Teaching Mining Software Repositories

Zadia Codabux, Fatemeh Fard, Roberto Verdecchia, Fabio Palomba, Dario Di Nucci, Gilberto Recupito

Abstract

Mining Software Repositories (MSR) has become a popular research area recently. MSR analyzes different sources of data, such as version control systems, code repositories, defect tracking systems, archived communication, deployment logs, and so on, to uncover interesting and actionable insights from the data for improved software development, maintenance, and evolution. This chapter provides an overview of MSR and how to conduct an MSR study, including setting up a study, formulating research goals and questions, identifying repositories, extracting and cleaning the data, performing data analysis and synthesis, and discussing MSR study limitations. Furthermore, the chapter discusses MSR as part of a mixed method study, how to mine data ethically, and gives an overview of recent trends in MSR as well as reflects on the future. As a teaching aid, the chapter provides tips for educators, exercises for students at all levels, and a list of repositories that can be used as a starting point for an MSR study.

Zadia Codabux
University of Saskatchewan, Canada, e-mail: zadiacodabux@ieee.org

Fatemeh Fard
University of British Columbia, Canada, e-mail: fatemeh.fard@ubc.ca

Roberto Verdecchia
University of Florence, Italy, e-mail: roberto.verdecchia@unifi.it

Fabio Palomba
University of Salerno, Italy, e-mail: fpalomba@unisa.it

Dario Di Nucci
University of Salerno, Italy, e-mail: ddinucci@unisa.it

Gilberto Recupito
University of Salerno, Italy, e-mail: grecupito@unisa.it

1 Introduction

Embedded within the intricate software development landscape lies Mining Software Repositories (MSR), an academic discipline dedicated to delving into the expansive dataset within software repositories. Software Repositories are among the most common sources of data that enable the analysis of software properties. Additionally, the promising success that this type of study had in the research community allowed to extend the mining data process beyond the software repositories, encompassing various essential aspects such as software changes, collaborative efforts, socio-technical aspects [GKS08, Men16], and the process dynamics [PSVDB11]. Combining these aspects serves to obtain the source of key information detailing the transformation of a software application from its preliminary stages to well-defined and sophisticated software.

During the iterative enhancement of software development, developers wield powerful tools that meticulously track software information. These tools, including robust version control systems like Git and Subversion (SVN) and efficient issue-tracking tools like JIRA and Bugzilla, facilitate collaboration and provide an exhaustive lineage of code changes. This comprehensive history, resulting from the union of these tools, allows developers to gain profound insights into all key aspects of the software evolution project, including software modifications, issue resolutions, bug fixes, and strategic refactoring initiatives, which are contained in the software repository [KCM07]. The definition of this dynamic and extensive source of information lays the basis for introducing MSR. MSR is the field that exploits the information contained in software repositories to conduct thorough investigation and analysis, unveiling aspects of the software development process [Has08].

MSR studies in the software engineering research field demonstrated the underlying capabilities of repository analysis. Several studies investigate the possibility of building prediction models for software quality issues. Prediction models could be built from issue tracking labels to detect bugs [MSRM04, RK11]. Moreover, the extreme amount of data possible to extract from source code repositories allows us to build complex and useful models with the use of Large Language Models, enabling us to build useful tools that perform programming tasks related to code generation, summarization, and analysis, also producing code that is not complex and highly maintainable (e.g., Copilot[1]) [NN22]. This chapter is intended to **target educators at MSc and Ph.D. levels**, providing them with a comprehensive understanding of MSR and its applications in software engineering research so that they may be equipped to teach this method. More specifically, the **learning objectives** include:

1. Understand **how to set up a mining software repository study**, uncovering the methodological design steps required to (i) define the objec-

[1] Github Copilot: https://github.com/features/copilot/

tives of an MSR study; (ii) identify suitable data sources and data cleaning methods to address the objectives; (iii) select the most appropriate code analysis instruments to extract relevant pieces of information useful to the analysis; and (iv) analyze and synthesize the data gathered throughout the mining process;
2. Realize **what are the typical threats to the validity involving mining software repository studies**, overviewing how multiple issues concerned with the mining and analysis of data sources might bias the interpretation of the results;
3. Identify **how to combine mining software repository studies with additional research methods**, discussing the flexibility of MSR research in addressing complex research objectives;
4. Indicate **what are the ethical considerations concerned with mining software repository studies**, uncovering the possible limitations of this research method, other than the responsibilities that the future generation of researchers must necessarily take into account when designing similar studies.

To ease the reader's understanding of the concepts in the chapter, we will focus on specific cases regarding extracting data from the source code of software repositories. Specifically, we will often refer to an exemplar case, which effectively used the methodologies described in the chapter. Among the vast array of valuable MSR examples in literature, we decided to focus on the work by Kamei et al. [KSA+13], published in IEEE Transactions on Software Engineering in 2012.

The study explored a well-known and well-established research theme in MSR research, namely *defect prediction*. This is a technique used to identify defect-prone files or packages throughout software development. More particularly, the study introduced the term *"Just-In-Time Quality Assurance"* and aimed at identifying defect-prone software changes while the developers commit their changes onto a shared repository. From a methodological standpoint, the study heavily relied on MSR instruments, as it was required to feed machine-learning models with features mined from software repositories. These features encompassed multiple properties that might be extracted from version control systems, such as product, process, and developer-oriented metrics, hence representing an ideal example to make the concepts in this chapter more practical. In addition, the study carefully designed the experimental procedures, touching on various data selection, cleaning, and analysis processes that are elaborated upon in this chapter.

