TU WIEN Informatics

A Lightweight and Integrated Software Repository Mining and Visualisation Approach for Software Engineering Education

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Matthias Weiß
Matrikelnummer 01627775

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Thomas Grechenig

Wien, 6. Mai 2022

Unterschrift Verfasser	Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

A Lightweight and Integrated Software Repository Mining and Visualisation Approach for Software Engineering Education

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Matthias Weiß
Registration Number 01627775

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, May 6, 2022

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

A Lightweight and Integrated Software Repository Mining and Visualisation Approach for Software Engineering Education

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Matthias Weiß
Matrikelnummer 01627775

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 6. Mai 2022

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Weiß

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Mai 2022

Matthias Weiß

Kurzfassung

Ein bekanntes Problem von Softwareprojekten ist, dass es für Außenstehende oft schwer ist, den aktuellen Fortschritt sowie die Arbeitsverteilung einzusehen. Das gilt insbesondere für Projekte in der Software-Engineering-Lehre, da Lehrpersonen üblicherweise mehrere Projekte gleichzeitig betreuen.

Eine Möglichkeit Softwareprojekte transparenter zu machen ist der Einsatz von Softwarevisualisierungen, welche aber in der Praxis noch nicht weit verbreitet sind. Um den Nutzen solcher Tools festzustellen, wird das Forschungsprojekt Binocular[1] im akademischen Kontext evaluiert. Im Zuge dieser Evaluierung werden ebenfalls die Informationsbedürfnisse in der Software-Engineering-Lehre erhoben. Ein Grund, warum sich Softwarevisualisierungen noch nicht in der breiten Masse etabliert haben, könnte deren hohe Eintrittsschwelle sein. Praktisch alle beliebten Tools benötigen entweder eine externe Infrastruktur oder zusätzliche Integration, bevor sie einsatzbereit sind. Um diesem Problem entgegenzuwirken, und um Softwareprojekte im akademischen Kontext transparenter zu machen, wird ein integriertes und kostengünstiges Lösungskonzept für Software Repository Mining und Softwarevisualisierung vorgestellt.

Im Rahmen der Evaluierung von Binocular wurden vier semistrukturierte Interviews mit Expert_innen der Software-Engineering-Lehre durchgeführt. Teil dieser Evaluierung ist ebenfalls die Erhebung der Informationsbedürfnisse in der Software-Engineering-Lehre. Diese werden in der Literatur, trotz zahlreicher Forschungsbeiträge zu den Informationsbedürfnissen im Software Engineering im Allgemeinen, fast vollständig vernachlässigt.

Um die Anforderungen für das Lösungskonzept zu definieren, werden Hypothesen einer geeigneten Architektur für prozessinternes Software Repository Mining und Softwarevisualisierung aufgestellt, welche wiederum von Defiziten existierender Lösungen abgeleitet werden. Durch die Umsetzung dieser Anforderungen im Proof of Concept wird die Realisierbarkeit des Lösungskonzepts gezeigt. Die Validität der Hypothesen, als auch die Zweckmäßigkeit des Lösungskonzepts für prozessinternes Software Repository Mining und Softwarevisualisierung, werden im Rahmen einer Evaluierung mit sechs Expert_innen des Forschungsbereichs bestätigt.

Keywords: *Software Repository Mining, Softwarevisualisierung, Software-Engineering-Lehre, Softwarearchitektur, Softwaremetriken*

[1] `https://github.com/INSO-TUWien/Binocular`, zuletzt aufgerufen am 06.05.2022

Abstract

Software projects are often intransparent to stakeholders who are not directly involved in the development. Student projects are especially challenging, since course instructors and tutors often have to supervise multiple, often identical, projects simultaneously. To address this issue, the research project Binocular[2] is evaluated in an educational software engineering setting. As part of this evaluation, the information needs specific to software engineering education are identified. Moreover, software repository mining and visualisation tools have not been adopted by the mainstream. One reason for this could be that practically all popular solutions either require external infrastructure or additional integration before they become applicable. To address this issue an integrated, low-cost software repository mining and visualisation solution is introduced. It aims at making software projects, especially in an educational context, more transparent for all stakeholders.

The evaluation of Binocular through four semi-structured interviews with experts of the field illustrates its limitations for use in software engineering education. Moreover, although there exists lots of research on information needs in software engineering, none of it is specific to software engineering education. These information needs are also gathered as part of the interviews.

The requirements for the proposed solution are based on various hypotheses about a suitable architecture for in-process software repository mining and visualisation, which in turn have been derived from the shortcomings of existing solutions. These requirements are implemented in a proof of concept, which shows that the proposed architecture is indeed feasible. The evaluation through six experts of the field confirms that the presumed hypotheses are valid and that the proposed solution is purposeful.

Keywords: *Software Repository Mining, Software Visualisation, Software Engineering Education, Software Architecture, Software Metrics*

[2] `https://github.com/INSO-TUWien/Binocular`, last accessed on 06.05.2022

Contents

CHAPTER 1

Introduction

The following chapter serves as an introduction to the thesis. Initially the problem description is illustrated. This is followed by the contributions that were made. Lastly, the structure of the thesis is described.

1.1 Problem Description

Software projects are often non-transparent for stakeholders that are not directly involved in the development. This is especially true for projects in software engineering education, since course instructors and tutors must supervise multiple projects simultaneously. Software engineering education typically is not amongst the most popular research topics as well. For example, even though there exist multiple studies [37, 10, 59, 7, 12] on software engineering information needs, none of them are specific to software engineering education.

Software repository mining and visualisation tools aid in superior understanding of software projects, and thus help in better satisfying the information needs in software engineering education. Current solutions require an external infrastructure and are not integrated into existing software engineering tools, particularly continuous integration (CI), which is ubiquitous in modern software projects. This also holds true for Binocular [25], a research project that visualises the combined, time-oriented data from the version control system (VCS), issue tracking system (ITS) and continuous integration. It is still under active development and currently offers six different visualisations. An integrated architecture and process for offline visualisation artefacts, based on Binocular and using existing software engineering tools, would improve upon the aforementioned issues and better satisfy the information needs in software engineering education. Automatically generated offline artefacts are easily shareable and portable, thus the entry barrier to these artefacts and their associated insights is lowered. Providing a suitable solution to this problem requires building onto the current state of the research areas of software

repository mining, software engineering information needs, software engineering education and visualisations.

Software engineering education deviates from traditional software engineering in a few key aspects. An obvious one is that course instructors and tutors usually supervise multiple, often identical, projects. As a result, the information needs of the various stakeholders differ as well. Identifying these deviations will be beneficial to improve the experience for course instructors, tutors and students. The design and development of the proposed architecture and process, following the successful identification and validation of these information needs, will provide further insights into the software engineering process and quality, for the benefit of all stakeholders. Furthermore, the proposed solution aims to enable continuous, individual insights into the student projects. As a result, the tasks of giving ongoing feedback to students, as well as drawing comparisons between students, is simplified for course instructors and tutors. While the proposed architecture and process will be employable in most software engineering environments, its applicability and impact in an educational context is the focus of this thesis. Formalising these challenges results in the following research questions:

- RQ1: What are the current limitations of MSR tools and software visualisations?
- RQ2:
  1. What information needs exist in software engineering education?
  2. What are the current limitations of Binocular [25] in educational software engineering environments?
- RQ3: What is a suitable process and architecture for in-process software repository mining and visualisation?
- RQ4:
  1. How purposeful do the stakeholders rate the proposed architecture and process?
  2. How do the stakeholders rate its ability to satisfy the information needs in software engineering education?

1.2 Contributions

The contributions of this thesis are twofold. Firstly, an evaluation of Binocular, which aims at identifying its shortcomings, the better understanding of the information needs in software engineering education, is presented. Four semi-structured interviews have been conducted to gain these insights. Secondly, an integrated, low-cost architecture and process for software repository mining and visualisation is introduced. Although this solution has been designed with an education context in mind, it can be applied in any non-education setting as well. In order to be able to define requirements for the

architecture, various hypotheses about a suitable architecture for software repository mining and visualisation had to be constructed. These requirements are educated guesses based on the shortcomings of existing solutions, which have been identified in the literature review. Based on these requirements, a proof of concept has been implemented to show the feasibility of the approach. To conclude, an expert evaluation, which was performed to validate the hypotheses that the architecture is based and to assess the purposefulness of the architecture, is described.

1.3 Structure

Subsequently, Chapter 2 describes the methodologies used. Chapter 3 provides an overview of the related literature. The information needs in software engineering, as well as the semi-structured interviews, which were used to derive these information needs, are presented in Chapter 4. Chapter 5 defines a set of requirements for the proposed architecture. The implementation of the proof of concept is illustrated in Chapter 6. Chapter 7 describes the evaluation of the proposed architecture and process. Lastly, Chapter 8 summarises the results and takes a look at potential future work.

CHAPTER 2

Methodology

The following chapter describes the different methodologies that were used to obtain the results of this thesis. Initially, the literature review, which surveys the current state of the art and aims at finding potential research gaps, is illustrated. Subsequently, the semi-structured interviews, which were conducted to identify the information needs in software engineering education, are mentioned. Afterwards, the requirement analysis for the proposed architecture is depicted. These requirements are realised in a proof of concept, which is described next. The section concludes with the expert evaluation of the proposed solution.

2.1 Literature Review

The research field of this thesis is an overlap of four research fields: information needs (of software engineers), mining software repositories (MSR), software visualisation and software engineering education. To get an overview of the current state of research in these fields a literature review was performed. Since one aim of this thesis is to provide the visualisations of Binocular as offline visualisation artefacts, its original publication [25] served as a good starting point for literature. Moreover, it is noteworthy that the contributions can often be associated with multiple fields. For example, many of the mentioned visualisation tools also perform MSR tasks or vice versa.

Multiple studies on information needs in software engineering are mentioned in the Binocular paper and its corresponding references. These publications are highly relevant to this thesis as well. Moreover, the ACM Digital Library[1] and IEEE Xplore[2] were surveyed to find further publications of this research area. Both platforms offer advanced search functionality, which allows for more complex searches, such as papers containing both the

[1] `https://dl.acm.org/`, last accessed on 02.03.2022

[2] `https://ieeexplore.ieee.org/Xplore/home.jsp`, last accessed on 02.03.2022

keywords "information needs" and "software engineering" in their abstract. Combining multiple keywords into an advanced search was also used to search for publications related to the information needs in software engineering education. Unfortunately, neither of the aforementioned libraries yielded any results.

The Binocular publication also contained various references to MSR platforms, which were used as a basis for further exploration. Furthermore, the GitHub topic for mining software repositories[3] too provided multiple software repository mining (and visualisation) approaches.

Again, the obvious starting point for literature was the Binocular publication. A source for multiple other publications was the IEEE Working Conference on Software Visualisation[4] (VISSOFT). Moreover, several popular non-scientific tools are listed.

The last surveyed research field is software engineering education. The IEEE Conference on Software Engineering Education and Training[5] (CSEE&T) served as solid foundation for publications. The ACM Digital Library and IEEE Xplore were additionally surveyed for further papers of this research area. As mentioned before, explicit searches for contributions related to the information needs in software engineering education were performed.

2.2 Semi-structured Expert Interviews

The goals of the semi-structured expert interviews were to identify the information needs in software engineering education and the current limitations of Binocular in educational software engineering settings. Initially the four interviewees, all of which were involved in the undergraduate software engineering course at TU Wien[6], were asked to rate the usefulness of various insights that were presumed to be beneficial in software engineering education. Moreover, all six visualisations of Binocular were showcased. After a short live demo, the participants were asked to rate the purposefulness of the shown visualisation for their particular role and whether or not they believe it is helpful for software engineering education in general.

2.3 Requirement Analysis

The insights gained through the literature research were used to construct hypotheses of a suitable architecture for in-process software repository mining and visualisation. These hypotheses were then formalised into requirements for the proposed solution.

[3] https://github.com/topics/mining-software-repositories, last accessed on 02.03.2022

[4] https://vissoft.info/, last accessed on 02.03.2022

[5] https://conferences.computer.org/cseet/, last accessed on 02.03.2022

[6] https://www.tuwien.at/en/, last accessed on 04.03.2022

2.4 Proof of Concept

Once the requirements for the proposed architecture were defined, a proof of concept was implemented to show the feasibility of the approach. The proof of concept was implemented using the GitLab[7] platform, since it is utilised in the undergraduate software engineering course at TU Wien.

2.5 Evaluation

To rate the purposefulness of the proposed approach, an expert evaluation with six members of the course staff of the undergraduate software engineering course at TU Wien was performed. Each participant received a short introduction, which included a live demo of several features of the proposed architecture, before answering a questionnaire. This questionnaire contained a section about the hypotheses that were derived during the requirement analysis and how well the final architecture actually meets these requirements.

[7] `https://about.gitlab.com/`, last accessed on 04.03.2022

CHAPTER 3

State of the Art

The following chapter takes a look at the current state of research in the fields of software engineering information needs, software repository mining and software visualisation. Initially some essential concepts are explained to aid in better comprehension. Subsequently, a look at the current state of research of the aforementioned fields is given. Lastly, the contributions of this thesis and how they are distinct from the current state of research are described.

3.1 Definitions

The following section contains definitions of some recurring concepts of this thesis. Knowing and understanding these concepts is essential for comprehending the contributions of this thesis.

Continuous Integration (CI)
Continuous Integration is the practice of automating the compilation, build, tests and deployment of software [29]. Vasilescu et al. [62] state that it is viewed as a paradigm shift in software engineering. They continue that in absence of CI, software is typically considered broken until a testing or integration stage verifies that it is functioning properly.

Issue Tracking System (ITS)
Stakeholders of a software project use issue tracking systems to keep track of the bug reports and feature requests of a software project, the so-called *issues* [5]. It is very common to reference these issues, which typically have a unique identifier, from the VCS.

Mining Software Repositories (MSR)
The field of software repository mining, often called *mining software repositories*,

evolves methods that use the data stored in software repositories (not only from the VCS, but from CI/CD and ITS too) to improve the understanding of software development and its associated practices [28].

Software Repository

Hassan, Holt and Mockus describe the data held in a software repository as the data of the VCS, defect tracking systems and archived communications explaining the rationale for decisions between project stakeholders [28]. In the context of this thesis a software repository is the storage location of a software project and its corresponding data, including its VCS, ITS and CI.

Software Visualisation

De Pauw, Reiss and Stasko [14] describe software visualisation as *"the use of pictures for looking at and understanding software systems"*. Furthermore, they argue that, due to humans being an inherently visual species, it is only natural that we try to create such pictures in order to understand our software systems.

Version Control System (VCS)

Version control systems are used to record the changes of a set of files, typically source code files, over time [11]. It enables users to revert files or even the entire project to a past state, view the changes of given files over time and countless other tasks. Typically, there is a server (or sometimes multiple servers) holding the entire repository, which clients can check out locally. In the context of this thesis the term version control system actually refers to a distributed version control system (DVCS). In a DVCS the clients do not only mirror the repository, but its entire history as well. This is especially helpful if any server dies, since any of the clients can restore the content of the repository on the server.

3.2 Current State of Research

The subject area of this thesis can be described as an intersection of the following fields: information needs (of software engineers), MSR tools, software visualisation and software engineering education. This section first provides a look at the current state of research of these subject areas and subsequently describes how the proposed architecture and process deviates from these approaches.

3.2.1 Information Needs

A better grasp of the information needs of software engineers yields many benefits, such as a better understanding of the decision-making-process. The following section takes a look at several contributions of this field.

Ko et al. [37] derived information needs through observations of typical work of developers. During these roughly 90 minute long sessions, 21 information needs were abstracted from 334 total instances of information seeking. For each of these information needs the time

spent searching, search frequencies, search outcomes (whether the needed information was acquired, deferred with the intention of resuming the search or if the developer gave up with no intention of resuming) and their source frequency were recorded. The most common information needs were "Did I make any mistakes in my new code?" and "What have my coworkers been doing?". Furthermore, the observations show that coworkers were the most common source of information, 13 of the 21 information needs accessed coworkers as a source at least once.

Buse and Zimmermann [10] studied the information needs of 110 developers and managers. One of their findings shows that managers rated data and metrics as the most important factor for decision making. Developers on the other hand rated their personal experience as the most crucial factor, even though they had 6 years less experience on average. They furthermore survey the importance of different artefacts, such as features, bug reports and classes. They infer that all of the artefacts are important and emphasise their simultaneous importance. An example showing the latter would be the need to find the classes, functions and teams connected to the code churn of a project, which a manager may wish to measure. Lastly, the developers and managers were asked about metrics that use to make decisions or would use, if they were available. It is noteworthy that a majority of these metrics, such as code churn or failure models, would have been used by developers and managers, but are not available. Based on the information needs identified in this study, they additionally propose a set of characteristics for software analytics tools, as well as a set of suitable analysis types for the information needs identified in the study.

Tao et al. [59] presented a quantitative and qualitative study that explores the importance of understanding code changes during software development, questions about related information needs and requirements for potential tools that developers may use during this process. The survey comprised a quantitative and a qualitative part. The former rates the importance and difficulty scores for 15 potential information needs that were inferred from research. It is noteworthy that the rationale for a code change, which was rated the highest for importance, also was the most easily obtainable, according to the survey. The qualitative part of the survey, a single open question asking for further information needs related to code changes, showed that current practices deviated from the true needs of developers. Amongst others, the risk of a code change and the quality of a code change did not have proper tool support.

Begel and Zimmermann [7] introduced a set of 145 questions, grouped into 12 categories, that software engineers would like data scientists to explore. An initial survey was sent out to 1,500 software engineers at Microsoft, asking them to list five questions they would like data scientists to answer, as well as why they would want to know that and what they would do with the answers. This was followed by a second survey to rank and prioritise the questions that were collected in the initial survey. 2,500 engineers were invited to the second survey, none of which have been part of the initial one. One interesting finding is that not only the top-rated question, "How do users typically use my application?", is concerned with customers, but in total 8 of the 20 questions with the highest percentage

of "Essential" ratings are.

In [26], Haenni et al. explored the information needs of developers in software ecosystems. A software ecosystem, as described in [41], is *"a collection of software projects which are developed and which co-evolve together in the same environment"*. They discovered that there typically are two perspectives in the context of a software ecosystem. The upstream, usually a framework or library, provides its source code to the downstream. Furthermore, they show that the information needs of upstream and downstream developers differ. The downstream needs typically correspond to the various stages of their relation with an upstream: the selection, adoption and co-evolution with an upstream project. The upstream needs on the other hand can be grouped into two categories: project statistics and code usage. In [27], they confirm these findings through a quantitative survey of 75 software developers and, moreover, conclude that there is no adequate tool support for developers working in the context of a software ecosystem.

Codoban et al. [12] investigated the reasons why software engineers inspect software history, how they do it and what obstacles exist when doing so. After interviewing 14 developers, a survey with 217 participants was conducted to quantify the observations from the interviews. They mention that, even though 61% of survey respondents needed to inspect the history of their software project(s) up to a couple of times a day, software history tools did not provide support for the needs that the developers have. They furthermore found that developers often have needs related to uncommitted changes, but existing tools typically do not take these into account. Lastly, they introduce a novel software history model, namely "3-LENS HISTORY", which provides the foundation to give identity to software history in the future.

Pascarella et al. [47] studied the information required to perform code reviews, as well as to improve the understanding of the code review process. They identified seven high-level information needs of reviewers in code reviews through manual analysis of 900 code review comments from the three open-source projects OpenStack[1] (nowadays called OpenDev), Android[2] and QT[3]. Based on their results they made several suggestions for possible code review tools, such as detection of dividable code changes or real-time communication.

In [32], Josyula et al. aimed at determining the information needs of software practitioners, the sources used to satisfy these information needs, as well as the perceived usefulness of the various sources. They identified that software practitioners have nine types of information needs, that are satisfied using thirteen different types of sources. While the information sources deviate from practitioner to practitioner, some sources (e.g. discussion with colleagues) appeared much more frequently than others (e.g. social networking sites and groups).

Kortum et al. [38] identified the information needs of agile software development teams through a survey with 90 international software engineers. In particular they described

[1] `https://review.opendev.org/q/status:open+-is:wip`, last accessed on 04.12.2021
[2] `https://android-review.googlesource.com/`, last accessed on 04.12.2021
[3] `https://codereview.qt-project.org/`, last accessed on 04.12.2021

which team-related problems (e.g. lacking communication) emerge most frequently amongst the participants, as well as the information usually used to address these issues.

In [1], Ahmad et al. identified the information needs in continuous integration and delivery (CI/CD) through a study with 34 participants from five different companies. They catalogued 27 information needs, categorised into five different groups (testing, code & commit, confidence level, bug, and artefacts) and associated with various stakeholders (developers, testers, project managers, release team, and compliance authority).

3.2.2 Mining Software Repositories (MSR)

Researchers have been interested in the available data of source code repositories ever since they exist [30]. This interest only has increased once distributed version control systems, in particular Git[4], have become increasingly popular [9, 33]. These days, source code repositories are typically hosted on platforms such as GitHub[5] or GitLab. While these platforms offer countless more features and integrations, therefore a lot of additional information can be extracted through the process of mining, there also arise various new pitfalls when looking at this newly available data [34]. Since the amount of information in software repositories is further increasing, the contributions of this field are especially diverse. For example, Luijten et al. [40] extracted information about issue handling efficiency from the GNOME issue database. Moreover, Robles [51] investigated the replicability of published MSR papers. Kovalenko et al. [39] highlight the importance of branches and merge commits when mining software repositories. Gold and Krinke [24] on the other hand, presented various ethical issues that emerge during the process of extracting information from software repositories.

MSR Tools

While there exist various solutions for software repository mining, most of them either require external infrastructure or further integration before they become applicable. The following subsection looks at some popular contributions.

PyDriller [57] is a Python[6] framework designed to extract information from Git repositories in significantly fewer lines of code with less complexity than past solutions. It is a wrapper around the popular GitPython[7] package. While GitPython offers almost all features of Git, PyDriller's features have been narrowed down to the ones which are essential when doing MSR tasks.

