqlite.org/index.html> pages 19
- [47] React (Accessed 19 June 2022) <https://reactjs.org/> pages 20
- [48] Bootstrap (Accessed 19 June 2022) <https://getbootstrap.com/> pages 20
- [49] Blueprint (Accessed 19 June 2022) <https://blueprintjs.com/docs/> pages 20
- [50] Stack Overflow. Stack Overflow Developer Survey 2021 (2021) <https://insights.stackoverflow.com/survey/2021technology> pages 20
- [51] PyGitHub (Accessed 19 June 2022) <https://pygithub.readthedocs.io/en/latest/index.html> pages 21
- [52] Vue.js (Accessed 19 June 2022) <https://vuejs.org/> pages 20
- [53] Recharts (Accessed 19 June 2022) <https://recharts.org/en-US/> pages 20
- [54] React Charts (Accessed 19 June 2022) <https://react-charts.tanstack.com/> pages 20
- [55] OpenAPI (Accessed 19 June 2022) <https://swagger.io/specification/> pages 19