Exemplary **teaching strategies** that might be relevant for teaching this chapter include, but are not limited, to the following:

- *Interactive discussions*. Educators may engage students in active discussions about the importance of MSR in software engineering research, its potential applications, and its limitations. Through these discussions, we

envision students to acquire awareness of when and how to use MSR instruments to empower their analyses;

- *Case studies.* Educators may use our chapter to present real-world examples of MSR research, such as the study by Kamei et al. [KSA+13], to illustrate key concepts and methodologies, making students aware of how the design choices taken in exemplary cases impacted the type of analysis and results of MSR studies;
- *Hands-on exercises.* Educators may provide students with hands-on experience in conducting MSR studies, including data collection, cleaning, and analysis, using relevant tools and datasets. Educators may require students to design an MSR study, producing reports that detail the whole set of activities and the rationale thereof required to address specific research objectives;
- *Group projects.* Educators may assign group projects where students can apply MSR techniques to analyze software repositories, like those reported at the end of this chapter, and address specific research questions or challenges. Using group projects, students might jointly design data collection, cleaning, and analysis steps for specific research objectives, having the chance to interact with other students, hence possibly increasing the collective awareness of the most appropriate methodological choices to take, even considering ethical limitations.

2 Setting up a Source Code Repository Mining Study

This section will overview the MSR process, including the study design, data extraction, and analysis, as illustrated in Figure 1. We will discuss the considerations for data extraction (e.g., API vs. Package) and storage. Since these steps are typically included to guarantee a systematic mining process, the definition of it might vary depending strictly on the goal of the study that an MSR researcher aims to design. Regarding data analysis, we will focus on aspects that should be considered and errors that could occur about the storage, dataset size, types of analysis, emergent problems, and cases of data contradicting Research Questions (RQs) or formulated hypotheses.

2.1 MSR Study Design

The first step to designing an MSR study should be systematically defining the research goal. To do so, the widely adopted Goal-Question-Metric (GQM) paradigm of Basili [Bas94] can be used. Designing the objective of a study

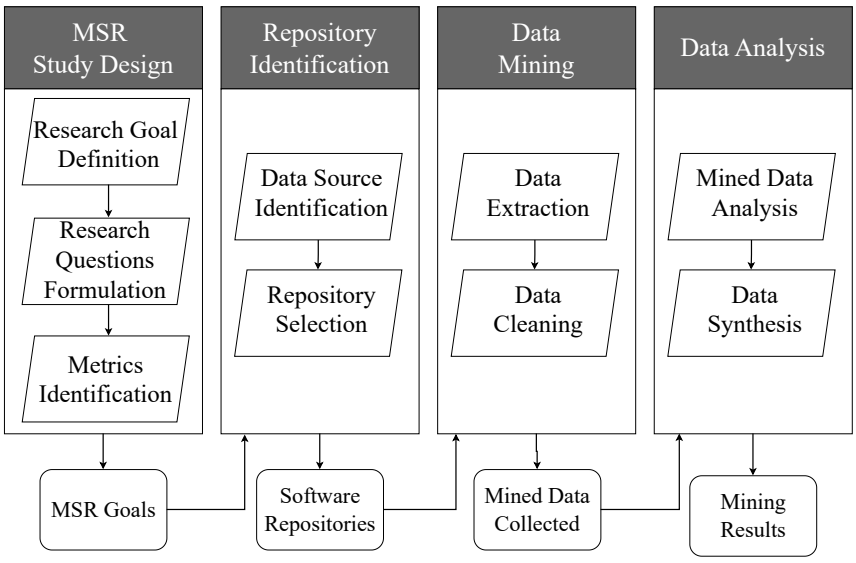

Fig. 1: Set of Activities Involved in the MSR Study Process.

via the GQM paradigm is composed of three main steps, namely (i) the formulation of the goal of the research, (ii) the design of the RQs that need to be answered to achieve the goal, and (iii) the definition of the metrics necessary to answer the goal.

2.1.1 Research goal formulation

The formulation of the research goal of a mining study through the GQM paradigm follows a systematic structure. More specifically, the goal is designed by adopting the following template:

> **Analyze** *the experimental object(s) of the study, e.g., repository source code or GitHub issues*
> **For the purpose of** *the focus of the study, e.g., identifying common bug types or understanding developer behavior*
> **With respect to** *the quality focus, e.g., functional suitability or performance*
> **From the viewpoint of** *the intended reader of the study, e.g., developers or researchers*
> **In the context of** *the context considered, e.g., the Apache ecosystem or Android apps.*

On the one hand, formulating the goal according to the template allows one to take the time to reason on the core objectives of the research and start reflecting on the research method to achieve such a goal. On the other hand,

reporting the goal by following the GQM template allows one to systematically document and swiftly communicate the research intention to those not conducting the study.

As an example of how a research goal can be formulated according to the GQM template, let us consider the study of Kamei et al. [KSA+13]. In the case of this study, a possible formulation of the research goal could be:

> **Analyze** *defect prediction models*
> **for the purpose of** *defect prediction*
> **with respect to** *risky changes*
> **from the point of view of** *software developers and reviewers*
> **in the context of** *open source and commercial projects from multiple domains.*

Note that, while introducing some systematicity, a single research goal could be correctly formulated in many ways while following the GQM template. Following Occam's razor, the most simple and informative solution should be preferred among different options. Ideally, the research goal is jointly discussed by all people partaking in the mining process. It serves as a collective moment of reflection on the goal and intent of the mining procedure.

A Do for Educators. When formulating research goals, among different options, follow Occam's razor; your students should prefer the most simple and informative solution.