Boa [18, 19] is a domain-specific language and infrastructure to simplify the extraction of data from large source code repositories such as GitHub or SourceForge[8]. It not only significantly reduces the necessary programming efforts and majorly improves scalability

[4] `https://git-scm.com/`, last accessed on 04.12.2021
[5] `https://github.com/`, last accessed on 04.12.2021
[6] `https://www.python.org/`, last accessed on 14.09.2021
[7] `https://github.com/gitpython-developers/GitPython`, last accessed on 14.09.2021
[8] `https://sourceforge.net/`, last accessed on 14.09.2021

(without the need of writing explicitly parallelised code), performed experiments are also very straightforward to reproduce by re-running the already existing Boa programs. A summary of Boa's infrastructure is shown in Figure 3.1. While Boa has many upsides, there certainly are some shortcomings as well. To address some of these problems, Hung and Dyer [31] introduced the concept of *materialised views* into Boa's language and infrastructure. A view in the Boa language is similar to the concept of a view in relational databases. Moreover, a materialised view is a static and cached snapshot of a view. Therefore, it is usable in future queries as well as shareable across different users.

In [60], Tiwari et al introduced Candoia, a platform and ecosystem dedicated to building and sharing MSR tools. The Candoia platform makes these tools, which are built as apps, easily portable and highly customisable. Candoia applications consist of four parts: the MSR logic, which is an extension of the Boa language, a JSON file containing the structure description, an HTML and CSS file with the layout for the visualisation and glue code for interactions located in a JavaScript file. Furthermore, the Candoia ecosystem enables sharing of these MSR tools by acting as an app store.

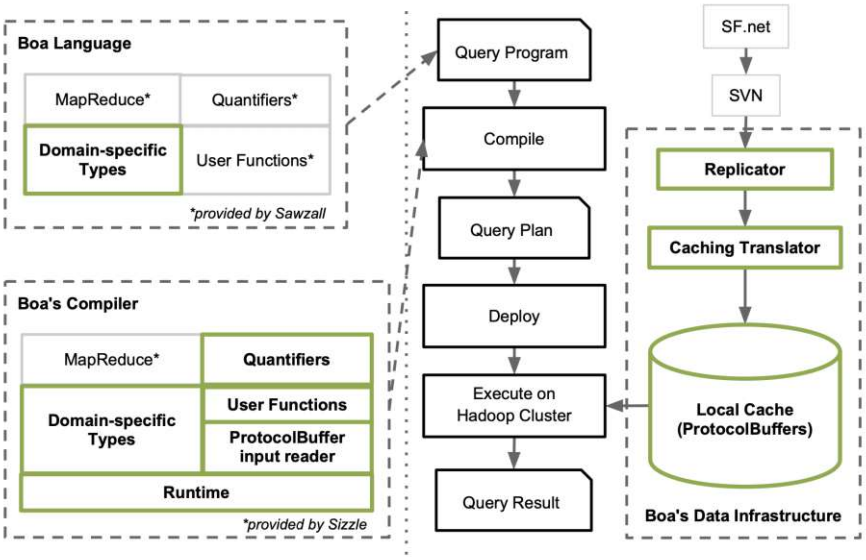


Figure 3.1: Overview of the Boa infrastructure

Crossflow [2] is a framework designed to support the creation of so-called "workflows", which are distributed (and optionally polyglot), multi-step software repository mining programs. The scalability of both parallel and distributed execution shows promising results, which can be attributed to its use of asynchronous, message-based communication, job-level caching and locality scheduling. Workflows can be either be deployed and executed manually or through an accompanying web application, which aids in simplifying these tasks. The architecture of Crossflow is shown in Figure 3.2.

In [50], Reza et al. noted that mining tools are typically tailored towards analysis and

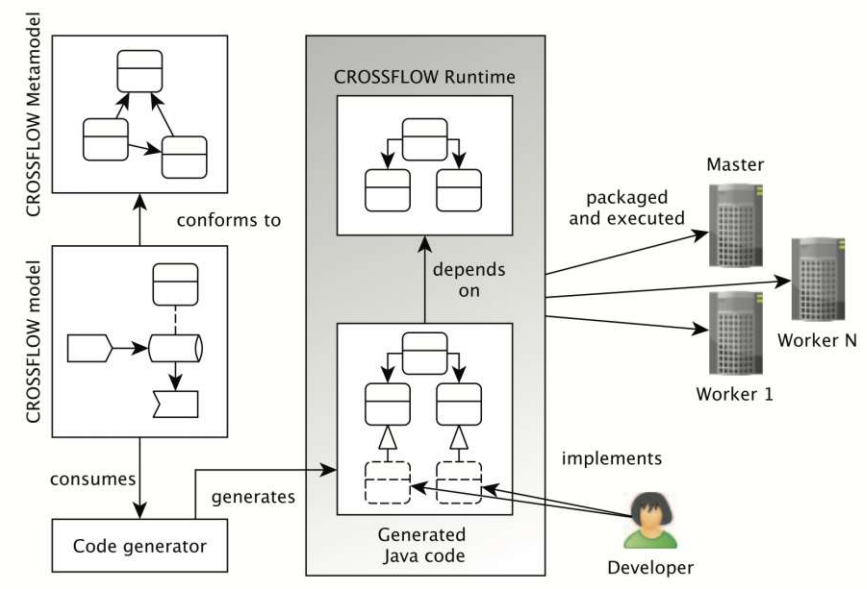


Figure 3.2: Architecture of the Crossflow framework

data extraction of textual artefacts. To address this issue they introduce *ModelMine*, a tool specifically built to mine model-based artefacts and designs. It outperformed the state-of-the-art PyDriller [57] in both performance and usability analyses, while also offering the ability to search repositories, which PyDriller does not.

Müller et al. [45] identified various issues concerning data acquisition in software analysis, such as integration and storage of heterogeneous data. Based on their findings they introduced *jQAssistant* an extendable open-source stack for software analysis and visualisation. Eventually, they showed the applicability of their solution through implementation of a proof of concept, which analyses and visualises JUnit[9].

3.2.3 Software Visualisation

While the field of software visualisation has been ever-present since its inception, some challenges still remain today. In the 2005 paper [49], Reiss described software visualisation as a paradox. While, even back then, there were obvious benefits to the concept, mainstream adoption had not yet succeeded. This still holds true today. In [42], Mattila et al. present the results of a systematic literature review they performed in 2016, which comprised the literature of the previous six years. They identified the adoption of software visualisation tools in the industry as a future research topic. Moreover, it suggests that the adoption in software engineering education too is a viable research topic.

[9] `https://junit.org/`, last accessed on 04.12.2021

Software Visualisation Tools

The following section introduces various software visualisations. While some of these solutions also perform mining tasks, thus would be appropriate in the previous section, all of them offer one or multiple visualisations of information related to software repositories.

Binocular [25] offers visualisations that combine time-oriented data from the VCS, ITS and CI. The project is still under active development, contributions to the project can be seen at the corresponding GitHub repository[10], and has seen some major changes and additions ever since its original publication. It currently offers six interactive visualisations, each of which offering several different options for customisation. All visualisations are described in detail in Section 4.1.2.

In [52], Dal Sasso et al. introduced a blended approach to software visualisation, which combines the data of several sources, with the aim of finding out, what has happened to a software system in the last few days. The three ingredients, as they call the sources, are the source code changes, stack traces, and IDE interaction data. Combining these three ingredients with the *software city* metaphor, a concept originally used in [63], results in the *blended city* visualisation.

Through the introduction of their tool *Hunter*, Dias et al. [16] aimed at easing the process of software comprehension. Their contribution complements the established views of integrated development environments (IDEs), where a list on the left shows the files and folders, in combination with a large text field in the centre displaying the currently selected file. The purposeful visualisations of Hunter, such as the File Dependencies View, enable more complex comprehension tasks of JavaScript applications. The File Dependencies View, which visualises dependencies between JavaScript files, their size, interaction between macro-components, as well as references to external libraries, was the most used pane in Hunter, as determined through analysis of the eye-tracking data of the controlled experiments. When compared to Visual Studio Code[11], Hunter, on average, achieves a higher percentage of correct answers, using fewer editor panels, and in less time on non-trivial tasks such as determining the most invoked JavaScript file.

Moreover, Schreiber et al. [54] extracted and visualised the provenance information of software development projects, to gather insights on software development processes. Provenance in the context of the web, as defined by Moreau and Groth [43], is a computer-processable record containing a description of the events that led to a document or piece of data being in a given state. Provenance can be conceptualised as a graph consisting of nodes, representing the factors that contributed to the state change, and edges, representing their relations. Schreiber et al. visualised the provenance information by integrating a graph visualisation, metrics representation, and development timelines into a web-based dashboard.

Fiechter at al. [21] proposed the concept of an *issue tale*, which is a (visual) representation

[10] `https://github.com/INSO-TUWien/Binocular`, last accessed on 14.09.2021
[11] `https://code.visualstudio.com/`, last accessed on 05.12.2021

of an issue and all of its accompanying actors and events. Moreover, two visualisations of issue tales were introduced in their contribution. The coarse-grained view gives an overview of all issue tales, including their size and duration, for a given project. Furthermore, the fine-grained, timeline-based visualisation illustrates the internal structure of a single issue tale.

In [36], Kim et al. introduced *Coding Time-Machine*, a tool that associates and visualises development tasks and their corresponding elements (e.g. classes, methods). Coding Time-Machine identifies causal relations between development tasks and its element in commits through comparisons with previous versions. It comprises four different views: a list of all commits of the repository, a list of development tasks present in a single commit, a view visualising the causal relations between the elements of a development task, and a diff view for displaying the updated code. This enables the inspection of development tasks, as well as their causal relationships, at any point in time.

Tamer et al. [58] noted that, even though there exist various contributions on the visual analysis of component-based frontend-frameworks, research typically does not regard the specifics of these systems. They address this issue by introducing a tool to visualise code quality metrics and dependencies of React[12] applications. Their approach consists of four panels: a panel displaying various quality metrics (e.g. lines of code, parameter count), a source code panel, a file tree, and a node-link diagram visualising the structure of the project.

SonarQube[13] is an open-source platform for continuous static code analysis. It supports various programming languages and can be extended through plugins. Moreover, SonarQube can not only detect bugs in the source code, but known vulnerabilities as well. It provides thousands of automated rules and can easily be integrated into existing tools like GitLab CI/CD[14] or Jenkins[15].

Gitinspector[16] is a used to analyse Git repositories. It visualises the statistics for each contributor, as well as a timeline illustrating the workload and activity of each author. Gitinspector was originally developed to fetch statistics for student project statistics at a course at Chalmers University of Technology[17] and Gothenburg University[18]. By default only source files are included in the statistics, the list of source file extensions can be customised. The results can be output to HTML, JSON, XML or plain text and multiple languages are supported.

Hercules[19] is a Git repository analysis engine that comprises the two (cohesive) command line tools `hercules` and `labours`. These tools can be used to create highly customised

[12] `https://reactjs.org/`, last accessed on 05.12.2021
[13] `https://www.sonarqube.org/`, last accessed on 05.12.2021
[14] `https://docs.gitlab.com/ee/ci/`, last accessed on 05.12.2021
[15] `https://www.jenkins.io/`, last accessed on 05.12.2021
[16] `https://github.com/ejwa/gitinspector`, last accessed on 04.12.2021
[17] `https://www.chalmers.se/en/`, last accessed on 05.12.2021
[18] `https://www.gu.se/en`, last accessed on 05.12.2021
[19] `https://github.com/src-d/hercules`, last accessed on 3.12.2021

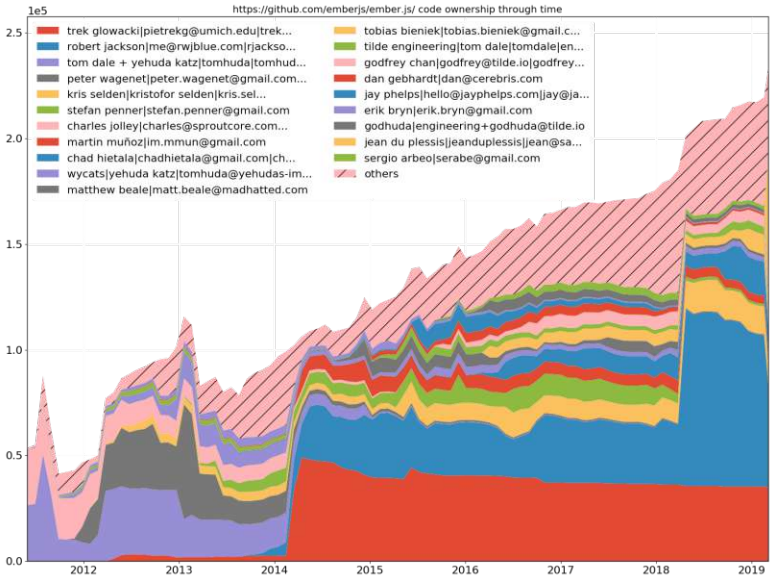


Figure 3.3: Code Ownership of the Ember.js repository[21] repository

analyses and corresponding plots of Git repositories. `hercules` is a program written in Go[20], which extracts data of a given Git repository by performing analysis tasks over the commit history. This data can then be used by `labours`, a Python script, to show various predefined plots. An example of the code ownership diagram can be seen in Figure 3.3.

3.2.4 Software Engineering Education

Researchers have been interested in improving software engineering education, as well as its associated processes, for decades [22, 56, 8, 17, 4]. The Conference on Software Engineering Education and Training[22] (CSEE&T), for example, dates back to the late 1980s. The following section highlights various contributions from different sub-fields of the software engineering education literature.

Xie et al. [65] described software engineering education as an intersection of software engineering, education and gaming. They argue, that, for educational tasks, gaming technologies frequently play a vital role alongside software engineering technologies. These gaming technologies, often educational games, can further be extended to dedicated events, such as Hackathons, which have seen quite some popularity in the literature [48, 23].

Aligning software engineering education with industry practices has been an ongoing

[20] `https://golang.org/`, last accessed on 14.09.2021
[21] `https://github.com/emberjs/ember.js`, last accessed on 11.10.2021
[22] `https://conferences.computer.org/cseet/`, last accessed on 03.02.2022

challenge for decades [6, 64]. Vanhanen et al. [61] described how software engineering is taught through real-world projects at Aalto University[23]. Teams of seven to ten students, guided by a mentor, implement projects for real customers. Students consistently rank the course amongst the best of the computer science curriculum, even though course feedback suggests that the required effort per credit unit is higher than most other courses. Dagnino [13] introduced a method to improve the graduate software engineering course at NC State University[24], by focusing on how software engineering theory applies to practice and through emphasis on industry experiences. The paper introduces eleven non-traditional elements of software engineering curricula, which were gradually implemented in the course. Incorporating these techniques has resulted in an improvement in the course evaluation performed by the students. Devadiga [15] illustrated existing gaps in software engineering education when it comes to startups. Due to the limited resources, software engineers working in startups often take up multiple roles. Therefore, a wide range of skills, including testing, deployment, cloud computing and DevOps, amongst others, are required. Shortcomings of current software engineering curricula, as identified in the paper, are application design, code structure and testing of distributed systems. Eddy et al. [20] presented a study on introducing CI/CD into an undergraduate software engineering course. An example pipeline was used to help instructors establish the concepts of continuous integration and delivery in existing courses. Furthermore, it allowed students to visualise and understand the accompanying processes. The study not only showed that their understanding of the studied concepts (e.g. version control, branching, static code analysis) improved, but also that the CI/CD pipeline was useful in understanding the concepts.

Moreover, there are various contributions that aim at improving software engineering courses at universities. Sedelmaier and Landes [55] introduced an active and inductive learning approach for software engineering, focusing on understanding the need of software engineering and its associated methods and techniques. Nytrø et al. [46]. compared two different pedagogical approaches, one of which emphasises systematic guidance and education, while the other one focuses on innovation and inspiration, for an undergraduate software engineering course.

Lastly, since distance learning (and therefore also distance teaching), due to COVID-19, has become as widely spread as ever, contributions addressing its accompanying challenges have also become increasingly popular [3, 35, 53, 44].

3.3 Distinction From Current Research

One aim of this thesis is to identify the information needs in software engineering education. Even though there exist multiple studies on information needs in software engineering, as mentioned in Section 3.2.1, and the field of software engineering education

[23] `https://www.aalto.fi/en`, last accessed on 10.12.2021
[24] `https://www.ncsu.edu/`, last accessed on 10.12.2021

is very substantial, as shown in Section 3.2.4, there currently are no studies on the information needs specific to software engineering education.

Once these information needs are collected, a suitable process and architecture for in-process software mining and visualisation will be implemented. Current MSR tools and visualisations typically either require external infrastructure or additional integration before they become applicable. The contribution of this thesis is a lightweight and integrated architecture and process for offline visualisation artefacts using existing software engineering tools (e.g. GitLab, Git). The proposed solution will deliver the visualisations of Binocular [25], which typically requires an external infrastructure, as offline artefacts through the integration of Binocular into existing and established software engineering tools, GitLab CI/CD in particular. Lastly, the proposed architecture and process is used to better satisfy the information needs in software engineering education.

CHAPTER 4

Information Needs in Software Engineering Education

The following section illustrates the information needs in software engineering education. In the beginning, the semi-structured interviews performed to determine these information needs are described. Subsequently, the data gathered in these interviews is used to derive the information needs in software engineering education.

4.1 Semi-Structured Expert Interviews

While there exist multiple studies on software engineering information needs, as described in Section 3.2.1, none of them are specific to software engineering education. To gain insights on this topic, as well as to find out the shortcomings of Binocular [25] in educational software engineering settings, four semi-structured expert interviews were conducted. At the time of writing, all of the participants were involved in the undergraduate software engineering course at TU Wien. Furthermore, in order to achieve the most objective results, it was made sure that the roles of the participants in the given course varied. The interviews were held remotely via Zoom[1] and lasted approximately 90 minutes each.

4.1.1 Interview Design

The interviews were guided by a questionnaire consisting of nine sections and a mixture of quantitative and qualitative questions. All quantitative questions used a 5-point scale to rate the usefulness/helpfulness/purposefulness of a given concept, ranging from 1 (not useful/helpful/purposeful) to 5 (very useful/helpful/purposeful).

[1] https://zoom.us/, last accessed on 07.01.2022

Demographics

The opening section contains demographic questions. Initially, the participants are asked about their age. The possible answers start from 18 years old and, apart from the first and last bucket, which are "18-24 years old" and "65 years or older" respectively, are grouped into buckets of 10 years, i.e. "25-34 years old". Subsequently, a question about gender of the participant follows. The possible options here are "Female", "Male", "Prefer not to say" and "Other", a free text option called to add any gender the participant may identify with. The third question aims at determining how long the interviewees have been working in software engineering education. The options are illustrated in Figure 4.1. Lastly, an open question asks which role the participants are performing in the undergraduate software engineering course at TU Wien. This information is vital in identifying potential differences in information needs between the various roles.

Figure 4.1: Question #3 and its possible options

Concepts

Up next were questions about the usefulness of several insights that were considered valuable after reviewing the literature (e.g. code ownership over time). This section serves as the main source for the identification of the information needs in software engineering education. Therefore, the information derived from this section lays the foundation to the answer of RQ2.1. The insights that are described in this section either are insights that Binocular already aims at providing, or aims at providing in the future (e.g. through additional visualisations). How well Binocular actually provides these insights will also be evaluated during the interviews.

Initially, the participants are asked to rate the usefulness of insights on large spikes in work contributions in software engineering education. Binocular, in particular its Dashboard and HotspotDials visualisations, both of which will be described later in this section, already aims at providing this insight.

The interviewees are then asked about the usefulness of insights on work distribution between students in software engineering education. Again, Binocular already aims at providing this information through its Dashboard visualisation.

The next two questions aim at determining the usefulness of insights on the number of active conflicts, as well as the history of conflicts, within the source code. These insights are currently not provided by Binocular, but are considered for future work.

Lastly, the participants are asked to rate the usefulness of insights on code ownership over time by each student. Binocular, in particular its Code Ownership River visualisation, already aims at providing this information.

Dashboard

Subsequently, one section for each of the six visualisations of Binocular follows. Before the participants get to answer these questions, a live demo of the given visualisation was performed.

The first of the six sections was about the Dashboard visualisation. It gives an overview of the number of code changes, the number of open and closed issues as well as the number of succeeded and failed CI/CD builds of a given time frame (year, month, week or day). The "Changes" chart has an option to show or hide individual contributors. Moreover, the changes can be shown as the number of changed lines (split into additions and deletions) or the number of commits. The "Issues" chart has an option to either show the number of open, closed or total issues. An example of the Dashboard visualisation can be seen in Figure 4.2. Hereafter, all of the screenshots of Binocular display data from the Binocular project[2] itself, as well as its associated ITS and CI/CD data. The state of the repository illustrated in these screenshots is stored in a Docker image made available on the Dockerhub[3]

The section about the Dashboard visualisations opens with the following five questions:

- How purposeful is the Dashboard visualisation for your role in this course?
- How could it be improved to ease your role?
- How helpful is the Dashboard visualisation for software engineering education?
- How can this visualisation be utilised in software engineering education?
- What are its shortcomings in software engineering education?

It should be noted that all sections about the visualisations of Binocular begin with similar questions. The aim of these questions is to identify the limitations of Binocular in an educational software engineering setting, which corresponds to the answer to RQ2.2.