2.1.2 Research Questions Definition

Once the study's goal is defined according to the GQM template (see previous section), the RQs the mining process aims to answer can be formulated. Formulating the RQs based on the research goal allows for further design, with a systematic step-by-step process, of the mining to be executed. RQs should be directly derived from the research goal formulated in the previous step. Once an answer to each RQ is provided, it is possible to assess the extent to which the research goal is achieved (and, in negative cases, further enhance the research process by adding further mining processes or RQs). If needed, during a preliminary repository mining design phase, RQs can be adapted to fit the refined research goal and envisioned repository mining process.

As noted in a recent work by Storey et al. [SRN+24] when conducting MSR studies, RQs are often formulated in a rushed manner. To avoid this, a deliberate approach to infer the most suited RQs is recommended. This involves considering the potential impact of the research, drawing inspiration from various sources, and intersecting these with possible phenomena and concepts related to those phenomena. The next step is to brainstorm RQs. This is a creative process where no idea is initially dismissed. The aim is to

generate a wide range of potential research questions that could be studied. Once a comprehensive list of potential research questions has been generated, the final step is to select specific research questions for further study. This selection should be justified based on the relevance of the question to the research objectives, the feasibility of answering the question, and the potential contribution of the findings to the field of study.

As a rule of thumb, a mining process is steered by two or three RQs. Having more than four or five RQs usually suggests that the research goal is not well-defined or that the RQs are too low-level. By taking into account the exemplary study of Kamei et al. [KSA+13], we can see how their research goal is decomposed into three different research questions, namely:

RQ_1 : *How well can we predict defect-inducing changes?*
RQ_2 : *Does prioritizing changes based on predicted risk reduce review effort?*
RQ_3 : *What are the major characteristics of defect-inducing changes?*

By comparing the RQs of Kamei et al. [KSA+13] with the research goal formulated in Section 2.1.1, we can observe that RQ_1 is the one more closely related to the goal. At the same time, RQ_2 and RQ_3 are utilized to build upon the research goal and provide further complementary notions on the considered topic. While slightly striving away from the standard application of the GQM approach, such a technique can be used to build upon and strengthen the results of the mining process set by the research goal.

As potential inspiration, examples of further RQs of an MSR study could be:

RQ : *Can Large Language Models be used to identify bugs?*
RQ : *Is there a relation between technical debt and software energy efficiency?*
RQ : *How does software quality evolve in microservice architectures?*
RQ : *Which code smells are more frequent in AI-centric software projects?*
RQ : *What are the most common causes of test flakiness?*

From a documentation point of view, once the mining results are collected and analyzed, it is considered a good practice to explicitly answer each RQ (and possible sub-RQs if present) in the mining process documentation. Explicit answers to the RQs are usually documented either in the Results section, Discussion section, or a section dedicated entirely to answering the RQs.

A Do for Educators. A mining process is usually steered by two or three RQs. Usually, having more RQs suggests that the research goal needs to be better defined or that the RQs need to be more high-level.

2.1.3 Metrics identification

As the last step of the GQM approach, each RQ is mapped to a specific set of metrics that do not overlap with those used to answer other RQs.

However, this is not always the case. For example, the number of GitHub issues could be used to answer both RQs regarding developer behavior and recurrent development impediments. The selection of metrics strictly depends on the specific RQs at hand, which can be qualitative, quantitative, or a mix of both. Selected metrics can be either atomic, e.g., lines of code or number of defects, or composite, e.g., defects per line of code. It is, therefore, possible to compose the same set of atomic metrics in different manners to answer different RQs.

A Do for Educators. When documenting the metrics used to answer the RQs, students should report their definitions and mappings to the RQs, supporting the metric definition with a sound reference or an unequivocal definition and measurement procedure description.

Teaching Exercises. Through the GQM approach, define a goal, formulate three RQs, and identify five metrics (mapped to the RQs) to investigate which development factors affect bug-proneness.

2.2 Identifying High-Quality Source Code Repositories

2.2.1 Data Sources

The MSR field collects and analyzes rich data from different types of repositories. The repositories can be of static and dynamic nature and can be classified as follows [Has08]:

- Historical repositories, including (i) bug or defect repositories, such as Jira and Bugzilla, which help track and manage defects of a software system, (ii) archived communications such as mailing lists, emails, and chat messages, which are a source of discussions regarding multiple aspects of a software system, and (iii) source control repositories that track all the changes made to software system artifacts (e.g., source code, Pull Requests (PRs), commit messages, documentation). Git is one of the most popular source control repositories.
- Source Code repositories control and manage software projects. Typical features include source code hosting, bug tracking, documentation facilities, mailing lists, and forums. Notable source code repositories include Sourceforge, GitHub, and Bitbucket.
- Runtime repositories such as deployment logs that track the information and actions related to the deployment of a software system. Deployment logs are essential to oversee the configurations and steps of deployment

instances. They usually include timestamps, error messages, and configuration settings.

For this chapter, we focused on source code repository mining to align with the running exemplar.

2.2.2 Source Code Repositories Selection Criteria

Code repositories have multiple characteristics [KGB+14] that can help in the decision-making process for selection. These characteristics can be used as inclusion and exclusion criteria for narrowing down the candidate repositories.

- Programming language(s), e.g., Java for the Apache Ecosystem projects.
- Size and complexity of the repository, e.g., number of lines of code (LOC) or number of commits
- Domain, e.g., gaming, visualization, database, parsers, testing.
- Active or inactive repository, e.g., were there recent (during the last 6 months) commit activities on the repository? When was the last commit?
- Types of repositories, i.e., base (not forked) or forked repositories.
- Purpose of the repository. Not all repositories are software repositories. Some are used for experimentation, website hosting, academic, and personal (not involving collaboration) projects.
- Location of the repository. Some projects are hosted on multiple platforms.
- Popularity of the repository, e.g., number of stars and contributors.