[2] `https://github.com/INSO-TUWien/Binocular`, last accessed on 20.01.2022

[3] `https://hub.docker.com/layers/insotuwien/binocular-database/develop-1c03d823/images/sha256-1bfd1f162f8dab1fba574aa7a8b8fff2df9060174981957057a482e53d48f095`, last accessed on 09.03.2022

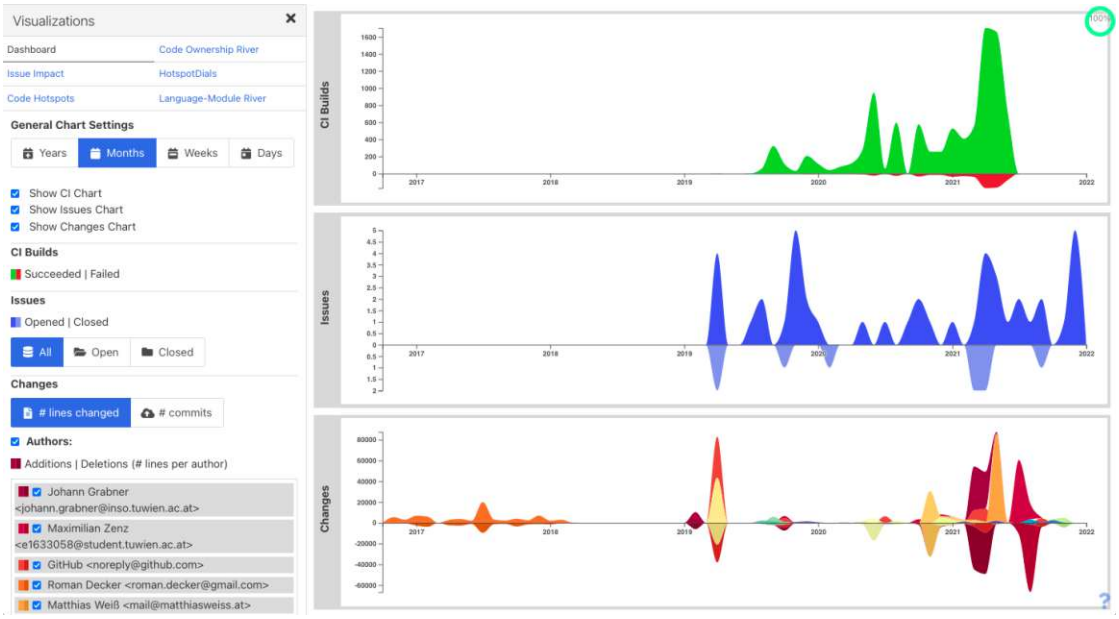


Figure 4.2: Example of the Dashboard visualisation

The section about the Dashboard visualisation contains six additional questions. These questions should be seen as groups of two related questions, since the follow-up question is optional. These questions also aim at answering RQ2.2 and look as follows:

- How purposeful is the dashboard visualisation to gain insights about large spikes in work contributions?
  - How could it be improved to gain the relevant insights?
- How purposeful is the Dashboard visualisation to gain insights about work distribution between students?
  - How could it be improved to gain the relevant insights?
- How purposeful is the Dashboard visualisation to gain insights about the amount of added/deleted source code by each student?
  - How could it be improved to gain the relevant insights?

Code Ownership River

The interviews continue with the section regarding the "Code Ownership River" visualisation. Depending on the setting, the Code Ownership River either displays the aggregated lines of code over time or the code ownership over time. Moreover, there is the possibility to select one of three different overlays. These overlays either highlight an issue and its

accompanying commits, the open and closed issues over time, or the successful and failed CI/CD builds over time. A screenshot of the Code Ownership River visualisation, in this case illustrating the code ownership over time in conjunction with the open and closed issues over time, is shown in Figure 4.3.

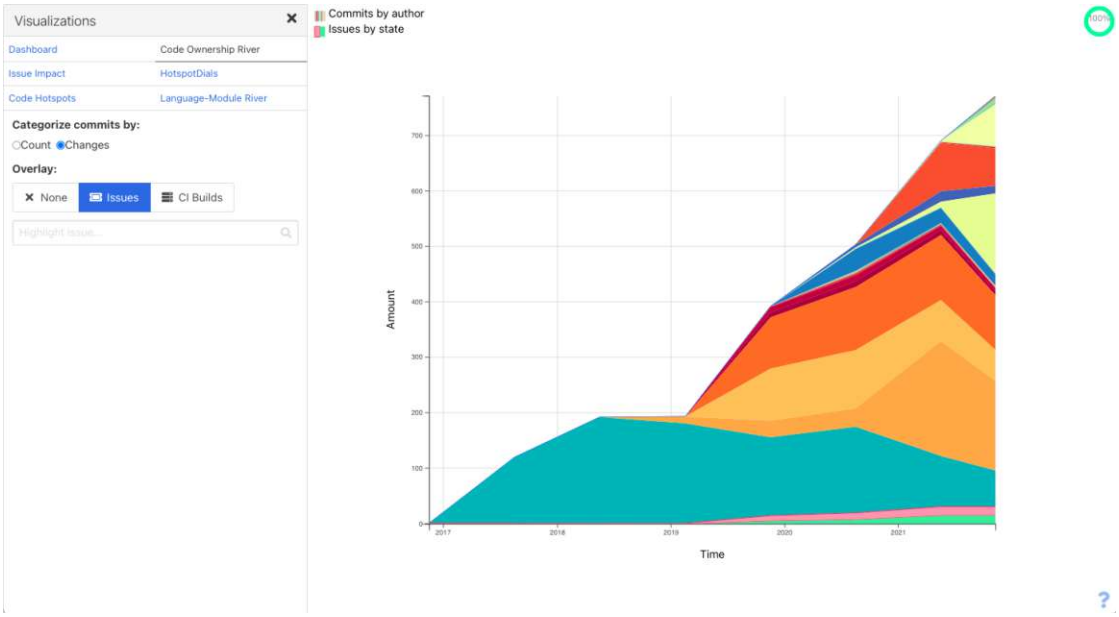


Figure 4.3: Example of the "Code Ownership River" visualisation

Apart from the five questions that exist for every visualisation, the section about the Code Ownership River visualisation does not contain any additional questions. The purpose of this visualisation, as well as its ability to gain insights on the code ownership over time by each student, is obvious. Since there are no additional questions, the section looks as follows:

- How purposeful is the Code Ownership River visualisation for your role in this course?
- How could it be improved to ease your role?
- How helpful is the Code Ownership River visualisation for software engineering education?
- How can this visualisation be utilised in software engineering education?
- What are its shortcomings in software engineering education?

Issue Impact Visualisation

Next up, the section about the "Issue Impact" visualisation follows. This visualisation focuses on a particular issue, thus it is much more fine-grained than the previous two. For

a given issue, it lists its corresponding commits on the horizontal time axis in the middle of the circle. The top half of the circle shows the files a given commit has changed, the size of the arcs represents the amount of changes. The bottom half of the circle shows the contributors for each commit. If CI/CD data for a given commit exist, the arc is coloured in green or red, depending on the pipeline status. Moreover, the commits and files to be shown in the visualisation can be customised through filters. Figure 4.4 shows an example of the Issue Impact visualisation.

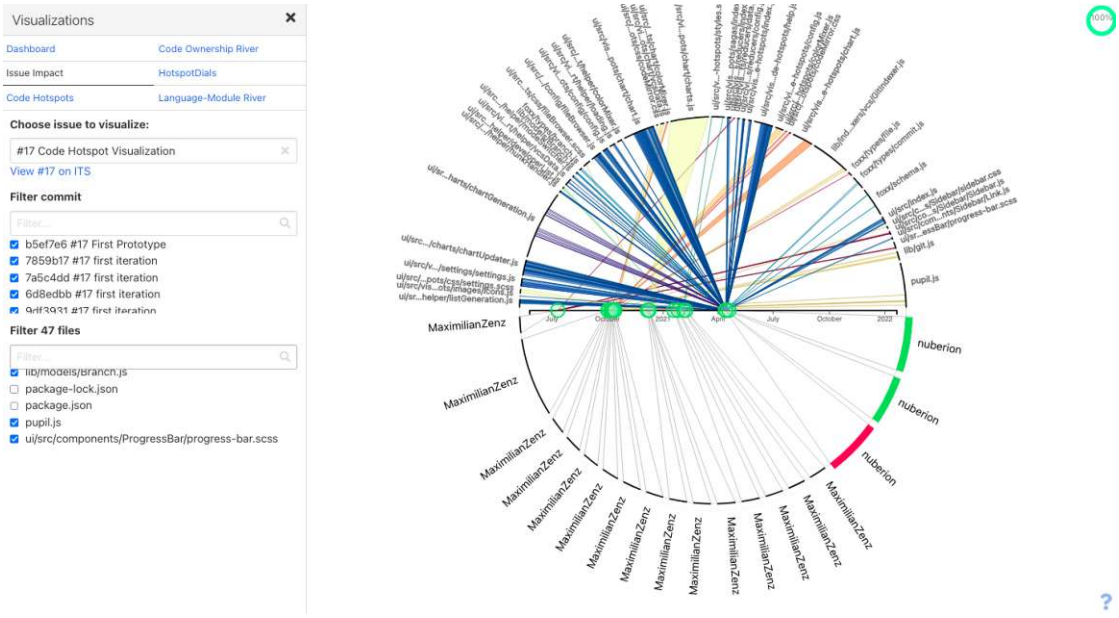


Figure 4.4: Example of the "Issue Impact" visualisation

Similar to the last visualisation, the section about the Issue Impact visualisation does not contain any additional questions. Thus, the section comprises the following questions:

- How purposeful is the Issue Impact visualisation for your role in this course?
- How could it be improved to ease your role?
- How helpful is the Issue Impact visualisation for software engineering education?
- How can this visualisation be utilised in software engineering education?
- What are its shortcomings in software engineering education?

HotspotDials Visualisation

The interviews continue with questions about the "HotspotDials" visualisation. The HotspotDials visualisation shows the total number of commits and the total number of issues. The commits are shown on the outside of the circle, whereas the issues are located

on the inside. Moreover, it offers the option to show the issues by creation or closing data and to split commits into "good" and "bad" ones (a commit is "good" if it has at least one successful CI/CD build). Moreover, it is a cyclic visualisation, which means that it aggregates the data over a given timeframe, for example for a given day of the week. This timeframe can be customised by changing the granularity of the visualisation, for example to the hours of a day. An example of the HotspotDials visualisation using aggregation over the different days of the week is shown in Figure 4.5.

Figure 4.5: Example of the "HotspotDials" visualisation

The section about the HotspotDials visualisation contains two additional questions. Therefore, the questions of this section look as follows:

- How purposeful is the HotspotDials visualisation for your role in this course?
- How could it be improved to ease your role?
- How helpful is the HotspotDials visualisation for software engineering education?
- How can this visualisation be utilised in software engineering education?
- What are its shortcomings in software engineering education?
- How purposeful is the HotspotDials visualisation to gain insights about large spikes in work contributions?
  - How could it be improved to gain the relevant insights?

While the HotspotDials visualisation is a cyclic visualisation, thus spikes will most likely not be visible, the author assumes that it could aid in identifying specific timeframes of increased work contributions, such as certain days of the week or certain hours of the day. Therefore, these questions have also been included.

Code Hotspots Visualisation

Up next are questions regarding the "Code Hotspots" visualisation. The Code Hotspots offers the possibility to see the changes of a given file. More specifically, a heat map overlay representing the history of changes is shown over the source code of the selected file. There is the opportunity to show the changes per version, per developer or per issue. Each of these options shows how frequently and to what extent the given file has been changed. Furthermore, the visualisation offers various filters to customise what information shall be shown. Figure 4.6 features an example of the Code Hotspots visualisation.

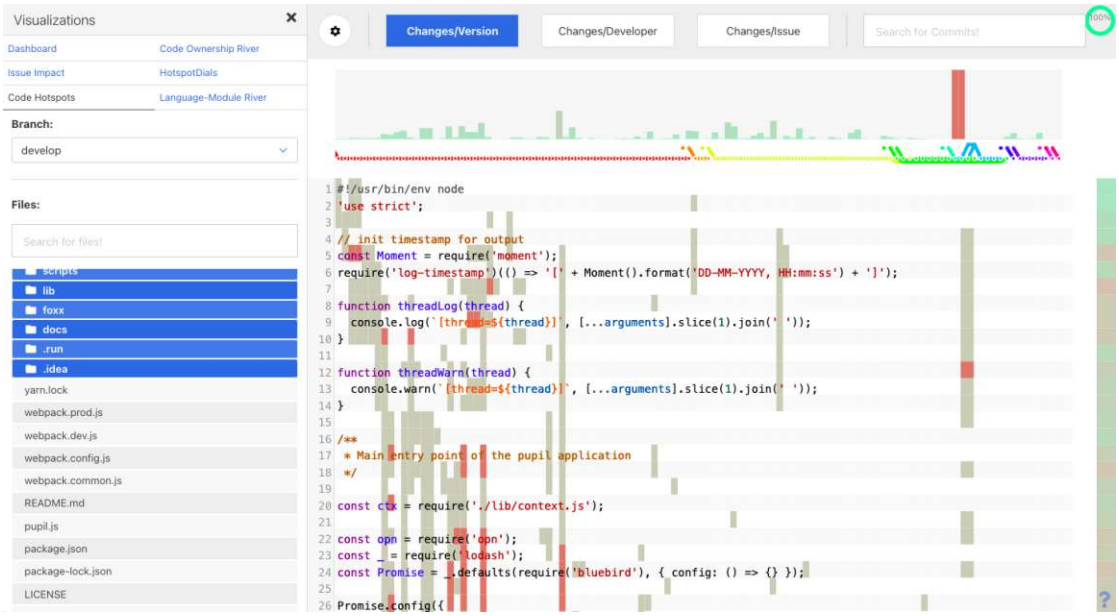


Figure 4.6: Example of the "Code Hotspots" visualisation

Once again, the section about this visualisation only contains the five questions that all sections about the six visualisations have in common, which look as follows:

- How purposeful is the Code Hotspots visualisation for your role in this course?
- How could it be improved to ease your role?
- How helpful is the Code Hotspots visualisation for software engineering education?
- How can this visualisation be utilised in software engineering education?

- What are its shortcomings in software engineering education?

Language-Module River Visualisation

To conclude the sections about the visualisations Binocular comprises, questions about the "Language River" and "Module River" visualisations follow. Both visualisations show the additions and deletions by a given contributor over time. Contrary to the Dashboard visualisation, more specifically its "Changes" chart, these changes are not centred on the x-axis. Instead, a so-called "build trend", which rewards consecutively successful CI/CD builds, is calculated and used to determine the vertical position of the given contribution. Another difference to the Dashboard visualisation is the distinction of changes. While the additions and deletions in the Dashboard visualisation are simply grouped per contributor, the Language-River and Module-River visualisations additionally distinguish changes by their file type or module respectively. This is illustrated through the use of two colours: one for the contributor and another one for the given file type or module. Furthermore, both visualisations add the ability to only show changes for a given file type or module. An example of the Language-River visualisation is shown in Figure 4.7. Due to their similarity, an example for the Module-River visualisation was omitted.

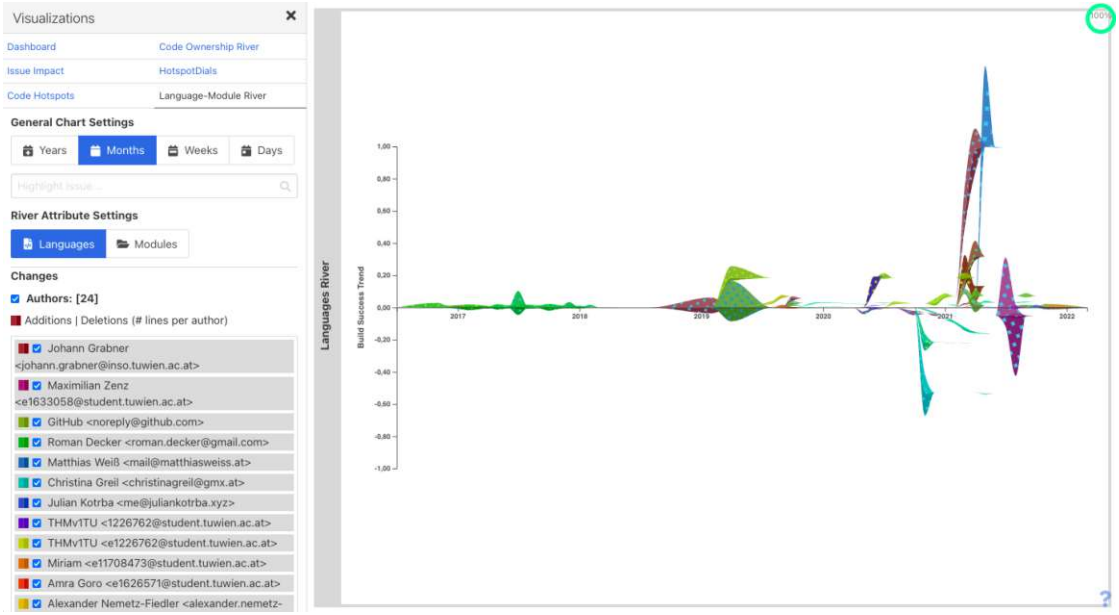


Figure 4.7: Example of the "Language River" visualisation

This section, again, does not contain any additional questions. Therefore the questions look as follows:

- How purposeful is the Language-Module River visualisation for your role in this course?

- How could it be improved to ease your role?
- How helpful is the Language-Module River visualisation for software engineering education?
- How can this visualisation be utilised in software engineering education?
- What are its shortcomings in software engineering education?

Additional Insights

Eventually, an open question about other additional insights that software engineering education could benefit from concludes the questionnaire. The full questionnaire can be found in the Appendix.

4.1.2 Results

Demographics

The introductory section of the interviews contained demographic questions. All four participants marked their age as 25-34 years old. One of the four participants identified as female, the other three as male. Their experience in software engineering varied: one of the interviewees had been working in software engineering education for 1-2 years, one for 5-10 years and the remaining two for 10-15 years. Moreover, the roles of the participants can be clustered into three categories: course administrators, course assistants and tutors. The participants consisted of two course administrators, one course assistant and one tutor. The distribution of roles and their associated experience is illustrated in Figure 4.8 (the age was omitted, since all participants selected the "25-34 years" bucket).

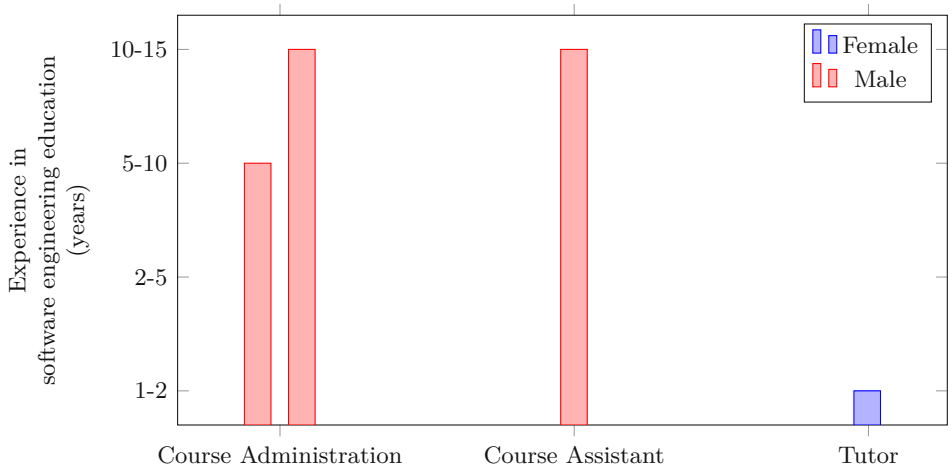


Figure 4.8: Responses to the "Demographics" section, grouped by role performed in the course

Concepts

The following section consisted of five questions regarding the usefulness of insights in software engineering education. These concepts were referenced in later sections as well, where the participants were asked if certain visualisations of Binocular are purposeful to gain these insights. Thus, it was important to find out if these pieces information were useful in the first place. The participants were asked to give ratings one a scale of 1 (not useful) to 5 (very useful). Insights on the work distribution between students had the highest median rating. This is followed by code ownership over time, insights on large spikes in work contribution and the number of active conflicts within the source code of the project (e.g. on different branches). Insights on the history of conflicts within the source code received the lowest median rating. The distributions of the responses are shown in Figure 4.9.

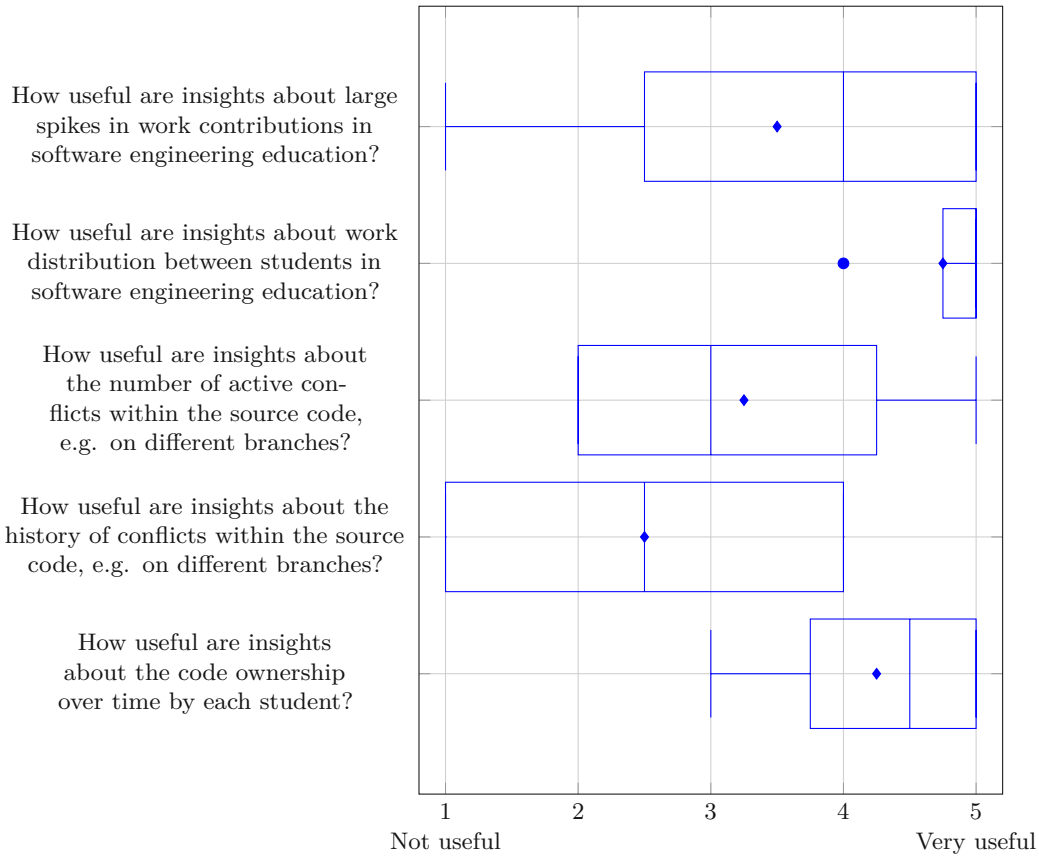


Figure 4.9: Responses to the questions of the "Concepts" section

Dashboard Visualisation

The next section contained questions about the Dashboard visualisation. The participants gave its purposefulness for their role a median rating of 4 out of 5, with two of the

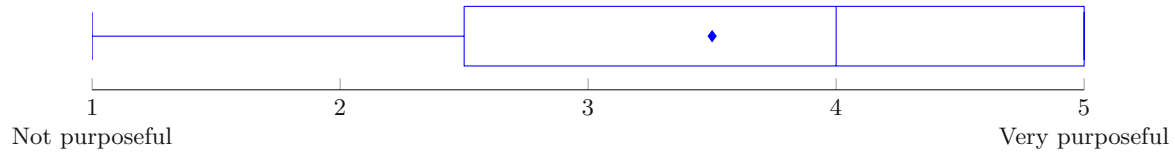
participants rating it "very purposeful". Moreover, the helpfulness for software engineering education even scored a median and mean rating of 4.5 out of 5. The distribution of the answers to these two questions can be seen in Figure 4.10.

The participants were mostly interested in the chart showing the code changes. The interviewees suggested that it would aid in grading, that it could be used as a consistent Dashboard for all stakeholders, and to better explain software metrics to students. The participants did not regard the "Issues" chart as useful as the code changes. This was due to the nature of student projects, where issues are typically created in the beginning of the semester or in periodic planning meetings. Moreover, the participants considered the number of CI/CD builds, especially over time, as not really meaningful, since these builds are not triggered for each commit, but every push to the repository. Thus, there could be multiple failed builds, or even none, solely depending on how frequently students decide to push their contributions to the repository. Additionally, the number of failed CI/CD builds typically is irrelevant, once the build has turned green again. Nevertheless, the interviewees suggested that the issues and CI/CD charts should visualise who created or closed and issue, as well as who triggered a given CI/CD job, as it is done in the "Changes" chart.

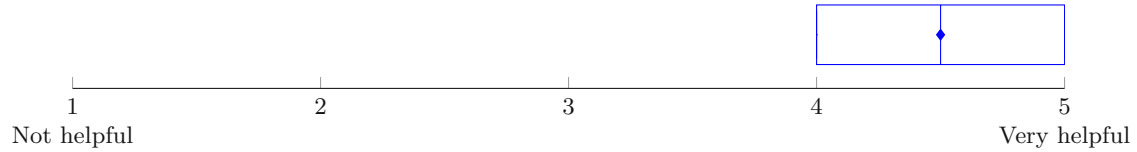
Moreover, a common suggestion amongst the participants was a method to obtain detailed information on changes, since there currently is no way to find out what kind of code students add or delete. An obvious shortcoming would be that, even though one student might have written the most lines of code, most of these contributions could come from auto-generated files, libraries or templates. An obvious solution to this problem would be a filter to show/hide files of a given file format or matching a given file name.

In addition, the participants were asked questions to determine the purposefulness of the Dashboard visualisation to gain three of the insights covered in the "Concepts" section. Three of the four interviewees scored its ability to obtain information on large spikes in work contributions with rating of 5 out of 5, resulting in a median rating of 5 out of 5 ("very purposeful"). It was once again noted by the participants that a possibility to show/hide certain groups of files would be beneficial. The next question determined how purposeful the visualisation is to attain information on the work distribution between students. The participants gave the Dashboard a median rating of 3.5 out of 5. They noted that there is no way to determine if students have made a similar amount of contributions, and that artefacts such as the wiki or documentation are not considered at all. The former issue could be addressed by adding the total number of contributions per student. Furthermore, the lines of code do not necessarily correspond with the amount of work done, since for example setting up a project can be hard to figure out, which may result in code being scrapped and re-written entirely. Moreover, the participants were asked to rate the ability of the visualisation to gain insights on the amount of added/deleted source code by each student. Two of the interviewees gave rating of 4 out of 5, while the other two even rated it 5 out of 5 ("very purposeful"). Once again it was mentioned that a way to show/hide certain groups of files and an absolute number of contributions per student would valuable. To conclude, Figure 4.10 illustrates the

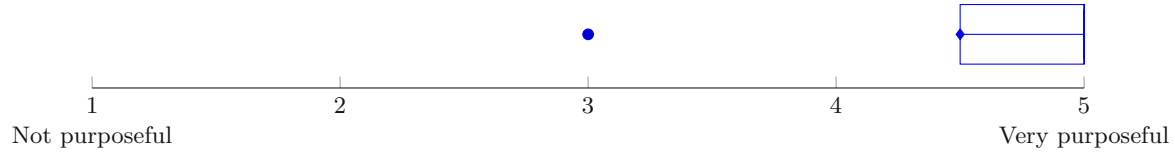
distributions of the ratings given by the participants.



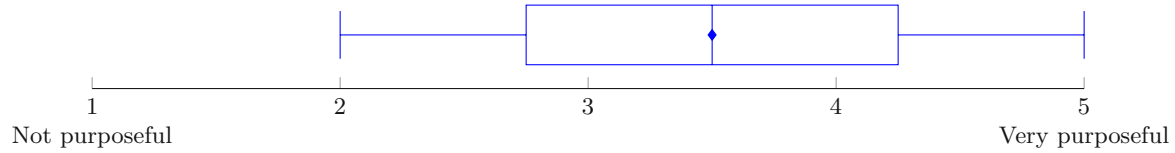
(a) *How purposeful is the Dashboard visualisation for your role in this course?*



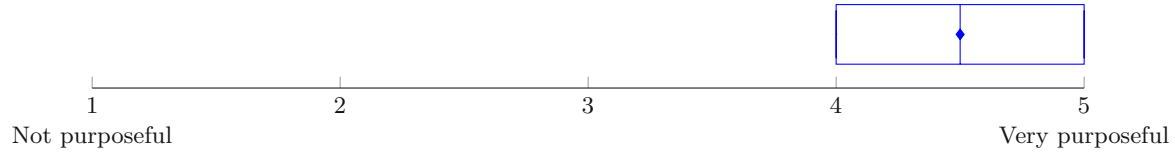
(b) *How helpful is the Dashboard visualisation for software engineering education?*



(c) *How purposeful is the Dashboard visualisation to gain insights about large spikes in work contributions?*



(d) *How purposeful is the Dashboard visualisation to gain insights about work distribution between students?*



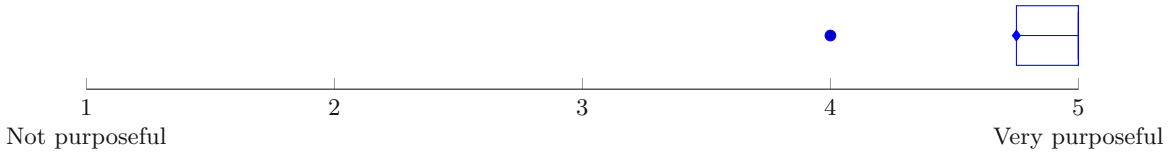
(e) *How purposeful is the Dashboard visualisation to gain insights about the amount of added/deleted source code by each student?*

Figure 4.10: Responses to the questions of the "Dashboard Visualisation" section

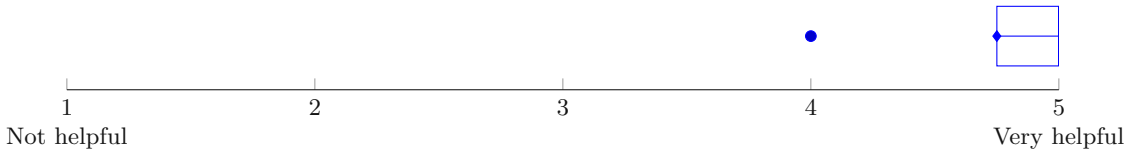
Code Ownership River Visualisation

Subsequently, questions about the Code Ownership River followed. Three of the four participants rated its purposefulness for their role with 5 out of 5 ("very purposeful"), resulting in a median rating of 5 out of 5. The interviewees gave identical ratings when asked about the helpfulness of the visualisation in software engineering education in general. The participants thought that it is very well-suited to determine how much each student has contributed and how the project came to be. Moreover, it was noted that the overlays are confusing and do not add any value. The interviewees also mentioned that zooming should also work for a single axis, during the interviews the x and y axis zoomed in simultaneously. Furthermore, it was stated that the possibility to only show a certain student would be helpful, as well as the ability to show/hide certain groups of files. Lastly, participants noted that the colour selection was irritating. This had to do with the project used to demonstrate Binocular, which was the Binocular project itself. Due to the nature of the project there are dozens of contributors, thus it required a lot of different colours to show all of their contributions in the river. This resulted in similar colours being used for different contributors, which confused some of the interviewees. However, this should not be an issue, since the number of contributors in student projects are limited.

The distributions of the responses are shown in Figure 4.11.



(a) *How purposeful is the Code Ownership River visualisation for your role in this course?*



(b) *How helpful is the Code Ownership River visualisation for software engineering education?*

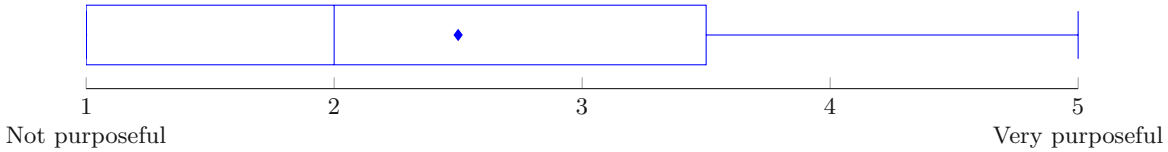
Figure 4.11: Responses to the questions of the "Code Ownership River Visualisation" section

Issue Impact Visualisation

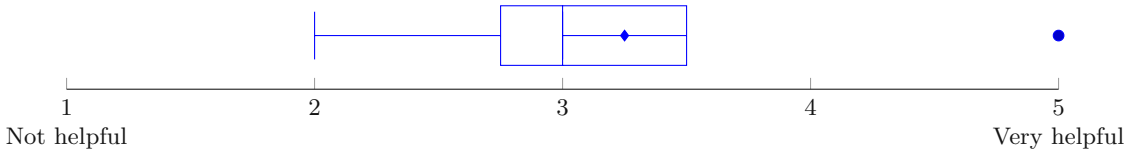
The third section was about the Issue Impact visualisation. The participants gave it a median rating of 2 out of 5 for its purposefulness for their role. However, it is noteworthy that one of the interviewees, more specifically the only tutor amongst the participants, rated it 5 out of 5 ("very purposeful"). The interviewees gave slightly higher ratings

when determining its helpfulness in software engineering education in general, resulting in a median rating of 3 out of 5. Again, the only tutor amongst the participants gave a 5 out of 5 rating ("very useful"). While one of the participants rated both questions identically, the remaining two (a course administrator and a course assistant) thought that the visualisation was more helpful for software engineering in general than it was purposeful for their role. This would suggest that the visualisation is more applicable for tutors, which is supported by the fact that the only tutor amongst the participants rated both questions 5 out of 5. The participants noted that the visualisation is very fine-grained, which is not relevant for course administrators and course assistants, since a broad overview is more helpful for these roles. Nevertheless, it can be well-suited for reviewing a concrete problem, for example if a person has been working on the same issue for a very long time. Apart from its applicability, the participants noted that the labels are not easily distinguishable if a certain issue touches too many files. However, this could also be a sign of an issue that should have been split up into smaller ones.

Figure 4.12 exemplifies the distributions of ratings given by the interviewees.



(a) *How purposeful is the Issue Impact visualisation for your role in this course?*



(b) *How helpful is the Issue Impact visualisation for software engineering education?*

Figure 4.12: Responses to the questions of the "Issue Impact Visualisation" section

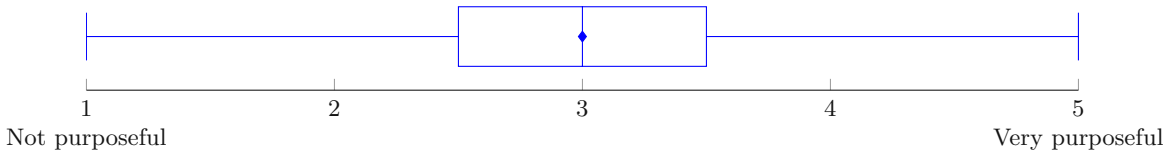
HotspotDials Visualisation

Up next were questions about the HotspotDials visualisation. The interviewees rated the purposefulness of the HotspotDials visualisation for their particular role with a median rating of 3 out of 5. The ratings improve slightly for the helpfulness in software engineering in general, where it received a median rating of 3.5 out of 5. The participants thought it is useful to determine when students are working. This can be helpful in various ways, for example to see if the development process is healthy or to schedule meetings in times when students are not working as actively. However, this information should be handled with a grain of salt, since the data obviously only knows the commit timestamp. Furthermore, the interviewees noted the chosen granularities were not ideal. For example, the hour granularity aggregates over the AMs and PMs (e.g. 1AM and

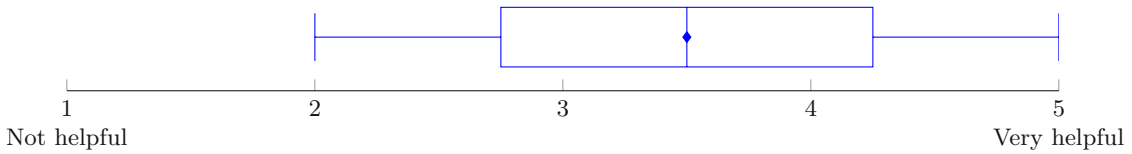
1PM) of each day, a with a dedicated arc for each hour would be better. In addition, a granularity on the basis of a sprint was also considered useful. Lastly, the interviewees suggested that filters for a timeframe or even per developer would be beneficial.

Afterwards, the participants were asked to rate the purposefulness of the HotspotDials visualisation to gain insights about large spikes in work contributions. The interviewees gave a median rating of 3 out of 5. They noted that the correlation of work and commits is not as strong as for example work and lines of code. Moreover, there are typically less contributions in the beginning of a sprint, since developers often have to read up on the subject. Thus, an accumulation of contributions typically occurs towards the end of a sprint.

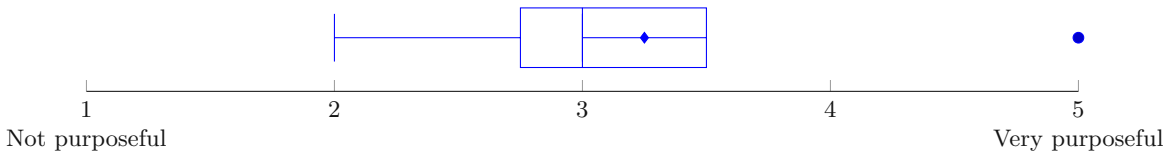
The distributions of responses to these three questions is shown in Figure 4.13.



(a) *How purposeful is the HotspotDials visualisation for your role in this course?*



(b) *How helpful is the HotspotDials visualisation for software engineering education?*



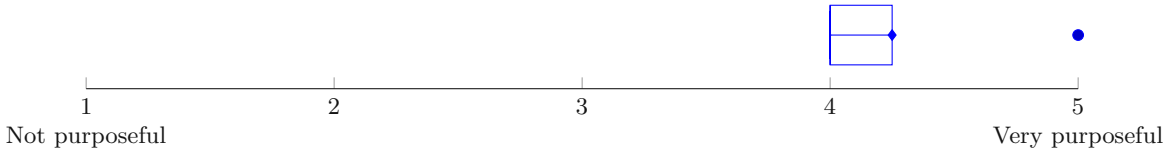
(c) *How purposeful is the HotspotDials visualisation to gain insights about large spikes in work contributions?*

Figure 4.13: Responses to the questions of the "HotspotDials Visualisation" section

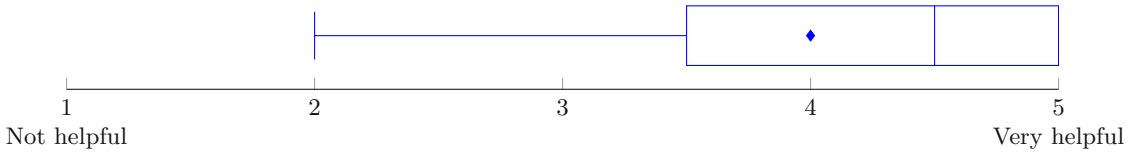
Code Hotspots Visualisation

The second to last section was about the Code Hotspots visualisation. The participants rated gave the Code Hotspots visualisation a median rating of 4 out of 5, when asked about its purposefulness for their role. The median rating for its helpfulness in software engineering education was even higher, at 4.5 out of 5. The participants noted that the visualisation is very complex and, even though there exist tooltips and a help page,

would require an introduction. Moreover, they noted that it could be used to get detailed information on the history of a given file. This could be beneficial when disputes occur, whether that might be due to grading or within a group. Shortcomings mentioned by the participants were the performance and stability, as well as the naive diff algorithm. The rating distributions can be seen in Figure 4.14.



(a) *How purposeful is the Code Hotspots visualisation for your role in this course?*



(b) *How helpful is the Code Hotspots visualisation for software engineering education?*

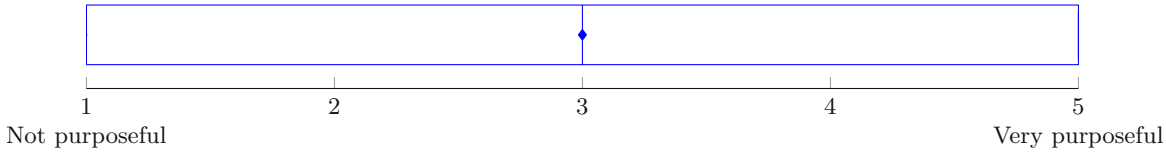
Figure 4.14: Responses to the questions of the "Code Hotspots Visualisation" section

Language-Module River Visualisation

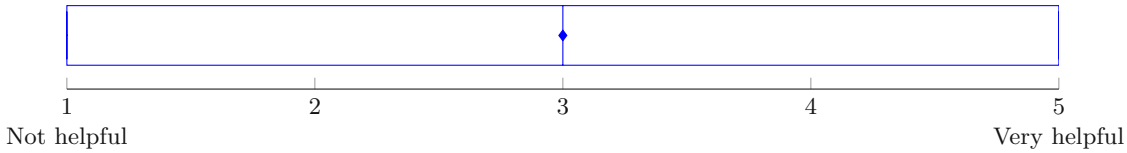
To conclude the sections about the six visualisations of Binocular, questions about the Language- and Module-River visualisations followed. The two aforementioned visualisations produced the most polarising results. Two participants rated its purposefulness for their role with 1 out of 5 ("not purposeful"), while the other two gave the visualisation a 5 out of 5 rating ("very purposeful"). These distributions are illustrated in Figure 4.15. The interviewees gave the same ratings when asked about the helpfulness of the visualisation in software engineering education in general. The two interviewees gave the 1 out of 5 ratings justified their answers by stating that they are rating the visualisation itself, not the filters. Moreover, all of the participants found the "build trend" metric irritating. However, it is noteworthy that all of the participants found the file type and module filters helpful and would think it is beneficial, if they would exist in other visualisations. The interviewees stated that these filters would enable them to identify what type of code the students contribute. This would be helpful in many ways, for example if students are only adding code in templates or configuration files. Additionally, contributions in auto-generated files or libraries could easily be hidden in the visualisation.

Additional Insights

The last section of the interviews contained a single open question. The interviewees were asked for any additional insights that software engineering education could benefit



(a) *How purposeful is the Language-Module River visualisation for your role in this course?*



(b) *How helpful is the Language-Module River visualisation for software engineering education?*

Figure 4.15: Responses to the questions of the "Language-Module River Visualisation" section

from. One participant mentioned that the integration of time tracking data would be beneficial. Moreover, code ownership on a file- or module-level was suggested. This would enable the course staff to see, for example, if the top 10 contributions of a student were in the template files. Lastly, it was mentioned that conversations with students are indispensable, and, that visualisations, like the ones offered by Binocular, are secondary at best when it comes to grading.

4.1.3 Threats to Validity

The first obvious threat to validity is the number of participants that took part in the semi-structured interviews. The small sample size of four interviewees cannot be seen as representative for the entirety of people working in software engineering education.

Furthermore, due to the nature of the thesis, the participants needed to have experience in software engineering education. This narrowed down the number of possible interviewees significantly, basically only leaving the members of the course staff of the undergraduate and graduate software engineering courses at TU Wien as candidates. As a result, all interviewees were from the immediate work environment of the advisor of this thesis, which introduces bias. Some of the participants have even seen a prototype of Binocular before, which additionally increases this bias.

Lastly, the live demo of each visualisation was performed by the author. This introduces additional bias, since the author already knows the strengths and weaknesses of each visualisation of Binocular.

4.2 Information Needs

One main goal of this thesis is the identification of the information needs in software engineering education. A proper understanding of these information needs could also aid in identifying requirements for the proposed architecture for software repository mining and visualisation. The main purpose of the semi-structured expert interviews, which are described in Section 4.1, was to better comprehend these information needs. Moreover, the limitations of Binocular [25] in educational software engineering environments shall be highlighted. The responses provide a good starting point to answer both of these questions.

While the participants were also asked open questions to potentially gather additional information needs, the main focus was to validate the presumed ones. After reviewing the literature, and additional consultation with experts of the field, five different insights, which were assumed to be useful in software engineering education, emerged. The main purpose of the "Concepts" section was to see whether or not these assumptions were correct. Moreover, if a given visualisation was assumed to be purposeful to gain insights on one of these information needs, its corresponding section in the questionnaire contained a question to rate this ability. For example, it was presumed that the "Dashboard" visualisation is well-suited to give insights on large spikes in work contribution, since the number of added and deleted lines of code is shown in the "Changes" chart. Therefore, its section contained a question to determine its purposefulness to satisfy this information need. These questions were specifically asked to find out the limitations of Binocular in educational software engineering environments.