2.2.3 Considerations when Selecting Data Sources

Several considerations must be undertaken to mitigate potential threats when assessing source code repositories for research purposes. The repositories encompass diverse entities beyond software projects, serving as repositories for free web storage, online books, or repositories housing projects with other characteristics. Numerous repositories exhibit transient traits, being short-lived, inactive, or dedicated to assignments, student endeavors, educational objectives, personal use, or archival purposes [KGB+14]. Thus, if the research question is related to software development, these projects should be removed from the dataset. Furthermore, it is crucial to prioritize repositories that house engineered software projects [MKCN17]. Consequently, in studies related to software development, it becomes imperative to purify the dataset for such repositories. Munaiah et al. [MKCN17] propose a framework to filter the engineered software projects from GitHub. Additionally, many projects have few commits, and not all use pull requests, resulting in skewed or imbalanced data. The process of commits in pull requests and for code reviews

depends on GitHub's practice of recording the commits, which should also be studied carefully.

A strategy to mitigate such potential threats involves refraining from considering repositories that include many non-registered users as committers. Furthermore, projects explicitly identifying themselves as mirrors in their descriptions should be carefully vetted or excluded. For an MSR study, we recommend having a list of inclusion and exclusion criteria based on study goals and RQs. Manual exploration can be conducted on sample repositories to assess whether any changes to the selection criteria are required to ensure a high-quality dataset is collected.

When selecting the repositories, researchers should note that selecting and ranking the repositories based on the number of stars might favor active marketing strategies and not the well-established software engineering practices [BT18]. Therefore, checking that the repositories are not starred in a short period due to social media activity is essential. It is also important to note that the number of stars is not strongly correlated with contributors, forks, commits, and the repository's age. Thus, relying only on the number of stars can threaten the validity of the collected data.

Another consideration is the correctness of the heuristics and key terms used for selecting the repositories. These heuristics should be scrutinized and documented [Has08]. The criteria for choosing the data sources may lead to a noisy collection of repositories and, therefore, a skewed dataset. Careful investigations reveal whether this skewness is related to the inaccurate metrics for selecting repositories or is due to the nature of the data.

Understanding the limitations of repository data is another consideration, as the repository data cannot lead to causal conclusions and only can show correlations [Has08]. Moreover, the active projects might not include all their development activities in GitHub. So, other resources should be considered.

In general, MSR findings must be considered in the context in which the studies are performed, which is crucial to revealing the actual cause of particular conclusions. Indeed, such findings may not generalize across projects, and repository use could vary between projects. So, researchers should closely examine socio-technical aspects [Hod21] to better understand the use of repositories before reaching conclusions.

When presenting the results of MSR research, the limitations of repository data should be thoroughly examined and communicated. This practice is essential to prevent misinterpretations and ensure research integrity.

A Do for Educators. Students should carefully select the sources before analyzing them. When selecting data sources, the above-stated considerations allow for effective mining, which is essential to answer the RQs, prevent misinterpretations, and ensure research integrity.

Teaching Exercises. A typical exercise would be to select three GitHub projects by focusing on three inclusion and exclusion criteria each, e.g., projects written in different programming languages, of different domains, and active projects with at least 200 commits over the last year as inclusion criteria. Exclusion criteria could include academic and personal projects and projects hosted on multiple platforms. Once the projects have been shortlisted, the issue-tracking data can be mined for further analysis.

2.3 Data Extraction

Data extraction starts once a source code repository is selected and depends on the goals of the mining campaign. In studies concerning predictive analytics, this phase entails mining data concerning independent and dependent variables. The goal is to extract data concerning the former variable able to accurately predict the former variable in the future. For example, let us consider just-in-time defect prediction [KSA+13], which aims to predict defective commits given their features and involves mining commit data to extract valuable information to forecast such defective commits. Such commits can be analyzed sequentially, or specific commits can be analyzed in a given time window.

The dependent variable we will need to extract is the failure proneness of our commits. In contrast, the independent variables are the characteristics that should lead to such failure. It is essential to consider that the described concepts are generalizable to predict defects [KSA+13], code smells [APSW19], and security vulnerabilities [YR+11] concerning not only traditional code but also Infrastructure-as-Code [DPDNPT21].

In just-in-time defect prediction, the first step is identifying *defect-fixing* commits. To achieve this goal, we need to collect the issues closed and related to bugs (e.g., with labels `bug` and `bugfix`). Source code repositories like GitHub provide issue trackers that link issue reports and bug-fixing commits. A commit message is tagged as fixing defect if it matches a regular expression like the following:

$$(bug|fix|error|crash|problem|fail|defect|patch)$$

Only the commits that modify at least one source code file are kept.

The second step is to identify the *fixed files*, which are the files modified by *defect-fixing* commits, and their related *defect-inducing* commits.

The commits from the most recent to the oldest are analyzed. For each *fixed file*, the *SZZ* algorithm [ŚZZ05] automatically identifies the *oldest* commit that modified the lines of code involved in the fix. It is worth noting that SZZ has evolved over the years, and several variants are available [RPS+21].

Once the defect-inducing commit is found, all commits between the defect-inducing commit (inclusive) and the defect-fixing commit (exclusive) are labeled *failure-prone*.

After the failure-proneness of the code components has been determined, metrics able to predict such failure can be mined. In the example, Kamei et al. [KSA+13] mined 14 metrics grouped into five dimensions, i.e., diffusion, size, purpose, history, and experience. It is essential to consider that although some metrics can predict several phenomena (e.g., the file size could lead to defects and bugs), different metrics apply in different contexts. Structural metrics [CK94] focus on structural properties extracted through source code analysis. Delta metrics [dBRBvD19] capture the amount of change in a file between two successive releases. Process metrics [MPS08] consider aspects concerning the development process rather than the code itself.

In just-in-time defect prediction, all commits could be used to train the models[2]. Nevertheless, analyzing all commits could be infeasible in other contexts; therefore, selection strategies should be applied. For example, only one snapshot for each software release could be analyzed by randomly selecting it or applying another strategy. For instance, the first, last, or middle snapshot of each release could be analyzed.

Finally, several tools are available to mine source code repositories and extract valuable metrics. Among them, PyDriller [SAB18] is a Python framework that helps developers mining software repositories. PyDriller allows the extraction of information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly exports to CSV files.

A Do for Educators. There is no silver bullet. Although some metrics can predict several phenomena, different metrics apply in different contexts.

A Do for Educators. Do not re-invent the wheel. Several tools are available to mine source code repositories and extract useful metrics. Before developing a brand-new tool for mining software repositories, explore existing ones.

2.3.1 Data Cleaning

Software repositories often contain various types of noise that can skew the results of analyses [BHWS21]. Therefore, data cleaning is a critical step in preparing data, which includes a range of techniques and procedures to ensure that datasets are accurate, consistent, and reliable. Several techniques are

[2] In this scenario, the commit order is particularly relevant [FHN+20].

employed to ensure the adoption of high-quality data before analysis, a crucial practice, particularly in the context of Artificial Intelligence (AI). Different methods of handling data greatly depend on the specific context in which it will be used.

This section is designed to help readers manage data in the context of MSR, emphasizing one of the most critical aspects of software repositories: the commit. This element can hinder different issues after the data are extracted and can lead to drift in the results of our analysis. Commits are the puzzle pieces that make up a project's story. Each commit contributes to the bigger picture, but sometimes, some do not fit perfectly, leaving gaps or distorting the intended image. Addressing these problematic commits is essential to maintaining the integrity of the entire project and ensuring the accuracy of our analyses.

When mining data from commits, the first issue concerns extracting data from *tangled commits* [HZ13]. This type of commit contains multiple pieces of information related to different changes, caused often by the introduction of *multiple-fixing change commits* [NNN13]. These changes during the development phase may not cause major issues. However, they can create problems while analyzing the corresponding version archive by introducing inaccuracies. For example, if a complex change is made to fix a bug, all the files associated with it may be mistakenly labeled as defective in the historical context. When treating these types of commits, it is suggested to understand the changes inside a commit and consider it separately as *change set partitions* [HJZ16].

An additional critical issue that arises during the data collection process is the inclusion of *merge commits*, which has been highlighted as a nuanced issue by Kovalenko et al. [KPB18]. This issue requires an additional effort since many merge operations may not be labeled as such, leading to inspecting the content of the changes manually to identify it [KGB+14]. Similar to *merge commits*, it is necessary to consider other types of changes and commits that can mislead the analysis. In this set, we can also find *quickly remedy commits* (i.e., commits aimed at implementing changes omitted in the previous commit) [WNLB20]. The decision to include or exclude these commits can significantly impact the complexity of the dataset and the richness of the information obtained. Wen et al. discovered that excluding this type of commit allows us to avoid introducing a significant amount of noise in MSR studies [WNLB22]. Incorporating all commits, including those outside the main branches, can elevate the complexity of the analysis. However, it can also enrich the dataset by adding important information to the software history and providing a more comprehensive view of collaborative efforts, branching strategies, and concurrent development threads.

On the other hand, limiting the analysis to the main branch can simplify the extraction of essential information. However, it may also lead to missing valuable insights into the other branches. Therefore, the inclusion or exclusion of these commits presents a delicate balance, and MSR researchers should

carefully consider the trade-offs involved. As such, it is crucial to weigh the benefits and drawbacks of each approach, keeping in mind the research goals and the nature of the data under investigation.

When determining whether to include or exclude commits based on the goals of the MSR study, it becomes evident that commits containing information not aligned with the study's objectives are considered noise. *Noisy commits* in software development and version control systems refer to changesets or commits deemed extraneous to the primary focus of analysis. These typically involve minor modifications like fixing typos, adjusting task names, or resolving lint warnings [LXH+18]. In MSR studies, researchers often encounter noisy commits and filter them out to improve the signal-to-noise ratio, allowing for a more targeted analysis of substantive changes. Dalla Palma et al. [DPDNPT21], for instance, exemplified this approach by excluding commits that addressed typos, task names, and lint warnings, prioritizing more impactful contributions to the codebase. Additionally, an MSR researcher should consider including or excluding bot commits [DVM20]. Detecting and considering the exclusion of this type of commit can significantly differ on several aspects involved in MSR analysis, including community-related aspects [DMP+20]. This commit type is detectable by observing the commit message and the list of changes the bot will perform.

To conclude, an MSR researcher must carefully navigate the decisions surrounding commit inclusion or exclusion, acknowledging the significance of commits in shaping the quality of data analysis. By aligning these decisions with the specific goals of the study, researchers ensure that the dataset remains focused on substantive contributions, enhancing the precision and relevance of their findings. Therefore, an MSR researcher should take the following steps to obtain the optimal dataset:

- Identifying the criteria for selecting the key commits that should be included in the study.
- Consider including or excluding tangled, merge, and quick-remedy commits.
- Based on the defined criteria, identify and exclude the noisy commits in the extracted data.

A Do for Educators. Students should carefully clean data. Each piece of data is the puzzle piece that makes up the story of an MSR campaign. They all contribute to the bigger picture, but sometimes, some do not fit perfectly, leaving gaps or distorting the intended image.

Teaching Exercises. A practical exercise for understanding the data extraction phase involves students applying algorithms from the study of Kamei et al. [KSA+13] to identify bug-inducing and fixing commits in repositories. Starting with chosen repositories, students identify commits that fix defects. During this process, they meticulously clean the data by removing any tangled and merge commits. Finally, students extract the commit message and the set of files modified by each defect-fixing commit.