The results, as described in Section 4.1.2, show that each of the mentioned insights was rated at least 4 out of 5 by at least half of the participants. This suggests that the presumed information needs are indeed useful in software engineering education. Furthermore, apart from insights on the history of conflicts within the source code, which was rated 4 out of 5 by two of the interviewees, every insight received at least one 5 out of 5 ("very useful") rating. Ranking the usefulness of the insights according to their median rating, which is shown in Figure 4.9, would result in the following order:

1. Work distribution between students
2. Code ownership over time by each student
3. Large spikes in work contributions
4. Number of active conflicts within the source code
5. History of conflicts within the source code

Even though a lot of information can be derived from just the quantitative questions, the open questions additionally aided in specifying the information needs of the interviewees,

as well as the current shortcomings of Binocular. For example, insights on the work distribution between students were regarded as the most useful by the participants. The "Dashboard" visualisation was assumed to be useful to obtain these insights, since its "Changes" chart shows the amount of added and deleted code per student. However, the interviewees noted that it would be beneficial to see what these contributions are. Moreover, the "Issues" and "CI builds" charts lack user information, and therefore do not aid in determining the work distribution. Artefacts, such as the Wiki or documentation, as well as time tracking data, are also not reflected in any of the visualisations of Binocular. Lastly, it is noteworthy that, while it may be a good indicator, lines of code should not be the exclusive metric to measure productivity. Setting up a project or committing auto-generated files may result in lots of added lines of code, but could take far less time than implementing a complex algorithm or database query that may only end up being a few lines of code. Using this reasoning, one participant even thought that it is not obvious whether or not students have contributed approximately the same.

Insights on the code ownership over time by each student were regarded as the second most useful of the mentioned insights. The "Code Ownership" river illustrates exactly that. The participants gave it the highest ratings with regards to its purposefulness for their role, as well as its usefulness in software engineering education in general. The responses to the questions asking about the purposefulness of a given visualisation for the role of an interviewee can be seen in Figure 4.16.

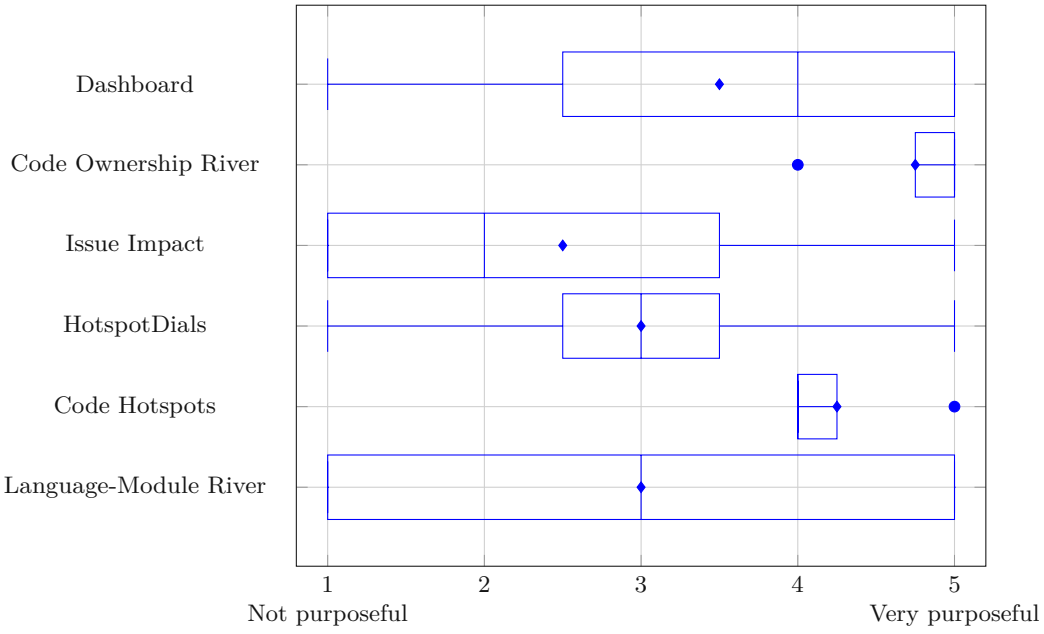


Figure 4.16: Responses to the "How purposeful is the [...] visualisation for your role in this course?" questions

As it turned out, the different roles in software engineering education have different

information needs. A good example for this is are insights on large spikes in work contribution. Both participants who are responsible for the course administration rated it "very useful" (5 out of 5), whereas the tutor gave a 3 out of 5 rating. The only course course assistant amongst the interviewees even gave the insight a "not useful" (1 out of 5) rating. It was assumed that the information needs of the different roles in software engineering education would differ. Therefore, each of the six sections with questions about the visualisations of Binocular contained one question about the purposefulness of a given visualisation for their role, in addition to one tasking about its usefulness in software engineering education in general. The responses to the latter, which are shown in Figure 4.17, turned out to be slightly different. This too suggests that the information needs of the different roles deviate.

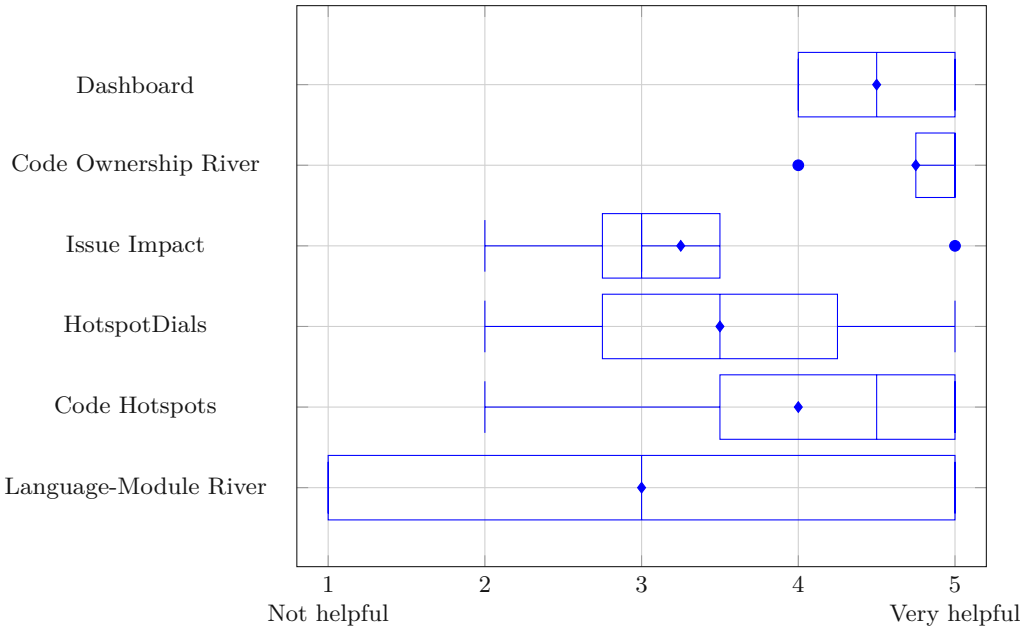


Figure 4.17: Responses to the "How helpful is the [...] visualisation for software engineering education?" questions

The usefulness of insights on the number of active conflicts received the second-worst median rating by the interviewees. The only participant who classified their role as tutor still rated it "very useful" (5 out of 5). Moreover, it is quite obvious that tutors are more concerned with source code than course administrators or course assistants. This additionally indicates that the information needs of the different roles vary.

The participants thought insights on the history of conflicts within the source code are the least useful, according to the median rating. Again, it is noteworthy that the only tutor amongst the interviewees rated it the highest, with a 4 out of 5 rating. This substantiates the argument that tutors are more concerned with the source code than course administrators or course assistants.

CHAPTER 5

Requirement Analysis

The following chapter describes a set of requirements for the proposed in-process software repository mining and visualisation approach. The shortcomings of existing solutions, which were described in Section 3.2.2 and Section 3.2.3, in combination with assumptions by the author, are used to construct hypotheses about a suitable architecture for in-process software repository mining and visualisation. These hypotheses have to be validated later on. Concrete requirements are then derived from these hypotheses. The final architecture should meet these requirements, while simultaneously providing improved satisfaction of the information needs in software engineering education.

The proposed solution should improve upon existing solutions, which have been surveyed in the literature review. Current software repository mining and software visualisation tools almost entirely require external infrastructure or additional integration before they become applicable. In order to solve this issue, the following hypothesis can be constructed:

H1) A software repository mining and visualisation architecture for software engineering education should be integrated in the existing infrastructure (e.g. GitLab CI/CD)

Moreover, the educational context of this thesis has to be kept in mind. Tutors, course assistants and course administrators typically have to supervise multiple, often similar projects, simultaneously. Thus, it is assumed that being able to quickly access, and optionally share, the produced artefacts is vital. This results in the following hypotheses:

H2) Visualisation artefacts should be viewable offline (i.e. without being hosted on a server)

H3) Visualisation artefacts should be portable and cross-platform

Additionally, it should not be necessary for students to add mining and visualisation logic to their projects. This allows the course staff to enable or disable the mining of any student project at any time, while simultaneously not cluttering the student projects. Therefore, the following assumption is made:

H4) A software repository mining and visualisation architecture for software engineering education should require no additional configuration

In order to make the project as transparent as possible, all stakeholders should have access to the visualisation artefacts. Since the architecture is designed and developed in an educational context, the following hypothesis emerges as a result:

H5) Both students and course staff should be able to access the visualisations of their projects

Due to the complex nature of the architecture, various administration tasks will be necessary during setup and maintenance. These may include setting up, updating and deleting projects, or even building its required artefacts. Apart from the initial setup of the project, which is not possible, it should be possible to setup and maintain the entire architecture through the existing infrastructure. It should, for example, not be necessary to have additional scripts to do certain tasks. Therefore, the following assumption is made:

H6) The administration of the architecture should be possible within the existing infrastructure

Since the architecture has at least one external dependency, in particular the GitHub repository[1] of Binocular [25], it should not be necessary to access these dependencies each time a repository is indexed. One the one hand this enables offline operation (after the architecture has successfully been set up), on the other hand performance can be improved significantly. Since the initial setup is required anyhow, the following hypothesis can be derived:

H7) After the initial setup, a software repository mining and visualisation architecture for software engineering education should be operable without external dependencies

It should once again be noted that these hypotheses are based on the findings of the literature review, as well as assumptions by the author. These hypotheses are validated later on. The following requirements are derived from the hypotheses that have been described in this section. Each requirement is associated with a set of hypotheses, which are suffixed in brackets.

[1] https://github.com/INSO-TUWien/Binocular

Use of existing CI/CD infrastructure (H1)

Integrating the proposed architecture in the existing CI/CD infrastructure, in this case GitLab CI/CD, yields many advantages. Primarily no new hardware has to be set up, thus minimising the additional costs. This is an obvious benefit, especially in educational context, where resources are typically more restricted than in the private sector. Furthermore, introducing an additional tool to provide and access the visualisation artefacts seems very cumbersome. To make the solution as universally usable as possible, the architecture shall only use features of the community edition of GitLab CI/CD.

Visualisations as offline artefacts (H2, H3)

The architecture should deliver visualisations, in particular the visualisations produced by Binocular, as offline artefacts. Additionally, the artefacts should be easily shareable and platform independent. Providing tools, such as Binocular, that would otherwise require an external infrastructure, in offline artefacts greatly reduces the entry barrier to their visualisations.

Applicability without additional configuration (H4)

The proposed architecture should be as non-invasive as possible. Student groups should be able to view visualisations of their projects without the need of adding mining or visualisation logic themselves. This not only allows the course staff to enable or disable to mining and visualisation of any project at any time, it also does not clutter the student projects with code unrelated to their specific project. Moreover, the architecture should be designed in a way that the mining and visualisation of projects is performed by default, i.e. student groups do not need to opt in.

Flexible permissions (H5)

Both students and course staff should have access to the visualisation artefacts produced by the architecture. Students could improve their understanding of the software engineering process through the additional insights illustrated in the visualisations. Moreover, the visualisations aid the course staff in guiding, as well as grading the students. Even though both the course staff and students should be able to view the artefacts produced by the proposed architecture, only the course staff should be allowed to enable or disable the mining of a given project. Thus, the architecture requires flexible permissions. GitLab offers predefined permissions and roles[2], which shall be used to address this challenge.

Administration within the existing infrastructure (H6, H7)

The administration of any projects related to the proposed architecture should be possible within the existing infrastructure. Therefore, an administration project should be created, which contains all logic require to set up and maintain the architecture. Once this project has been cloned, the entire administration of the

[2] `https://docs.gitlab.com/ee/user/permissions.html`, last accessed on 02.03.2022

architecture should be possible through this project (or projects created by this administration project). Scripts that are used to set up or maintain projects related to the architecture should be made available as jobs[3] in the CI/CD pipeline of this project. Furthermore, the architecture should be able to build all of its required artefacts, such as custom Docker[4] images.

[3] https://docs.gitlab.com/ee/ci/jobs/, last accessed on 02.03.2022
[4] https://www.docker.com/, last accessed on 03.02.2022

CHAPTER 6

Implementation

The following chapter describes the implementation of the proof of concept. Initially, Section 6.1 describes the technologies that were used. Next up, Section 6.2 takes a look at the final architecture, as well as some other options that were considered in the process. Lastly, Section 6.3 covers the repositories dedicated to the administrative purposes.

6.1 Adaptation of Binocular

Binocular utilises a client-server-based architecture. More specifically, the back-end is a Node.js[1] application responsible for gathering and persisting the data from the VCS, ITS and CI/CD into ArangoDB[2]. Furthermore, Express[3] is used to host the precompiled front-end over HTTP and installs the GraphQL[4] service, which the front-end uses to query the data stored in ArangoDB. The front-end is a single-page web application using React[5], D3.js[6] and Redux[7]. How these technologies work together is shown in Figure 6.1.

As mentioned in Section 3.2.2, it is still under active development and currently offers six different visualisations: the "Dashboard", "Code Ownership River", "Issue Impact" (originally called "Change Impact Wheel"), "HotspotDials" (originally called "Activity Peak Dial"), "Code Hotspots" and the "Language-Module River" visualisation. Porting all of these visualisations would be out of the scope of this thesis. To proof the feasibility of the concept the Dashboard visualisation has been adapted to work in the offline artefacts produced by the designed architecture.

[1] https://nodejs.org/, last accessed on 15.09.2021
[2] https://www.arangodb.com/, last accessed on 15.09.2021
[3] http://expressjs.com/, last accessed on 09.03.2022
[4] https://graphql.org/, last accessed on 15.09.2021
[5] https://reactjs.org/, last accessed on 15.09.2021
[6] https://d3js.org/, last accessed on 15.09.2021
[7] https://redux.js.org/, last accessed on 15.09.2021

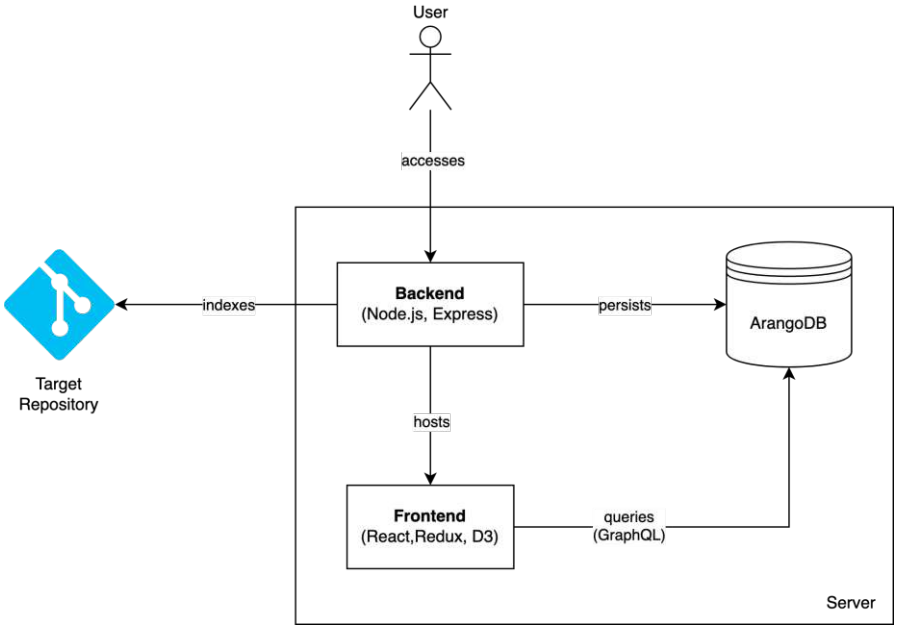


Figure 6.1: Client-server architecture used by Binocular

Before creating the mining pipeline it was necessary to adjust the behaviour of Binocular. The first step was to stop the services started by the back-end after the indexing and data export has been finished. Moreover, it was quite obvious that the data stored in ArangoDB had to be exported and bundled with the front-end in one way or another. While there exist tools to mock GraphQL servers (e.g. `json-graphql-server`[8]), as well as tools to fake REST servers (e.g. `FakeRest`[9]), they are not sufficient on their own. The offline artefacts would require to run in-browser GraphQL queries against a fake REST server, thus a combination of both would be necessary. Exporting the data to JSON files using `arangoexport`[10] and bundling it with the client code seemed more straightforward. Lastly, the GraphQL queries that are used to fetch the data in the front-end had to be replaced.

To stop the services started by the back-end, in particular the web server hosting the front-end and the GraphQL service, a `--no-server` command line option was introduced. Whenever it is present all services are stopped immediately after the data has been indexed and stored into ArangoDB.

Before the data can be bundled with the front-end, it needs to be exported using `arangoexport`. One command suffices to export all collections to corresponding JSON

[8] `https://github.com/marmelab/json-graphql-server`, last accessed on 16.09.2021
[9] `https://github.com/marmelab/FakeRest`, last accessed on 16.09.2021
[10] `https://www.arangodb.com/docs/stable/programs-arangoexport.html`, last accessed on 16.09.2021

files. PouchDB[11] is then used in the front-end to replicate the graph data model from ArangoDB. The adapted version of Binocular assumes that the ArangoDB collections have already been exported to the directory of the front-end. These JSON files are then used to create two document stores in PouchDB. The first document store stores all documents that are located in non-relational collections, e.g. `commits` or `issues`. The second one acts as a triplestore to store all relations between these documents. It stores the entries of all documents that store relations. For example, the documents in the ArangoDB relation `commits-files`, which stores the relationships between commits and files, contains (amongst others) two attributes: `_from` (holding the ID of the file) and `_to` (holding the ID of the associated commit). This entry would be stored in the triplestore by adding a `name` attribute with the value of its relation name, in this case `commits-files`. This way all collections and relations can be reproduced using just two documents, which are created during the initialisation of the Dashboard visualisation component.

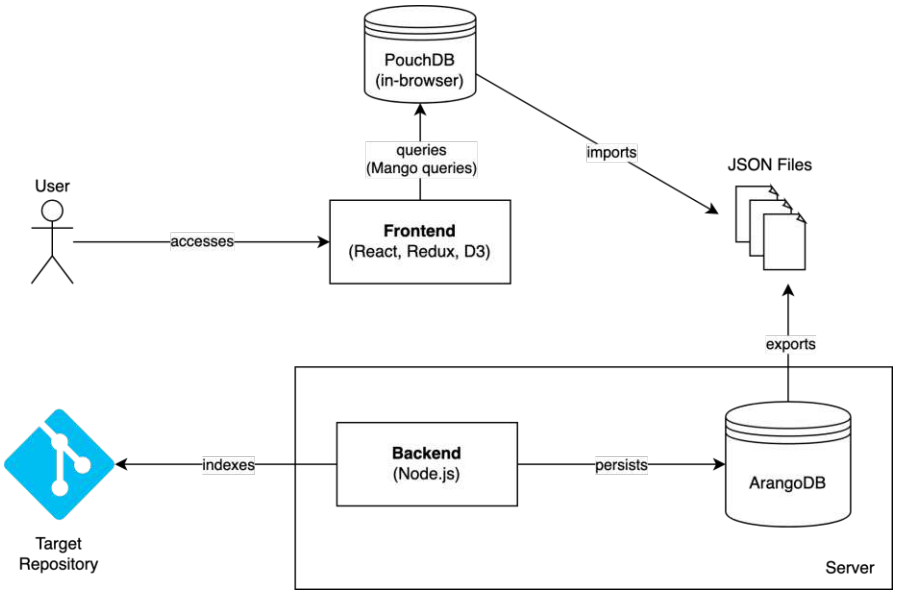


Figure 6.2: Architecture of the adapted version of Binocular

Mango queries[12] were used to replace the GraphQL queries to fetch the relevant data. The Dashboard visualisation uses four different queries to fetch all of the necessary data. The query `getBounds` retrieves the Git signatures of all committers, the first and last issue as well as the first and last commit to the repository. The other three queries, `getBuildData`, `getCommitData` and `getIssueData` return all builds, commits and issues respectively. While the `getCommitData` and `getIssueData` simply return all records of the given document, the records returned by `getBuildData` also include a

[11] `https://pouchdb.com/`, last accessed on 15.09.2021
[12] `https://pouchdb.com/guides/mango-queries.html`, last accessed on 15.09.2021

`stats` object, which holds the number of succeeded, failed, pending and cancelled builds up to that point.

Eventually, some minor changes, such as hiding the progress bar of the data import or disabling the web socket connection that checks the current status of the import, were done to tidy up the offline artefacts. Figure 6.2 illustrates how the individual components of the adapted version work together.

6.2 Architecture

An integral part of the implementation was coming up with a suitable architecture, based on the requirements defined in Section 5. The goal was to integrate Binocular into GitLab CI/CD by making its visualisations available as offline artefacts in the job artefacts[13] of each CI/CD build.

The initial idea was to write a custom GitLab Runner[14] that automatically executes the mining logic and outputs the offline artefacts as job artefacts in GitLab CI/CD. This could be possible by creating one or more runners with a given tag, as well as adding dedicated a mining job to the `.gitlab-ci.yml` file of each project. Furthermore, it would also be necessary to add the tag of the custom runners and the directory of the job artefacts to the configuration. This is not desirable for many reasons, one of the most significant ones being that students could simply decide not to include the given configuration, thus not utilising the architecture at all. This directly conflicts with the requirements that were identified in Section 5.