2.4 Code Analysis

Source code analysis is extracting information about a program from its source code or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools [Bin07]. Source code analysis can be textual, static, dynamic, and historical [DRGP13]. We focus on static code analysis.

2.4.1 Static Code Analysis

Source code is a crucial artifact that can be analyzed to reveal information. A technique to analyze source code is static code analysis (also known as structural analysis) - the process of checking the source code of a program for issues without executing the program. Static code analysis is helpful in (i) ensuring that code adheres to rules about good coding practices, (ii) finding defects and security issues and (iii) identifying code smells and technical debt instances. Static analysis can be applied during the early stages of software implementation for early fault detection since the code does not need to be fully functional or executable.

Source code can be analyzed at different levels of granularity. The analysis level depends on the study's goal and research questions, which will, in turn, dictate the level of granularity at which the analysis needs to be performed. Generally, analysis can be performed at (i) method level - the lowest level of granularity which allows finer-grained analysis, (ii) class level - a coarser level of granularity compared to method level but still popular among researchers, (iii) file level - a higher level of granularity and lower level of details when analyzing groups of classes and (iv) system level - the highest granularity level and the lowest details level.

Tools such as **Automated Static Analysis Tools (ASATs)** are increasingly used for static analysis. ASATs can be of general purpose (e.g., PMD, SonarQube, Understand, Cppcheck), bug-focused (e.g., SpotBugs, Coverity),

and security-focused (e.g., Flawfinder, Fortify). Static analyzers differ based on the level of sensitivity regarding precision and analysis time [EN08]. A flow-sensitive analysis is more precise but more time-consuming and considers the order of the statements compared to a flow-insensitive analysis. A path-sensitive analysis considers only valid paths and can be precise but costly, while a path-insensitive analysis considers all execution paths, including infeasible ones. Context-sensitive (also known as inter-procedural) analysis considers the context during the analysis compared to an intra-procedural analysis. The latter is faster but more imprecise compared to an inter-procedural analysis.

The Open Worldwide Application Security Project (OWASP)[3] provides a comprehensive list of security static analyzers. Some commonly used static analyzers are presented next.

PMD[4] is a general-purpose static analyzer that supports 16 languages but primarily focuses on Java. For instance, Java rules enforce accepted best practices and coding styles, uncover design issues, detect constructs that are either broken, extremely confusing, or prone to runtime errors, flag code documentation issues, suboptimal code, potential security flaws, and issues when dealing with multiple threads of execution.

SpotBugs[5] (formerly FindBugs) is a static analyzer mostly focused on defect detection based on a predefined set of more than 400 bug patterns in Java code. These bug patterns check for bad practices, correctness, performance, malicious code, and security issues.

Fortify[6] is one of the most popular static analyzers specifically for violations of security-specific coding rules and guidelines in multiple languages. Fortify comprises eight vulnerability analyzers: buffer, configuration, content, control flow, dataflow, null pointer, semantic, and structural.

2.4.2 Dynamic Program Analysis

Dynamic program analysis involves analyzing the properties of a program while it is executing with real input data. Unlike static analysis, dynamic analysis aims to identify issues while the code runs. Dynamic analysis is useful in identifying (i) a lack of code coverage, (ii) memory allocation and leaks, (iii) performance bottlenecks, (iv) software vulnerabilities and defects, and

[3] https://owasp.org/www-community/Source_Code_Analysis_Tools

[4] https://pmd.github.io/

[5] https://spotbugs.github.io/

[6] https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer

(v) concurrency issues such as deadlocks and race conditions.

A comprehensive list of dynamic analysis tools can be obtained from the `analysis-tools.dev` website[7]. Some notable **dynamic analysis tools** include:

Valgrind[8] provides a suite of tools for building dynamic analysis tools. For instance, Memcheck can detect many memory-related errors that are common in C and C++ programs and can lead to crashes and unpredictable behavior, Helgrind is a thread debugger that finds data races in multithreaded programs, and Massif performs detailed heap profiling by detecting which parts of the program are responsible for the most memory allocation.

Application Verifier (AppVerifier)[9] by Microsoft is a runtime verification tool for unmanaged code to detect and help debug memory corruptions, critical security vulnerabilities, and limited user account privilege issues that would be difficult to detect during regular application testing.

Code Pulse[10] is a real-time code coverage tool for penetration testing activities by OWASP and Code Dx (acquired by Synopsys[11]) and automatically detects coverage information while the tests are being conducted. Code Pulse currently supports Java and .NET Framework programs.

Teaching Exercises. A useful exercise for understanding the utility of performing static or dynamic analysis on code is to analyze the impact on code quality before and after a defect-fixing commit. Students will retrieve versions of the code from a specific repository both before and after a defect-fixing commit. By utilizing a chosen analysis tool, such as SpotBugs, students will explore the implications of defect fixes regarding correctness, performance, and security issues.

2.5 Mined Data Analysis

The analysis of the collected data depends mainly on the data type, but also on other factors such as the purpose of the analysis and the tools and techniques available. Even when source code repositories are analyzed, there are

[7] https://github.com/analysis-tools-dev/dynamic-analysis

[8] https://valgrind.org/

[9] https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/application-verifier

[10] https://code-pulse.com/

[11] https://www.synopsys.com/software-integrity.html

multiple data sources and types: source code, commits, GitHub issues, questions and answers, review comments, code comments, numerical data, etc. In this section, we will review some data types and techniques for analyzing the mined data. As we have focused on code in the previous section, we cover a broader range of artifacts in this section. Although source code analysis requires specific techniques, which are beyond the scope of the current chapter, the topic modeling and deep learning-based approaches mentioned in this chapter are widely used for analyzing the source code.

2.5.1 Unstructured Data

"Unstructured data does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records." [CTH16]

Textual data such as commits or source code are unstructured, as they do not have a pre-defined format. Instead, the authors can write the text in any order and with different contexts, even though a template is used. Understanding the content of the text is of main interest in many studies and requires analysis. We can use several techniques to analyze textual data. The main factor of using the analysis technique is 'what' we intend to identify in our study. This can vary from more straightforward techniques, such as analyzing the frequency of words based on Term-Frequency Inverse-Document-Frequency (TF-IDF) metrics, to more in-depth analysis to understand the content of the documents or categorize them, such as topic modeling, clustering, and classification techniques to sophisticated analysis such as finding the reasons, relations, or causality among concepts, using more advanced text analysis techniques such as causal inference. In all cases, the textual data could be represented with numerical vectors (also referred to as vector embedding) that can be learned using deep neural models.

Topic Modeling

Topic modeling or latent topic modeling is a category of techniques for automatically extracting topics from a corpus of text documents. The corpus of documents can be short or long texts, such as commit messages, app review feedback, or bug reports. Please note that each commit, app review, or bug report is considered a document in this example. A topic is a collection of terms that frequently occur in the documents. When topic modeling is applied, the main topics of the documents' corpus are extracted; therefore, we can compute the topics of each document as well. Thus, topic modeling uncovers the latent semantic relationships of the documents. As this ensures a faster analysis of many documents, topic modeling is used for several soft-

ware engineering applications such as bug triaging, finding the most discussed topics or issues, traceability link recovery, and concept location. Multiple applications of topic modeling in software engineering research are discussed in the literature [SGG21].

Examples of topic modeling algorithms are Latent Semantic Indexing (LSI), Probabilistic Latent Semantic Indexing (PLSI), Latent Dirichlet Allocation (LDA) and its variations, such as Hierarchical Topic Models (HLDA), non-negative matrix factorization (NMF), and Biterm Topic Model (BTM). While LDA and other algorithms are designed for longer text, BTM is a newer algorithm for short text. The BTM algorithm considers the Biterms in the whole corpus to enhance the topic learning in short texts. Online algorithms for BTM, i.e., online BTM (oBTM) and incremental BTM (iBTM), are also introduced to speed up the inference of BTM on large data sets.

The algorithms mentioned above are mainly used to extract the topics from documents at a static point in time. As the distribution of the terms in a document changes over time, so can its topical topic modeling techniques, which are developed to detect the evolution or variations of topics in time-stamped documents. Dynamic Topic Model is one of these algorithms. Online Latent Dirichlet Allocation (OLDA) is another method that tracks the variations of topics over text streams. The OLDA models the texts' topics of one time slice based on the topics of the last time slice. Newer algorithms are developed that consider previous time slices of the documents for better topic modeling. Adaptively Online Latent Dirichlet Allocation (AOLDA) is an algorithm specifically developed for software engineering and app review analysis. It improves OLDA by adaptively combining the topic distributions of previous versions to extract topics in the current time slice. Adaptive Online Biterm Topic Modeling (AOBTM) is a similar algorithm developed for short texts. It analyzes the statistical data of previous time slices to identify the topic distribution of the current time slice.

The more recent topic modeling algorithms are based on word embeddings or neural network-based topic models, such as Top2Vec, LDA2Vec, and BERTopic.

Considerations. When applying topic modeling, several considerations are necessary to improve the results:

- Text pre-processing: Though the text pre-processing varies based on the study context and data, the common techniques are to remove punctuation and non-textual symbols, check the language of the text (e.g., English only), spell check and correct the spellings of the words, stemming and lemmatization, and removing stop words.
- Investigating the length of the text: It is shown that the topic modeling algorithms do not work correctly for short text. So, investigating the length of the documents is necessary in selecting the topic modeling algorithms, whether they are designed for short or long text.
- The evaluation metrics: The evaluation metrics of topic models span over quality, interpretability, stability, diversity, efficiency, and flexibility. Two

well-known terms to measure the topic models are perplexity and coherence. The former refers to how well the model explains the data (predictive power of the model), and the latter identifies a measure of whether the generated words for a topic can be associated with a single semantic meaning. One way to evaluate coherence is Pointwise Mutual Information (PMI). For other evaluation metrics, the work of Abdelrazek et al. [AEG+23] provides a good starting point.

- Number of topics: The user should choose the number of topics in most algorithms. Rather than using a random number or based on the heuristics, a common way is to explore a range of different topic numbers and evaluate the coherence score of the results, which could be a good determination to select the number of topics. There are other metrics for models such as BERTopic that should be evaluated for selecting the best number of topics.
- Hyperparameter setting: Similar to the previous point, setting the hyperparameters is important in running the algorithms. So, care should be taken to experiment with different choices before applying the topic modeling.
- Running time and size of data: One main consideration in choosing the topic modeling algorithm is the time it takes to execute the algorithm and whether it can be applied to a large dataset.

Sentiment Analysis.

Sentiment analysis, also referred to as opinion mining, refers to gathering and analyzing people's opinions, emotions, or attitudes towards an entity (i.e., individuals, product, topic, etc.) [WRK22]. In software engineering, sentiment analysis can analyze various MSR artifacts, such as app reviews, users' product feedback, and developers' discussions. Sentiment analysis determines whether the opinion is positive, negative, or neutral. The analysis can be done at different levels of granularity: document, sentence, phrase, or even certain aspects of an entity. Several algorithms are developed for sentiment analysis, from lexicon-based approaches to unsupervised and supervised machine learning techniques, including deep learning models and pre-trained language models, hybrid approaches, and transfer learning approaches.