The second approach that was considered is an architecture comprising not only the student projects, but also projects solely responsible for mining. Each student project has an associated *shadow project*, as these projects are called, with the single purpose of performing the mining. The shadow projects are created using a template, which too is hosted on the same GitLab instance. This project, which is called *Minocular*, contains the entire mining logic. In addition to the CI/CD pipeline, which is defined in the `.gitlab-ci.yml` file, the project also contains the script being executed inside the pipeline. Since the mining logic is located in the shadow projects, the offline visualisations produced by the pipeline are also provided there. Therefore, students need read access to their corresponding shadow project. It is noteworthy that, due to the mining pipeline being relatively resource-intensive, it definitely should not be triggered every time students push changes to their student project. While the pipelines in the proof of concept, due to simplicity of the implementation, have to be triggered manually, mining each student project once per day seems reasonable.

The mining pipeline itself is straightforward. It is based on a custom Docker image and consists of a single job. It adds the `arangodb:3.7` image as a service, since the main

[13] `https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html`, last accessed on 15.09.2021

[14] `https://docs.gitlab.com/runner/`, last accessed on 15.09.2021

container needs access to an instance of ArangoDB. A custom image is used to eliminate the need of re-building the adapted version of Binocular each time a mining pipeline is triggered within a shadow project. The job executes a single script that contains the entire mining logic, which looks as follows: after a couple of variable declarations the target repository is cloned into the main container. This is followed by the creation of the main configuration file, which is done using the existing environment variables. Afterwards, the target repository is indexed using the adapted version of Binocular, which was described in Section 6.1. Subsequently, the data stored in the ArangoDB container is exported to JSON files using the `arangoexport` tool, which is provided by ArangoDB. Once the JSON files are ready, the front-end is build using `npm run build`, which uses `webpack` to create the offline visualisations. Eventually, the artefacts are moved to the job artefacts folder, which is defined in the `.gitlab-ci.yml` file. By doing so an archive of the offline visualisations can be downloaded directly for each build.

To simplify the administration of the student projects (and their corresponding shadow projects) there exist two additional projects for administrative purposes. The first one, called *Labocular*, is used set up to manage the entire architecture. It hosts a CI/CD pipeline that can be used to clone the necessary projects, update them and to create shadow projects. The other project, called *Visocular*, contains a CI/CD pipeline that can be used to build the Docker image used by the shadow projects. These two projects are described in detail in Section 6.3. The final architecture, and how students the course staff interact with it, is shown in Figure 6.3.

6.3 Administration

Two additional projects were created to make the accompanying processes of obtaining the offline visualisations, in particular the creation and maintenance of the shadow projects, easier. *Labocular* offers functionality to set up and maintain the entire architecture. Furthermore, it aids in the creation of the shadow projects. *Visocular* contains logic to build the custom Docker image used by the shadow projects.

6.3.1 Labocular

Once the Labocular project has been cloned into a GitLab group, it can be used to set up and maintain the proposed architecture. It offers six different, jobs, all of which can only be manually triggered. Figure 6.4 displays these six jobs and their corresponding stages. These stages, i.e. "build", "setup" and "maintenance", are not representing stages in the traditional sense of CI/CD, where jobs need to run subsequently for each pipeline, but rather an indication in which order these jobs should be triggered when the infrastructure is first set up. Once the setup is finished, these jobs may be triggered independently from each other.

The first of the six jobs, namely `build_labocular_image`, is very straightforward. It is responsible for building the Docker image used by all other jobs in its CI/CD

pipeline. Therefore, it needs to be run at least once before all other jobs. The script logs authenticates with the local container registry[15] using predefined CI/CD variables[16]. Then the image is built using the `Dockerfile`[17] located in the Labocular project. Subsequently, the image is pushed it to the local container registry, so it can be used by the other jobs. The image is based on the `python:alpine3.13` image, since some of the scripts are written in Python. Afterwards, `git` is installed via `apk`[18]. Lastly, the Python dependencies listed in `requirements.txt` are installed via `pip`[19].

At this point the Docker image is present in the local container registry. Before shadow projects can be created, the `setup_required_projects` job has to be executed. It is responsible for setting up the Minocular and Visocular for the current group. If one of these projects already exists for a given group, this setup is skipped. Otherwise, the given project is cloned from the `INSO-TUWIEN` group[20], where these repositories are maintained.

Since manually creating shadow projects for all student projects would be very cumbersome (there typically are dozens of projects per semester), Labocular offers logic to simplify this task. This logic can be used once the Docker image has been built and the required projects have been set up using the aforementioned jobs. The `create_shadow_projects` job can automatically create shadow projects for selected repositories or for all repositories of a group that match a given prefix (e.g. `21ws-xyz` for a given semester and course name). It creates these repositories by creating a new GitLab project through the GitLab REST API[21], importing the Minocular template, adding all members of the student project as guests (i.e. they have read access to the artefacts) and setting various environment GitLab CI/CD variables, such as the GitLab access token and the name of the student project.

The behaviour of the script depends on the set environment variables. Authentication happens through personal access tokens[22]. Therefore, a variable called `GITLAB_TOKEN` with a valid access token has to be set. If `SELECTED_REPOSITORIES`, which should contain a comma separated list of repository names, is set, shadow projects will only be created for repositories of the current group matching these names. If `PROJECT_PREFIX` is set to a prefix, shadow projects are only created for repositories of the current group matching the given prefix. By default, shadow projects add a `-shadow` suffix to the name of the student projects, e.g. a student project `21ws-xyz` would get a corresponding

[15] `https://docs.gitlab.com/ee/user/packages/container_registry/`, last accessed on 09.03.2022

[16] `https://docs.gitlab.com/ee/ci/variables/predefined_variables.html`, last accessed on 09.03.2022

[17] `https://docs.docker.com/engine/reference/builder/`, last accessed on 09.03.2022

[18] `https://wiki.alpinelinux.org/wiki/Package_management`, last accessed on 09.03.2022

[19] `https://pip.pypa.io/en/stable/`, last accessed on 09.03.2022

[20] `https://gitlab.com/INSO-TUWien`, last accessed on 09.03.2022

[21] `https://docs.gitlab.com/ee/api/api_resources.html`, last accessed on 29.09.2021

[22] `https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html`, last accessed on 29.09.2021

shadow project called `21ws-xyz-shadow`. This suffix can be overwritten by setting the `SHADOW_PROJECT_SUFFIX` variable. To aid in executing this job with the correct variables, all of the relevant variables are shown when the pipeline is triggered manually, as shown in Figure 6.5.

The three jobs in the "maintenance" stage are very similar. Each of the three relevant projects, i.e. Labocular, Minocular and Visocular, can be updated to their latest version, or any specific commit. By default the projects located in the `INSO-TUWIEN` group[23], are used as the source. However, the URL to clone from can be overwritten if necessary.

6.3.2 Visocular

There are multiple reasons why a custom Docker image is used in the CI/CD pipelines of the shadow projects. The most obvious one is that the adapted version of Binocular does not have to be built every time a mining pipeline is triggered within a shadow project. By putting the finished build into an image it suffices to build Binocular once every time the image is re-built. This not only uses far less resources, but also improves performance significantly. Moreover, it enables the pipeline to run without additional dependencies once the images have been built. How the actual images are built is described at the end of this section.

The Visocular project, which is responsible for building the custom Docker image, is very straightforward. Apart from the `README.md` file, it only contains the CI/CD configuration within `.gitlab-ci.yml`. The CI/CD pipeline contains a single job. The job logs in to the local container registry, clones the Minocular project, builds an image using its Dockerfile and eventually pushes the built image to the local container registry. The proof of concept publishes the finished image under the `latest` tag. Even though this may be convenient for development, it should definitely be be adapted to work with different tags before the architecture is deployed in a real-world setting.

While it may seem excessive to have an entire project dedicated to building a single image, this decision was made with expandability in mind. Different software repository mining and visualisation tools may require different images (or different artefacts altogether). Having a dedicated project for managing these artefacts has multiple benefits. The CI/CD pipeline of the Labocular project does not get bigger, every time the architecture is extended with a new tool. Moreover, there exists a clearer separation of concerns. The Labocular project is responsible for setting up and maintaining the projects of the infrastructure, while Visocular is solely responsible for the required artefacts.

The custom image for the mining pipeline is based on the `node:14` Docker image. Since the mining script needs access to the `arangoexport` executable, an ArangoDB installation is downloaded and extracted manually, thus making it available to use. This at first requires `wget`[24] to be installed. Once both of these tasks are finished, the adapted

[23] `https://gitlab.com/INSO-TUWien`, last accessed on 09.03.2022
[24] `https://www.gnu.org/software/wget/`, last accessed on 16.09.2021

version of Binocular is cloned (the adapted version is located on branch `feature/30` of the GitHub repository[25]). Subsequently, the dependencies of the adapted version of Binocular are installed.

[25] `https://github.com/INSO-TUWien/Binocular`, last accessed on 16.09.2021

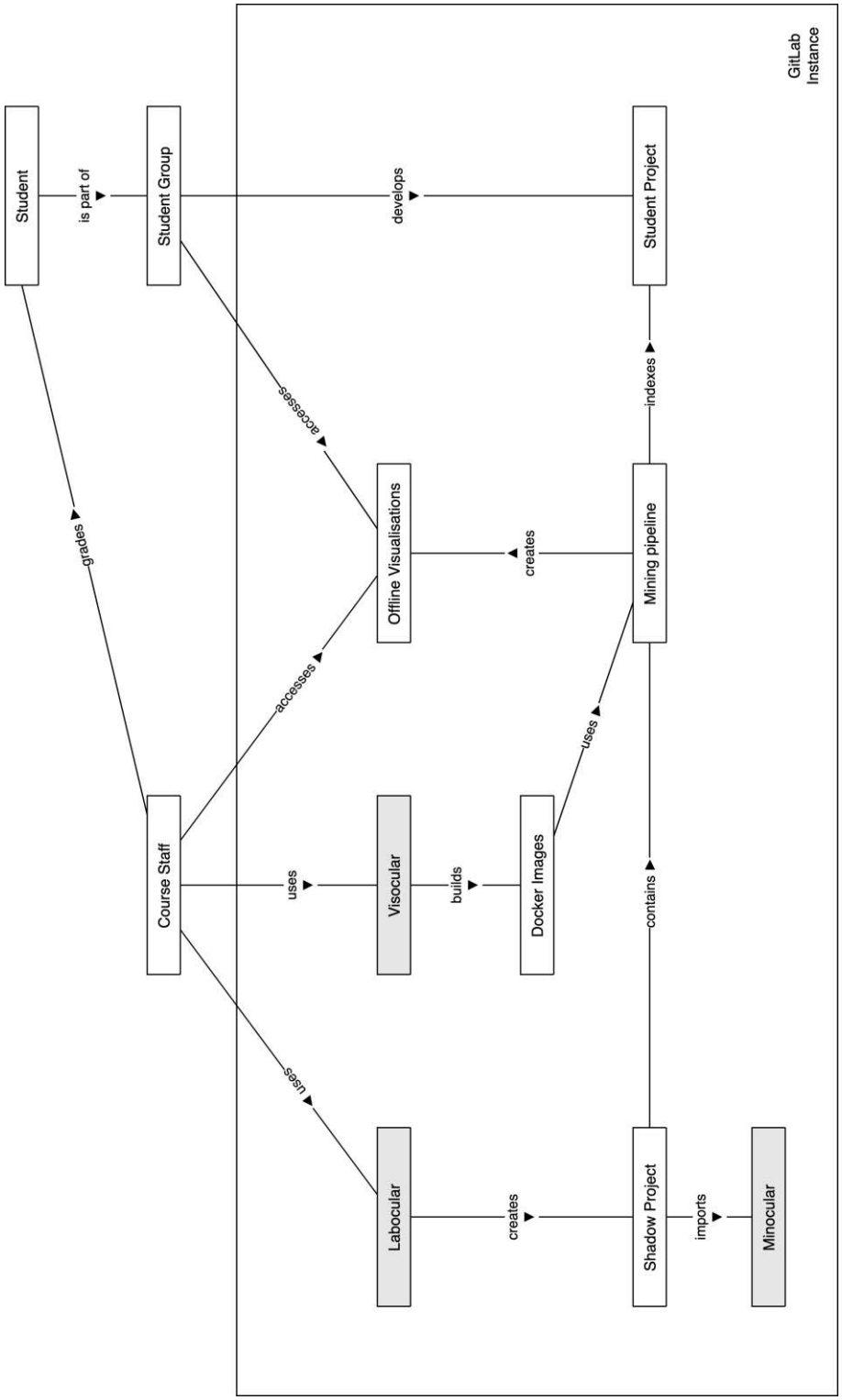


Figure 6.3: The final architecture

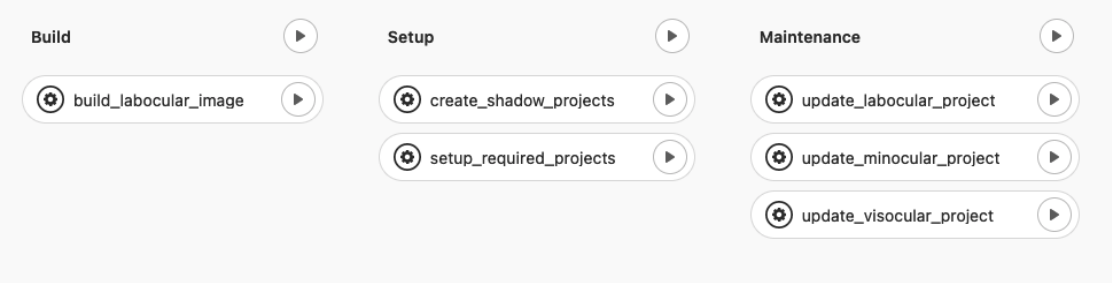


Figure 6.4: Available CI/CD jobs in the Labocular project

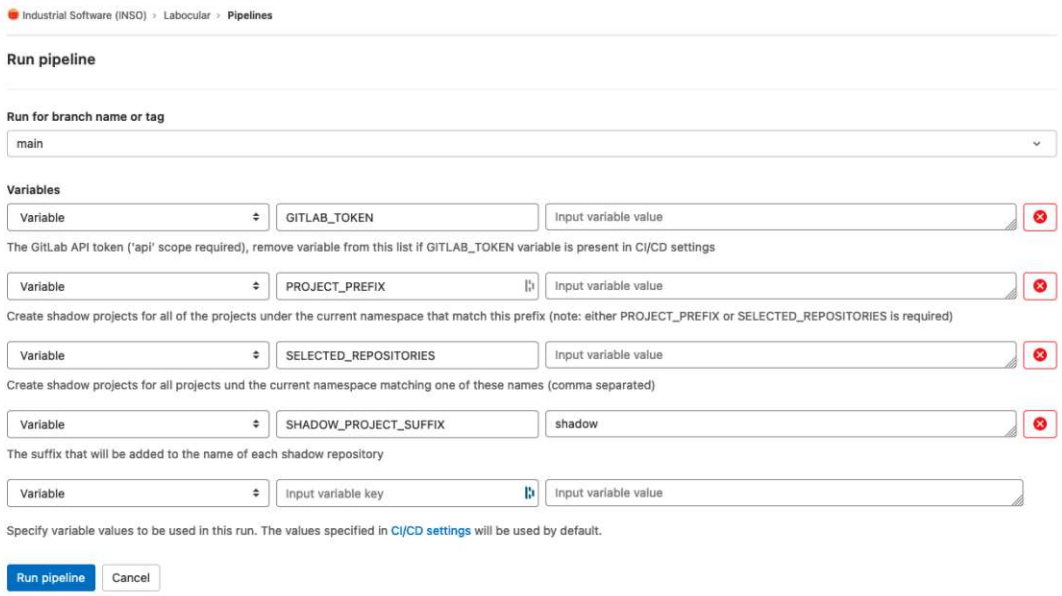


Figure 6.5: "Run pipeline" screen in the Labocular project

CHAPTER 7

Evaluation

This chapter describes the evaluation of the proposed architecture. Initially the goals of the evaluation round are described. Afterwards, the method used to evaluate the approach, the expert interviews, is illustrated. Subsequently, the results of these interviews are presented, summarised and discussed. Lastly, potential threats to validity are mentioned.

7.1 Goals

Section 3.2.2 and Section 3.2.3 highlighted the current limitations of software repository mining and visualisation tools. The information needs in software engineering education, as well as the current limitations of Binocular [25] were derived from the semi-structured interviews mentioned in Section 4.1. As a consequence RQ3, RQ4.1 and RQ4.2 remain open. The goal of the evaluation is to get answers to these research questions.

7.2 Method

To gain the aforementioned insights, six semi-structured interviews with experts of the field in software engineering education were conducted. More specifically, all interviewees were part of the undergraduate software engineering course at TU Wien at the time of writing this thesis. While there is some overlap with the interviewees of the first interview round, not all four were available for the second round. The interviews started with an introduction to the topic and the proposed architecture. Afterwards, a live demo of the architecture was performed to provide a better understanding. An in-depth look at the three projects, in particular Labocular, Minocular and Visocular, was given. Moreover, the process of creating shadow projects and obtaining the offline visualisation artefacts produced by the architecture was showcased. The interviews concluded with a questionnaire consisting of four sections. The first of the four sections contained demographic questions, which were identical to the introductory section of the

questionnaire described in Section 4.1. Subsequently, a section about hypotheses of a suitable architecture for software repository mining and visualisation, independently of the proposed solution, followed. An example would be "a software repository mining and visualisation architecture for software engineering education should be integrated in the existing infrastructure". The participants were asked to give ratings from 1 ("strongly disagree") to 5 ("strongly agree"). These hypotheses were constructed based on the current limitations of MSR tools and software visualisations. Rating the purposefulness of the designed architecture and process was the aim of the third section. It contained statements about the architecture fulfilling the aforementioned hypotheses, e.g. "the proposed architecture is integrated in the existing infrastructure". Once again, the interviewees were asked to rate their acceptance of these statements from 1 ("strongly disagree") to 5 ("strongly agree"). Lastly, a section with a single open question for any further improvements or suggestions concluded the interviews. The full questionnaire can be seen in the Appendix of this thesis.

7.3 Results

The following section describes the results of the expert evaluations. Its structure is identical to the individual sections of the evaluation questionnaire. The initial subsection highlights the demographic information of the participants. Subsequently, the acceptance of the various hypotheses is illustrated. The results of the evaluation of the proposed architecture is described next. Lastly, additional inputs provided by the interviewees are mentioned.

7.3.1 Demographics

The first section of the evaluation interviews contained demographic questions. Three of the six interviewees picked the 25-34 years bucket, while the other three chose 35-44 years. One of the participants identified as female, the remaining five as male. The experience in software engineering education varied all the way from 2-5 years up to 15-20 years. Four of the six participants described their role to be course assistant in the undergraduate software engineering course at TU Wien, with the other two interviewees being course administrator. The demographic information of the interviewees is illustrated in Figure 7.1. It should be noted that, even though it was omitted from the plot, the interviewee who identified as female is the course assistant aged 25-34 with 10-15 years of experience in software engineering education.

7.3.2 Hypotheses

One main goal of the evaluation was to determine a suitable architecture for software repository mining and visualisation. Various shortcomings of current solutions were identified after reviewing the literature. As a consequence, multiple hypotheses of a suitable architecture were constructed. The acceptance of these hypotheses is shown in Figure 7.2.

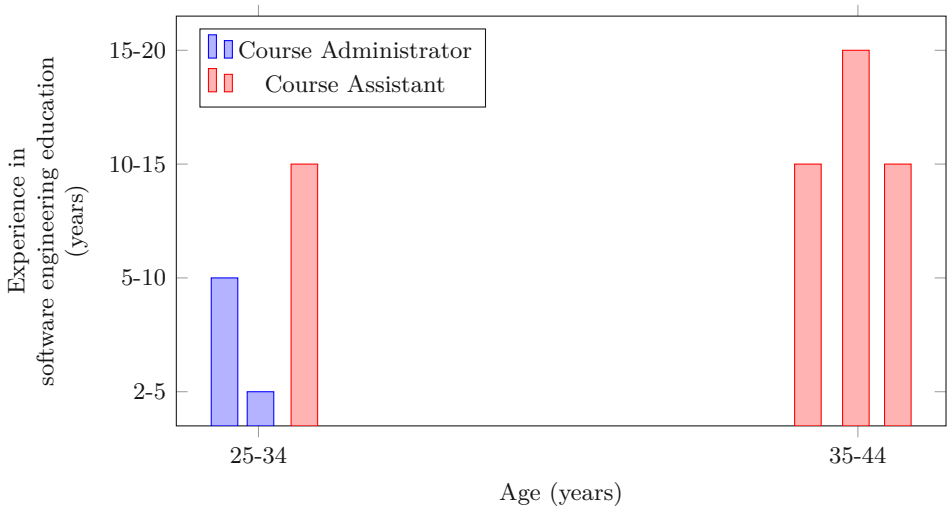


Figure 7.1: Responses to the "Demographics" section of the evaluation questionnaire

The first hypotheses was that a software repository mining and visualisation architecture for software engineering education should be integrated in the existing infrastructure. Four of the six interviewees gave a 5 out of 5 rating ("strongly agree"). The other two participants rated it 4 out of 5. Moreover, they noted that, while an integration into existing tools is favourable, setting up an external infrastructure would not be a major downside to them.

The next hypothesis was that visualisation artefacts should be viewable offline. The participants gave a median rating of 3.5 out of 5, which was the lowest among all hypotheses. One interviewee, in particular one of the course administrators, even rated it 5 out of 5 ("strongly agree"). The phrasing of this hypothesis may have not been ideal, since multiple interviewees stated that they do not care whether or not it an artefact is located online. While this information is useful, the question, in conjunction with the next one, actually aimed at finding out whether or not offline artefacts, i.e. artefacts that do not require an infrastructure to be viewable, are favourable over solutions that require additional setup.

Subsequently, it was asked whether or not visualisation artefacts should be portable and cross-platform. Five of the six interviewees gave a 5 out of 5 rating ("strongly agree"). The remaining participant gave a 4 out of 5 rating. As stated before, the former and this question a have not been formulated ideally gain the desired insights. The process of drafting the questions was done across multiple iterations, both questions were rephrased multiple times. Therefore, the phrasing may have not been ideal.

The next hypothesis was that a software repository mining and visualisation architecture for software engineering education should require no additional configuration. Four of the six interviewees strongly agreed with this hypothesis, i.e. gave a 5 out of 5 rating. All six participants noted that the wording "no additional configuration" was mediocre and that

something along the lines of "minimal additional configuration" would have been better. The ratings were given under the assumption that this was the actual phrasing of the question.

The fifth of the seven hypotheses was that both students and course staff should be able to access the visualisations of their projects. All six interviewees gave a 5 out of 5 rating ("strongly agree"). It was noted that, in addition to the course staff, the students would also benefit from these visualisations. Moreover, several participants emphasised the importance of transparency in software engineering education, both from the view of a student and the course team.

The second to last hypothesis aimed at finding out whether or not the administration of the architecture should be possible within the existing infrastructure. Three of the six participants strongly agreed with this hypothesis, i.e. gave a 5 out of 5 rating. The remaining three interviewees all gave a 4 out of 5 rating, resulting in a median acceptance of 4.5 out of 5 for this hypothesis.

The last of the seven hypotheses was that a software repository mining and visualisation architecture for software engineering education should be operable without external dependencies, once it has been set up. The median rating of this hypothesis was 4 out of 5. The only participant who strongly agreed with this hypothesis was one of the two course administrators.

7.3.3 Evaluation

After rating the constructed hypotheses, the interviewees were asked to evaluate the proposed architecture. This section of the questionnaire contained statements about the final solution and its core concepts and processes. The responses to the statements of this section are illustrated in Figure 7.3. It is noteworthy that, apart from one, all propositions received a median rating of 5 out of 5 ("strongly agree").

The first statement was that the concept of shadow projects, which is an essential part of the architecture, is easy to grasp. Five of the six interviewees strongly agreed with this proposition, i.e. gave a 5 out of 5 rating. It should be noted that the concept was explained in detail in the introduction of the interview. However, every single participant found the name "shadow project" confusing. Suggestions for alternative namings are discussed in Section 7.3.4.

Then the participants were asked whether or not the proposed architecture is integrated in the existing infrastructure. Again, fix of the six interviewees rated this statement with 5 out of 5 ("strongly agree"). The remaining participant, who gave a 4 out of 5 rating, noted that automatic updates and the renovates[1] would improve the level of integration even further. Moreover, there could be an option for automatic exploration of projects to create shadow projects for.

[1] `https://docs.renovatebot.com/`, last accessed on 13.03.2022

The third thesis was that visualisation artefacts produced by the infrastructure are easily viewable offline, i.e. without being hosted on a server. Four of the six interviewees strongly agreed with this statement, i.e. gave a rating of 5 out of 5. The remaining two participants gave a 4 out of 5 rating.

Afterwards, the participants were asked to rate the statement "the visualisation artefacts are portable and cross-platform". Five of the six interviewees gave a 5 out of 5 rating ("strongly agree"), with an additional 4 out of 5 rating from the sixth participant.

The only statement with a median rating less than 5 out of 5 followed next. The participants rated the proposition that the proposed architecture requires no additional configuration with a median rating of 4.5 out of 5. Again, it was noted that the phrasing was not ideal.

The next statement was that both students and course staff can access the visualisations of their projects. All six participants strongly agreed with this statement, i.e. gave a 5 out of 5 rating.

The second to last proposition was that the process of creating shadow projects for existing projects is straightforward. Four of the six participants gave a 5 out of 5 rating ("strongly agree"). The remaining two interviewees rated it 4 out of 5.

To conclude this section, the participants where asked to rate whether or not the proposed architecture is operable without external dependencies, once it has been set up. Once again, all six participants strongly agreed with this proposition, i.e. gave a 5 out of 5 rating.

7.3.4 Additional Inputs

The interviews concluded with an open question asking for any further suggestions or possible improvements. The most common feedback was that the name "shadow project" was confusing. Some notable suggestions were "mining project", "visualisation project", "vault" and "mine".

Once participant suggested that commands executed in the Dockerfile should be chained to avoid creating a new intermediary image for each command. Moreover, it was discussed whether or not putting the mining script in the Docker image would make sense. However, since GitLab automatically clones the repository when starting a CI/CD job, and the mining script has to live in some repository anyway, the author decided against it.

Lastly, it was noted that an additional option to support mining within an existing repository, i.e. without a shadow project, would be desirable.

7.4 Summary

All in all, the hypotheses about a suitable architecture for software repository mining and visualisation seem valid. Looking at the distributions of ratings to the hypotheses,

as shown in Figure 7.2, further supports this argument. Each hypothesis had at least one expert who rated it 5 out of 5, i.e. strongly agreed with it. Moreover, even the lowest rated hypothesis had a median rating of 3.5 out of 5, which indicates that the participants rather agree than disagree.

The evaluation of the architecture also yielded very pleasant results. The participants gave a median rating of 5 out of 5 ("strongly agree") on seven of the eight propositions, with an additional median rating of 4.5 out of 5 on the final statement. In addition, the qualitative feedback given by the interviewees resembled these ratings. Multiple participants even asked when the architecture would become operational in the undergraduate software engineering course at TU Wien. Even though the architecture has only been realised as a proof of concept, therefore is far from being applicable, that is probably the best praise imaginable.

7.5 Discussion

As mentioned in Section 7.1, the goal of the evaluation interviews was to answer the remaining research questions. RQ3 is concerned with determining a suitable architecture for in-process software repository mining and visualisation. After reviewing the literature, seven hypothesis about such an architecture were constructed. All of these hypotheses were validated during the evaluation interviews, as described in Section 7.3.2. As a consequence, a suitable architecture for in-process software repository mining and visualisation should implement all ideally implement all seven of them.

RQ4.1 aims at answering how purposeful the stakeholders rate the proposed architecture and process. The ratings presented in Section 7.3.3 show that the stakeholders involved in software engineering education think that the proposed architecture and process fulfils the characteristics identified in RQ3. It can therefore be concluded that the stakeholders see the proposed architecture and process as purposeful for in-process software repository mining and visualisation.

Lastly, RQ4.2 asks how the stakeholders rate its ability to satisfy the information needs in software engineering education, which have been identified in Section 4.2. While the proposed architecture and process does not directly satisfy these information needs, the visualisations of Binocular do, it allows the stakeholders in software engineering education to easily access these visualisations. Since the conclusion of RQ4.1 was that stakeholders see the proposed architecture and process as purposeful for in-process software repository mining and visualisation, the purposefulness to satisfy the information needs in software engineering education is tied to Binocular's ability to satisfy these information needs. The most useful insight, according to the interviewees, were insights on the work distribution between students. The participants rated Binocular's ability, more specifically the ability of its "Dashboard" visualisation, to gain these insights with a median rating of 3.5 out of 5. Code ownership over time was rated as the second most important insight. This information is illustrated in the "Code Ownership River" visualisation. Insights on large spikes in work contributions received the third highest rating. Its "Dashboard"

visualisation was rated "very purposeful" (5 out of 5) by three of the four interviewees. Moreover, purposefulness of the "HotspotDials" to gain these insight received a median rating of 3 out of 5, with one participant even rating it 5 out of 5 ("very purposeful"). As for the remaining two information needs, Binocular currently neither offers visualisations to see the number of active conflicts, nor the history of conflicts within the source code.

7.6 Threats to Validity

The threats to validity of the evaluation round are very similar to those of the semi-structured interviews, which are described in Section 4.1.3. The results of the evaluation round cannot be seen as representative, due to the small sample size of six participants. Again, bias is introduced, since all participants were from the immediate professional environment of the advisor of this thesis. Moreover, the live demos introduced further biased, since they were performed by the author of this thesis.

The aims of the evaluation round were to determine characteristics of a suitable architecture for software repository mining and visualisation and to assess the purposefulness of the proposed architecture. The former was done by validating various hypotheses that were constructed. Due to the complexity of the proposed architecture, an introduction had to be done at some point during the evaluation interviews. This introduction made the most sense in the beginning, since it was required before the live demo. Moreover, by placing it at this point in the interview, the questionnaire could be filled out in one go. As a result the final architecture was introduced before hypotheses were validated, which could have additional further bias.

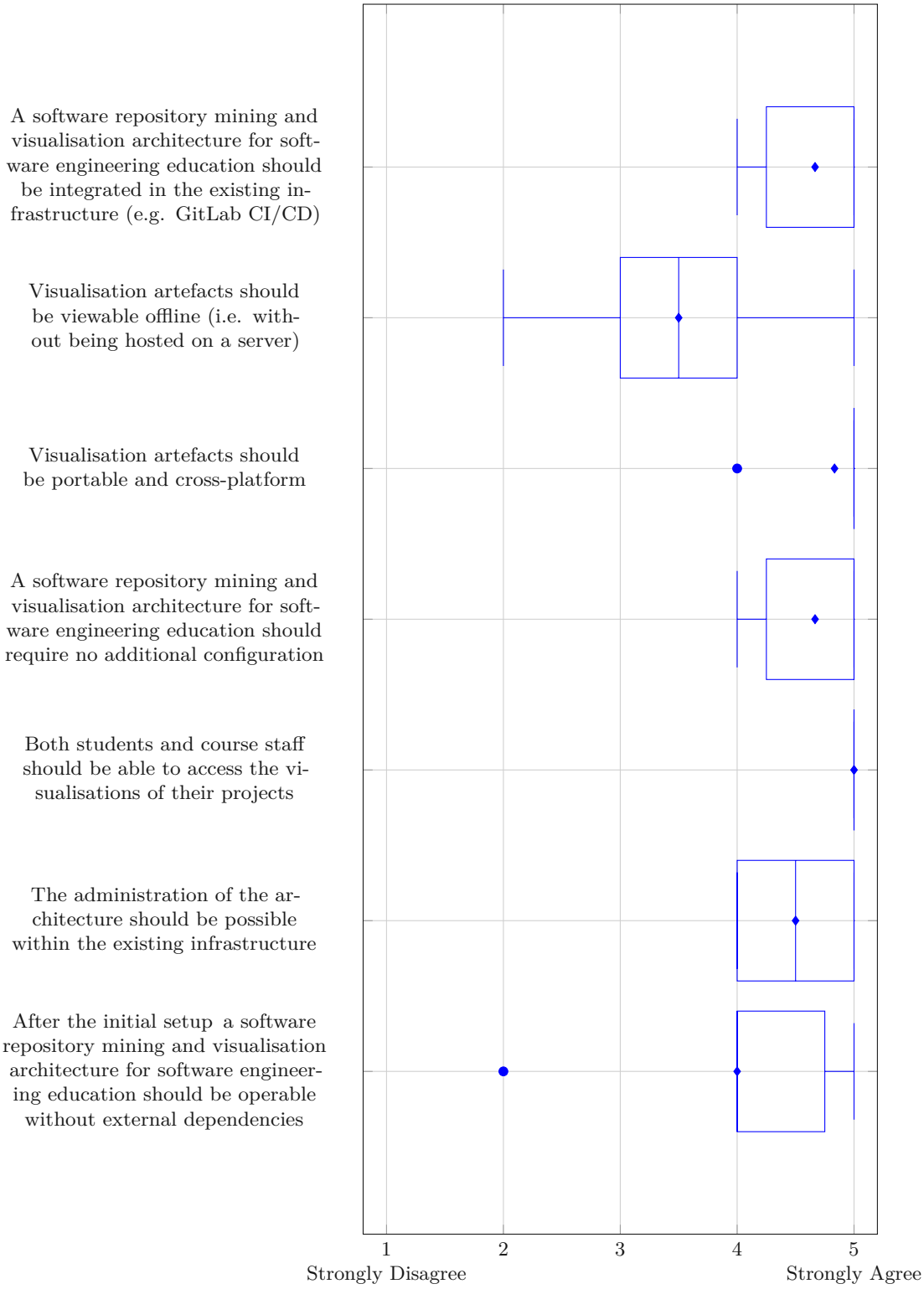


Figure 7.2: Responses to the questions of the "Hypotheses" section

Figure 7.3: Responses to the questions of the "Evaluation" section

CHAPTER 8

Conclusion

This thesis compasses two main contributions. Firstly, the research project Binocular[25] was evaluated to identify its limitations in educational software engineering settings and to better understand the information needs in software engineering education. Secondly, an integrated software repository mining and visualisation approach was introduced. While the solution was designed and implemented in an educational setting, its applicability is not limited to software engineering education.

A literature review has shown that, while there exists lots of research on the information needs in software engineering, none of it is specific to software engineering education. Moreover, current software repository mining and visualisation tools either require external infrastructure or further integration before they become applicable. This would be the essence of an answer to RQ1.

To better understand the information needs in software engineering education, semi-structured interviews with four members of the undergraduate software engineering course at TU Wien were conducted. Moreover, the limitations of Binocular in educational software engineering environments should be identified. The interviews gave lots of awareness on these shortcomings, as described in Section 4.1.2. The information needs that were derived from these interviews are mentioned in Section 4.2. These insights were used to answer RQ2.1 and RQ2.2 respectively.

Before a solution could be designed and implemented, requirements had to be defined. In order to define these requirements, various hypotheses of a suitable architecture for in-process software repository mining and visualisation were constructed. These hypotheses were then used to derive requirements for the proposed solution. The final requirements are listed in Section 5.

To proof that the proposed architecture and process is indeed feasible, a proof of concept based on GitLab CI/CD was implemented. Moreover, Binocular was adapted to produce

portable, offline visualisation artefacts. The implementation of this proof of concept is described in Section 6.

To evaluate the solution, six interviews with members of the undergraduate software engineering course at TU Wien were conducted. The goal was to validate the hypotheses about a suitable architecture for in-process software repository mining and visualisation and to rate the purposefulness of the proposed solution. Section 7 describes the goals, method and results of these six interviews. The insights from these evaluation interviews were used to answer RQ3, RQ4.1 and RQ4.2.

8.1 Future Work

Due to the limited scope of this thesis, not all ideas that were explored could be realised. This section lists possibilities for future work in this field.

Since the architecture was implemented as a proof of concept, it is quite far from being applicable in practise. First of all, the mining pipeline within the shadow projects would need to be scheduled in some way to be useful. GitLab's scheduled pipelines[1] would most likely be the easiest solution for this. While this could be done manually on a project basis, scheduling the pipelines through the API[2] when initially creating the shadow project would be favourable. Moreover, due to the prototypical implementation of the proof of concept, there most likely is room for improvement within the source code. One example would be the chaining of commands in the Dockerfile, as mentioned in Section 7.3.4.

Moreover, the proof of concept currently only supports the "Dashboard" visualisation of Binocular. An obvious next step would be to port the five remaining visualisations. This would the replacement of the GraphQL queries within the `sagas` folder of the various visualisations. These queries would have to be replaced with Mango queries, as described in Section 6.1.

Another logical next step is the integration of further software repository mining and visualisation tools into the template project, thus made available in any subsequently created shadow projects. Section 3.2.2 and Section 3.2.3 list various tools that could be considered for integration into the architecture.

Furthermore, as mentioned in Section 7.3.4, an option to add the mining logic to a given project, without the need of setting up shadow projects, would be desirable. While the architecture was specifically chosen to work the way it does, more specifically a non-invasive manner, there certainly are use cases where one might want the mining logic within the existing project.

[1] `https://docs.gitlab.com/ee/ci/pipelines/schedules.html`, last accessed on 13.03.2022

[2] `https://docs.gitlab.com/ee/api/pipeline_schedules.html`, last accessed on 13.03.2022

Due to the nature of a software engineering course, where all students typically work on identical, or at least similar projects, there were also conversations about anonymous comparisons or rankings within a course. An example could be something along the lines of "95% of groups have a higher test coverage than your team". This idea could also be explored in the future.

Bibliography

[1] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Software professionals' information needs in continuous integration and delivery. pages 1513–1520. ACM, 2021.

[2] Konstantinos Barmpis, Patrick Neubauer, Jonathan Co, Dimitris Kolovos, Nicholas Matragkas, and Richard F. Paige. Polyglot and Distributed Software Repository Mining with Crossflow. pages 374–384. ACM, 2020.

[3] Matthew Barr, Syed Waqar Nabir, and Derek Somerville. Online Delivery of Intensive Software Engineering Education during the COVID-19 Pandemic. pages 1–6. IEEE, 2020.

[4] Matthew Bass. Software Engineering Education in the New World: What Needs to Change? pages 213–221. IEEE, 2016.

[5] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Situational awareness: Personalizing issue tracking systems. pages 1185–1188. IEEE, 2013.

[6] Kathy Beckman, Neal Coulter, Soheil Khajenoori, and Nancy R. Mead. Collaborations: closing the industry-academia gap. *IEEE Software*, pages 49–57, 1997.

[7] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. *Proceedings - International Conference on Software Engineering*, pages 12–23, 2014.

[8] Mario Bernhart, Thomas Grechenig, Jennifer Hetzl, and Wolfgang Zuser. Dimensions of software engineering course design. pages 667–672. ACM Press, 2006.

[9] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. pages 1–10. IEEE, 2009.

[10] Raymond P. L. Buse and Thomas Zimmermann. Information needs for software development analytics. pages 987–996. IEEE Press, 2012.

[11] Scott Chacon and Ben Straub. In *Pro Git*, pages 8–11. Apress, 2014.

[12] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. Software history under the lens: A study on why and how developers examine it. pages 1–10, 2015.

[13] Aldo Dagnino. Increasing the effectiveness of teaching software engineering: A university and industry partnership. pages 49–54. IEEE Computer Society, 2014.

[14] Wim de Pauw, Steven P. Reiss, and John T. Stasko. ICSE workshop on software visualization. *Proceedings - International Conference on Software Engineering*, pages 758–759, 2001.

[15] Nitish M. Devadiga. Software Engineering Education: Converging with the Startup Industry. pages 192–196. IEEE, 2017.

[16] Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. Evaluating a Visual Approach for Understanding JavaScript Source Code. pages 128–138. ACM, 2020.

[17] Louwarnoud Van Der Duim, Jesper Andersson, and Marco Sinnema. Good practices for Educational Software Engineering Projects, 2007.

[18] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. pages 422–431. IEEE, 2013.

[19] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Transactions on Software Engineering and Methodology*, pages 1–34, 2015.

[20] Brian P. Eddy, Norman Wilde, Nathan A. Cooper, Bhavyansh Mishra, Valeria S. Gamboa, Keenal M. Shah, Adrian M. Deleon, and Nikolai A. Shields. A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses. pages 47–56. IEEE, 2017.

[21] Aron Fiechter, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing GitHub Issues. pages 155–159. IEEE, 2021.

[22] Peter Freeman and Anthony I. Wasserman. A Proposed Curriculum for Software Engineering Education. pages 56–62. IEEE Press, 1978.

[23] Gabriela Dorfman Furman and Zeev Weissman. On Adding Interdisciplinary Elements to the Classical Engineering Studies. pages 684–687. IEEE, 2020.

[24] Nicolas E. Gold and Jens Krinke. Ethical Mining: A Case Study on MSR Mining Challenges. pages 265–276. ACM, 2020.

[25] Johann Grabner, Roman Decker, Thomas Artner, Mario Bernhart, and Thomas Grechenig. Combining and Visualizing Time-Oriented Data from the Software Engineering Toolset. pages 76–86. IEEE, 2018.

[26] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. pages 1–5. ACM Press, 2013.

[27] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. A quantitative analysis of developer information needs in software ecosystems. ACM Press, 2014.

[28] Ahmed E. Hassan, Richard C. Holt, and Audris Mockus. MSR 2004: International Workshop on Mining Software Repositories. pages 770–771. IEEE Computer Society, 2004.

[29] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. pages 426–437. ACM, 2016.

[30] James Howison and Kevin Crowston. The perils and pitfalls of mining SourceForge. *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, 2004.

[31] Che Shian Hung and Robert Dyer. Boa Views: Easy Modularization and Sharing of MSR Analyses. pages 147–157. ACM, 2020.

[32] Jitendra Josyula, Sarat Panamgipalli, Muhammad Usman, Ricardo Britto, and Nauman Bin Ali. Software practitioners' information needs and sources: A survey study. pages 1–6. IEEE, 2019.

[33] Sascha Just, Kim Herzig, Jacek Czerwonka, and Brendan Murphy. Switching to Git: The Good, the Bad, and the Ugly. pages 400–411. IEEE, 2016.

[34] Eirini Kalliamvakou, Leif Singer, Georgios Gousios, Daniel M. German, Kelly Blincoe, and Daniela Damian. The promises and perils of mining GitHub, 2014.

[35] Tanjila Kanij and John Grundy. Adapting Teaching of a Software Engineering Service Course Due to COVID-19. pages 1–6. IEEE, 2020.

[36] Taeyoung Kim, Suntae Kim, and Duksan Ryu. Coding™: Development Task Visualization for SW Code Comprehension. pages 23–32. IEEE, 2021.

[37] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. pages 344–353, 2007.

[38] Fabian Kortum, Jil Klunder, Oliver Karras, Wasja Brunotte, and Kurt Schneider. Which Information Help agile Teams the Most? An Experience Report on the Problems and Needs. pages 306–313. IEEE, 2020.

[39] Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. Mining file histories: should we consider branches? pages 202–213. ACM, 2018.

[40] Bart Luijten, Joost Visser, and Andy Zaidman. Assessment of issue handling efficiency. pages 94–97. IEEE, 2010.

[41] Mircea Lungu. Towards reverse engineering software ecosystems. pages 428–431. IEEE, 2008.

[42] Anna Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väätäjä. Software visualization today - Systematic literature review. pages 262–271. ACM, 2016.

[43] Luc Moreau and Paul Groth. In *Provenance: An Introduction to PROV*, pages 3–3. Morgan & Claypool Publishers, 2013.

[44] Malte Mues and Falk Howar. Teaching a Project-Based Course at a Safe Distance: An Experience Report. pages 1–6. IEEE, 2020.

[45] Richard Müller, Dirk Mahler, Michael Hunger, Jens Nerche, and Markus Harrer. Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization. pages 107–111. IEEE, 2018.

[46] Oystein Nytro, Anh Nguyen-Duc, Hallvard Tratteberg, Madeleine Loras, and Babak Amin Farschian. *Unreined Students or Not: Modes of Freedom in a Project-Based Software Engineering Course*. 2020.

[47] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction*, 2018.

[48] Jari Porras, Jayden Khakurel, Jouni Ikonen, Ari Happonen, Antti Knutas, Antti Herala, and Olaf Drögehorn. Hackathons in software engineering education: Lessons learned from a decade of events. pages 40–47. IEEE Computer Society, 2018.

[49] Steven P. Reiss. The paradox of software visualization. pages 59–63. IEEE, 2005.

[50] Sayed Mohsin Reza, Omar Badreddin, and Khandoker Rahad. ModelMine: A tool to facilitate mining models from open source repositories. pages 441–450. Association for Computing Machinery, Inc, 2020.

[51] Gregorio Robles. Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories Proceedings. pages 171–180. IEEE, 2010.

[52] Tommaso Dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. Blended, not stirred: Multi-concern visualization of large software systems. pages 106–115. IEEE, 2015.

[53] Paul Schmiedmayer, Lara Marie Reimer, Marko Jovanovic, Dominic Henze, and Stephan Jonas. Transitioning to a Large-Scale Distributed Programming Course. pages 1–6. IEEE, 2020.

[54] Andreas Schreiber, Lynn von Kurnatowski, Annika Meinecke, and Claas de Boer. An Interactive Dashboard for Visualizing the Provenance of Software Development Processes. pages 100–104. IEEE, 2021.

[55] Yvonne Sedelmaier and Dieter Landes. Active and Inductive Learning in Software Engineering Education. pages 418–427. IEEE Computer Society, 2015.

[56] Mary Shaw. Software engineering education: A roadmap. pages 371–380. ACM Press, 2000.

[57] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. pages 908–911. ACM, 2018.

[58] Hagen Tamer, Daniel van den Bongard, and Fabian Beck. Visually Analyzing the Structure and Code Quality of Component-based Web Applications. pages 160–164. IEEE, 2021.

[59] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? - An exploratory study in industry. pages 51:1–51:11. ACM, 2012.

[60] Nitin M. Tiwari, Ganesha Upadhyaya, Hoan Anh Nguyen, and Hridesh Rajan. Candoia: A platform for building and sharing mining software repositories tools as apps. pages 53–63. IEEE, 2017.

[61] Jari Vanhanen, Timo O. A. Lehtinen, and Casper Lassenius. Teaching real-world software engineering through a capstone project course with industrial customers. pages 29–32, 2012.

[62] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. pages 805–816. ACM, 2015.

[63] Richard Wettel and Michele Lanza. Program Comprehension through Software Habitability. pages 231–240. IEEE, 2007.

[64] Claes Wohlin and Björn Regnell. Achieving industrial relevance in software engineering education. pages 16–25. IEEE, 1999.

[65] Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Educational software engineering: Where software engineering, education, and gaming meet. pages 36–39, 2013.

Appendix

Questionnaire: Information Needs in Software Engineering Education

Information Needs in Software Engineering Education

A survey on information needs in software engineering education.

* Required

Demographics

1. What is your age? *

Mark only one oval.

- ◯ 18-24 years old
- ◯ 25-34 years old
- ◯ 35-44 years old
- ◯ 45-54 years old
- ◯ 55-64 years old
- ◯ 65 years or older

2. What is your gender? *

Mark only one oval.

- ◯ Female
- ◯ Male
- ◯ Prefer not to say
- ◯ Other: _____

3. How long have you been working in software engineering education? *

Mark only one oval.

- ◯ < 1 year
- ◯ 1-2 years
- ◯ 2-5 years
- ◯ 5-10 years
- ◯ 10-15 years
- ◯ 15-20 years
- ◯ 20+ years

4. What is your role in this course? *

Concepts

5. How useful are insights about large spikes in work contributions in software engineering education? *

Mark only one oval.

	1	2	3	4	5	
Not useful	◯	◯	◯	◯	◯	Very useful

6. How useful are insights about work distribution between students in software engineering education? *

Mark only one oval.

	1	2	3	4	5	
Not useful	◯	◯	◯	◯	◯	Very useful

7. How useful are insights about the number of active conflicts within the source code, e.g. on different branches? *

Mark only one oval.

	1	2	3	4	5	
Not useful	◯	◯	◯	◯	◯	Very useful

8. How useful are insights about the history of conflicts within the source code, e.g. on different branches? *

Mark only one oval.

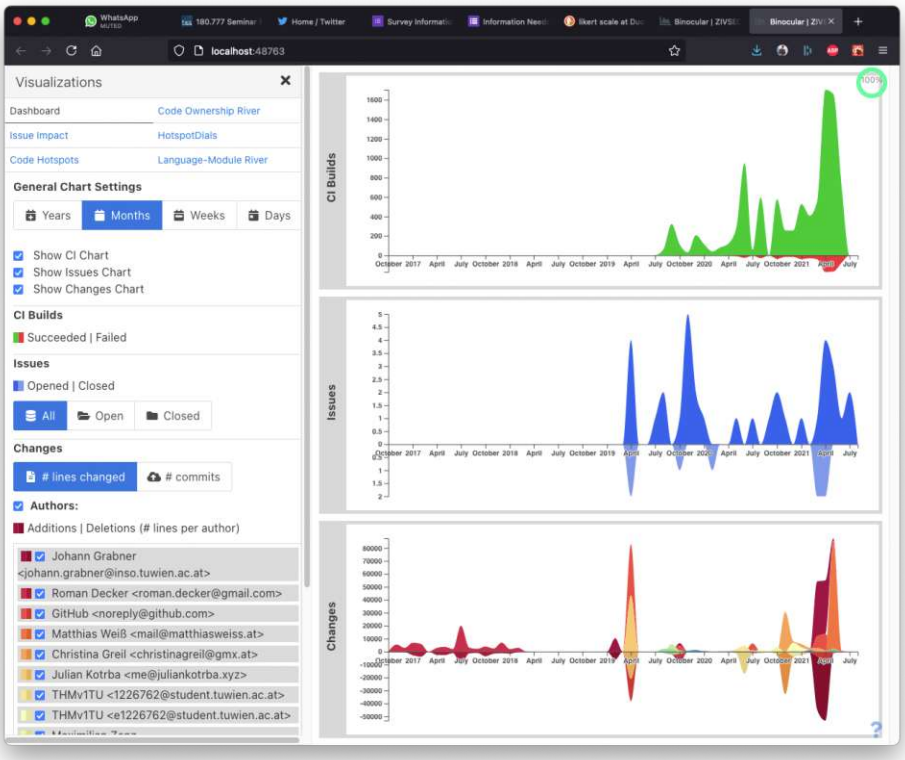
	1	2	3	4	5	
Not useful	◯	◯	◯	◯	◯	Very useful

9. How useful are insights about the code ownership over time by each student? *

Mark only one oval.

	1	2	3	4	5	
Not useful	◯	◯	◯	◯	◯	Very useful

Dashboard Visualisation



10. How purposeful is the Dashboard visualisation for your role in this course? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

11. How could it be improved to ease your role?

12. How helpful is the Dashboard visualisation for software engineering education? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

13. How can this visualisation be utilised in software engineering education? *

14. What are its shortcomings in software engineering education? *

15. How purposeful is the dashboard visualisation to gain insights about large spikes in work contributions? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

16. How could it be improved to gain the relevant insights?

17. How purposeful is the Dashboard visualisation to gain insights about work distribution between students? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

18. How could it be improved to gain the relevant insights?

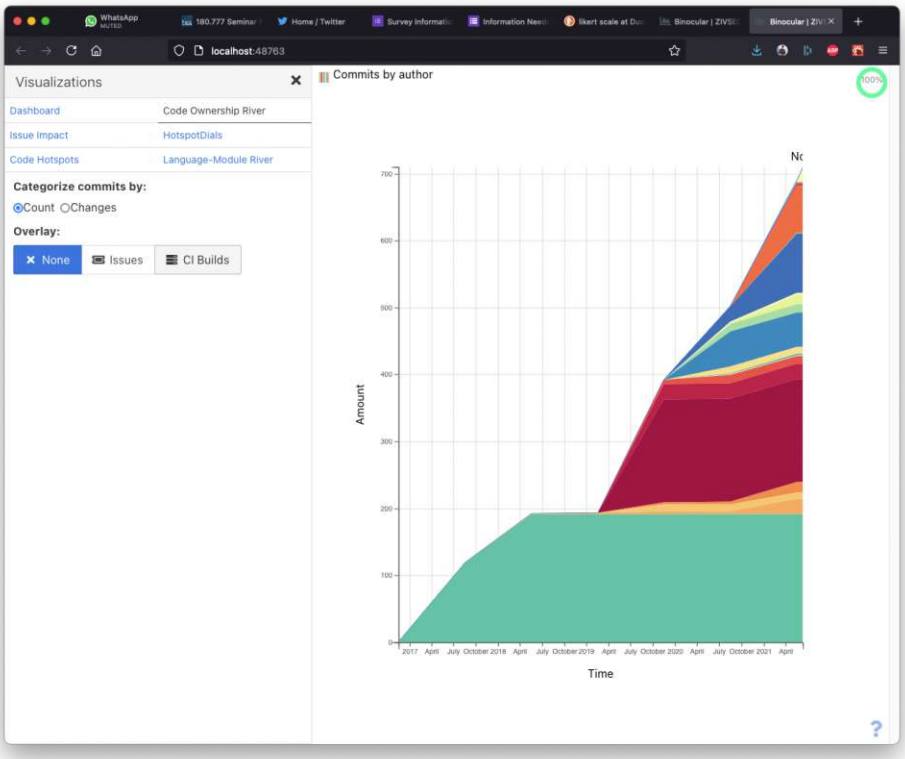
19. How purposeful is the Dashboard visualisation to gain insights about the amount of added/deleted source code by each student? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

20. How could it be improved to gain the relevant insights?

Code Ownership River Visualisation



21. How purposeful is the Code Ownership River visualisation for your role in this course? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

22. How could it be improved to ease your role?

23. How helpful is the Code Ownership River visualisation for software engineering education? *

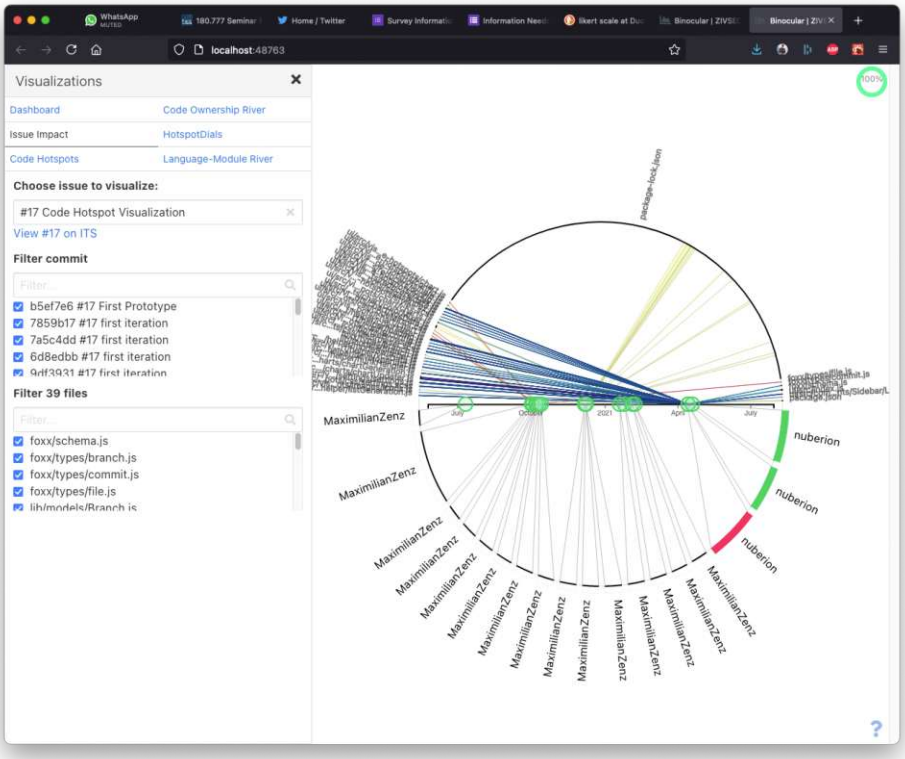
Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

24. How can this visualisation be utilised in software engineering education? *

25. What are its shortcomings in software engineering education? *

Issue Impact Visualisation



26. How purposeful is the Issue Impact visualisation for your role in this course? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

27. How could it be improved to ease your role?

28. How helpful is the Issue Impact visualisation for software engineering education? *

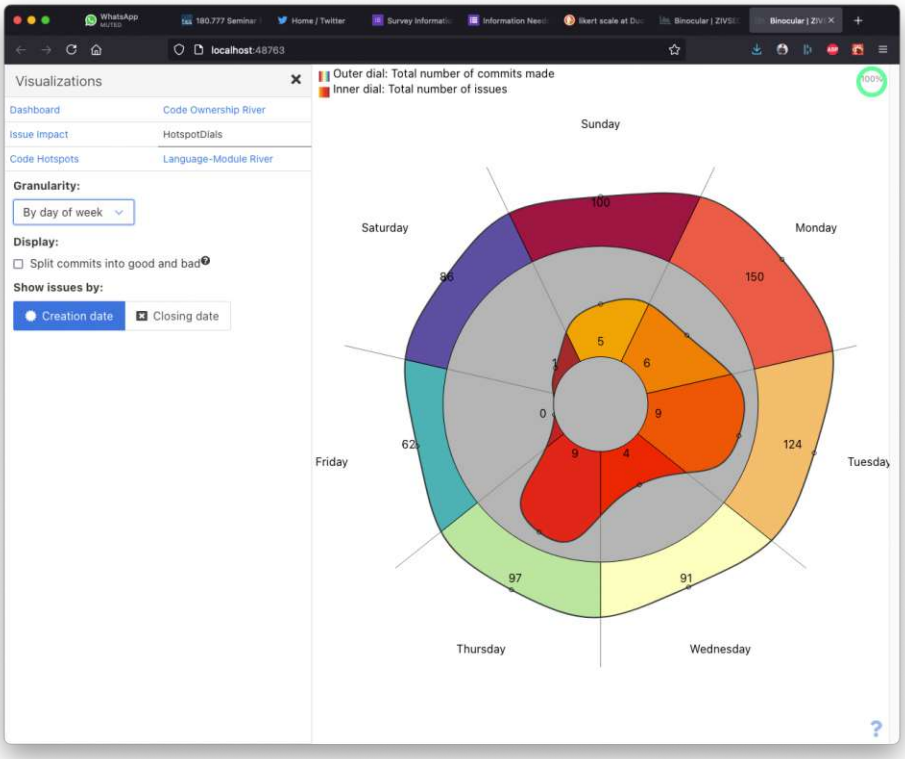
Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

29. How can this visualisation be utilised in software engineering education? *

30. What are its shortcomings in software engineering education? *

HotspotDials Visualisation



31. How purposeful is the HotspotDials visualisation for your role in this course? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

32. How could it be improved to ease your role?

33. How helpful is the HotspotDials visualisation for software engineering education? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

34. How can this visualisation be utilised in software engineering education? *

35. What are its shortcomings in software engineering education? *

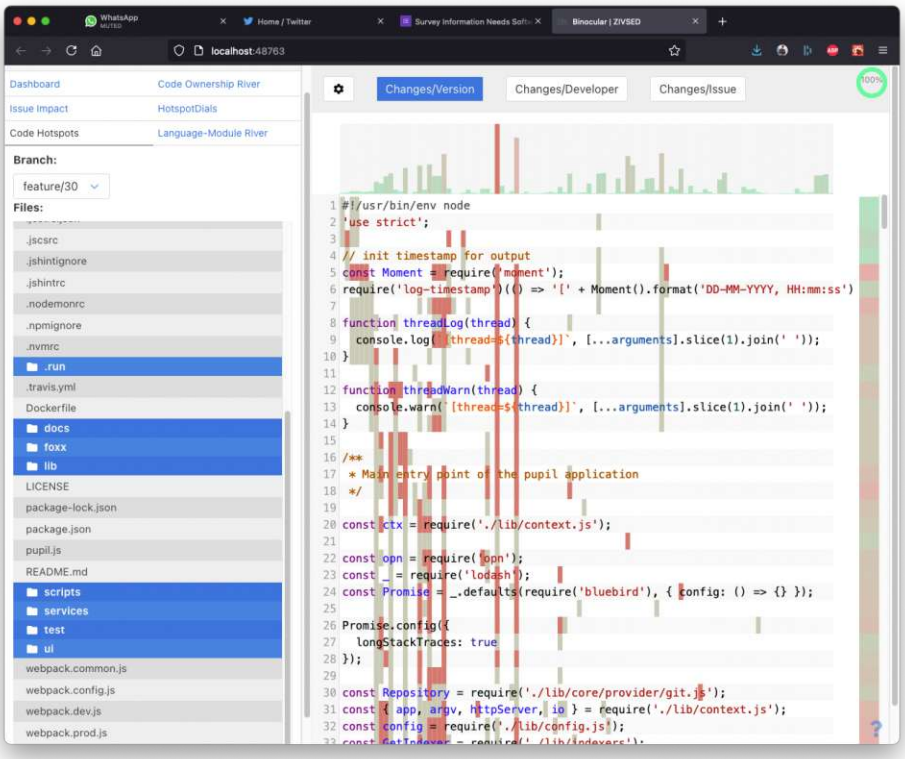
36. How purposeful is the HotspotDials visualisation to gain insights about large spikes in work contributions? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

37. How could it be improved to gain the relevant insights?

Code Hotspots Visualisation



38. How purposeful is the Code Hotspots visualisation for your role in this course? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

39. How could it be improved to ease your role?

40. How helpful is the Code Hotspots visualisation for software engineering education? *

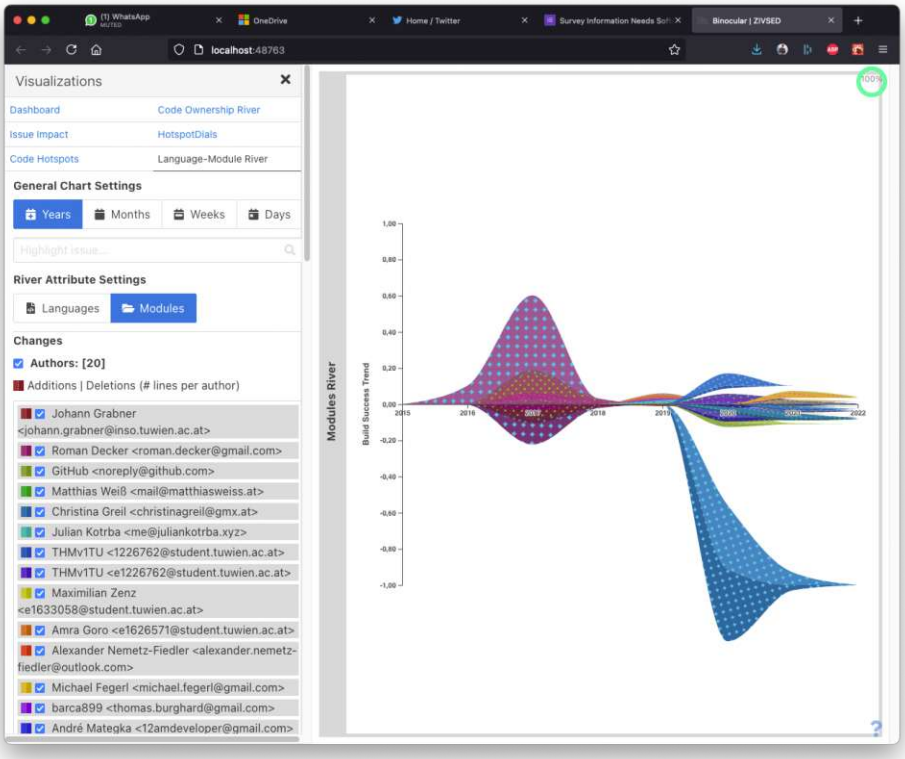
Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

41. How can this visualisation be utilised in software engineering education? *

42. What are its shortcomings in software engineering education? *

Language-Module River Visualisation



43. How purposeful is the Language-Module River visualisation for your role in this course? *

Mark only one oval.

	1	2	3	4	5	
Not purposeful	◯	◯	◯	◯	◯	Very purposeful

44. How could it be improved to ease your role?

45. How helpful is the Language-Module River visualisation for software engineering education? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

46. How can this visualisation be utilised in software engineering education? *

47. What are its shortcomings in software engineering education? *

Additional inputs

48. Are there any additional insights that software engineering education would benefit from? *

Questionnaire: Evaluation of a Software Repository Mining and Visualisation Approach

Evaluation of a Software Repository Mining and Visualisation Approach

Evaluation of a software repository mining and visualisation approach.

* Required

Demographics

1. What is your age? *

Mark only one oval.

- ◯ 18-24 years old
- ◯ 25-34 years old
- ◯ 35-44 years old
- ◯ 45-54 years old
- ◯ 55-64 years old
- ◯ 65 years or older

2. What is your gender? *

Mark only one oval.

- ◯ Female
- ◯ Male
- ◯ Prefer not to say
- ◯ Other: _____

3. How long have you been working in software engineering education? *

Mark only one oval.

- ◯ < 1 year
- ◯ 1-2 years
- ◯ 2-5 years
- ◯ 5-10 years
- ◯ 10-15 years
- ◯ 15-20 years
- ◯ 20+ years

4. What is your role in this course?

_____

_____

_____

_____

_____

Hypotheses

5. A software repository mining and visualisation architecture for software engineering education should be integrated in the existing infrastructure (e.g. GitLab CI/CD) *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

6. Visualisation artefacts should be viewable offline (i.e. without being hosted on a server) *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

7. Visualisation artefacts should be portable and cross-platform *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

8. A software repository mining and visualisation architecture for software engineering education should require no additional configuration *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

9. Both students and course staff should be able to access the visualisations of their projects *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

10. The administration of the architecture should be possible within the existing infrastructure *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

11. After the initial setup, a software repository mining and visualisation architecture for software engineering education should be operable without external dependencies *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Evaluation

12. The concept of shadow projects is easy to grasp *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

13. The proposed architecture is integrated in the existing infrastructure *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

14. The visualisation artefacts are easily viewable offline (i.e. without being hosted on a server) *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

15. The visualisation artefacts are portable and cross-platform *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

16. The proposed architecture requires no additional configuration *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

17. Both students and course staff can access the visualisations of their projects *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

18. Creating shadow projects for existing projects is straightforward *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

19. After the initial setup, the proposed architecture is operable without external dependencies *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Additional inputs

20. How could the proposed approach be improved?