Considerations. While for some text analysis applications, the text should be cleaned and punctuation marks removed, the text pre-processing considerations for sentiment analysis should be done carefully. Several essential features can reveal sentiments, including punctuation marks, emojis, and slang words, which are some selected features for sentiment analysis.

(Deep) Neural Network Based.

Another common technique for analyzing textual data is representing the document with a vector. This vector, called embedding, can be a non-contextual or contextual text representation. For non-contextual representations, the model learns a fixed vector for each word, no matter their context.

Famous examples of such embeddings are word2vec, glove, and code2vec. In the contextual representation, the word embedding depends on its surrounding words. Contextual representation has recently been used in many applications in which the embedding of the words or text is extracted from a language model, a model that learns a probabilistic model of a natural language from a large corpus of textual data. Multiple models can be used to extract the embeddings of text or code, including Sentence Transformers or GPT family for text or code-specific language models such as CodeBERT, GraphCodeBERT, CodeT5, or Code Llama. Many newer models can also be used, and we refrain from naming them due to the increasingly fast introduction of newer models. While topic modeling and sentiment analysis are techniques aiming at a specific goal (i.e., understanding what is being discussed and what emotion is expressed), deep learning models are generally discussed and can be used for various purposes, including topic modeling and sentiment analysis. These models or embeddings extracted from them can be applied in other analyses such as predictive analysis or training machine learning models. Some examples of the studies that use deep learning for mining software repositories are mentioned in the survey by Yang et al. [YXLG22].

Considerations. In the following, we list a few points to consider when choosing the models.

- Cost of using a model. Most of the recent language models are based on deep learning approaches. The computational costs available to the researchers should be considered when using them. Additionally, if using models such as a GPT-based family, one may intend to pay for the calls to the model and receive the embeddings. Therefore, the budget associated with the research based on the dataset size is an important factor.
- Model selection. Several non-contextual and contextual models are specifically developed for software engineering tasks and code. Depending on the application, we recommend a search to identify the model to use in the analysis. Several SE-specific language models have been developed, such as seBERT, which is a BERT model trained from scratch on software engineering data. These specific models are available for various applications and domains in software engineering. It is necessary to conduct at least some initial studies if there is insufficient literature to ensure the final model is chosen properly.

2.5.2 Structured Data

Structured data is data that has a standard format. Examples of structured data are time stamps, number of commits, and years of experience. The structured data are analyzed through various means, such as correlation analysis, time series analysis, statistical analysis, clustering, regression analysis, and developing prediction models. Note that some techniques, such as clustering, and developing machine and deep learning models might apply to the unstructured data. A discussion about machine learning strategies for MSR can be found in [GPLNMSMR18]. Both structured and unstructured data can be used for the analysis or extraction of quantitative metrics while working with unstructured data. For example, the mean number of accepted answers on Stack Overflow can be calculated while also analyzing the contents of the questions and answers.

In the following, we briefly describe some of the statistical tests. Statistical methods, or models, are powerful tools for analyzing data and supporting arguments. They are mathematical formulas used to analyze numerical raw data. The study conducted by De Oliveira Neto et al. [dONTF+19] provides a survey on different tests that are used for empirical studies in software engineering, which can be used as a starting point to review the widely used statistics tests. Parametric and non-parametric tests are specific classes that refer to the distribution of the population's parameters. The t-test, ANOVA, and F-test are some common parametric tests. Examples of non-parametric tests are Mann–Whitney U test and its variations, Kruskal–Wallis test, and the χ^2 test. If multiple tests are conducted, correction tests such as Bonferroni or its variations should be considered.

Another method is statistical power analysis is a test conducted to accept or reject a hypothesis. The statistical power refers to the probability of a hypothesis finding an effect if there is an effect. Power analysis uses a significance level, effect size, and statistical power to estimate the required minimum sample size.

Considerations. In the following, we list a few points to consider when analyzing structured data.

- Data distribution. When applying statistical tests, it is imperative to assess data distribution. Statistical tests are often developed for normal or non-Gaussian distributions, and one cannot be applied to the other. Therefore, the results are unreliable if the data follows a non-Gaussian distribution, but the test is for normal distributions. Shapiro–Wilk and Kolmogorov–Smirnov tests are commonly used to assess the normality distribution of datasets.
- Reporting effect size. When applying statistical tests, it is necessary to understand and reflect on the effect size and significance level and not choose them arbitrarily. When reporting the results of a statistical significance test, usually p-values are reported but not the effect size. However, a p-value only informs the existence of an effect but not its magnitude

(i.e., effect size). Therefore, the effect size should be reported to find out the size of a significance [SF21].
- Often, to develop an analysis, the model's features combine structured and unstructured data. In this case, the embeddings of the text or code could be used as features, or the text itself could be considered. There are several approaches to be used, and one can experiment with different combinations to evaluate the results.

A Do for Educators. Data analysis depends on the data types. Even for source code repositories, there are multiple data sources and types. Tools must be selected considering the nature of the repositories and the analysis to conduct.

Teaching Exercises. A useful exercise for understanding data analysis in MSR involves performing topic modeling on defect-fixing and non-defect-inducing commits. Students will start with a set of commits from Kamei et al. [KSA+13] and preprocess the commit messages to ensure data quality. Students will identify the main topics discussed in defect-fixing and non-defect commits using a specified topic modeling technique, such as Latent Dirichlet Allocation (LDA) or BERTopic. They will analyze and interpret these topics, comparing the topics of defective and non-defective commits and exploring their relevance to different types of defects and their implications for software maintenance practices. A similar practice can be applied using sentiment analysis to investigate whether the sentiments of commits change when fixing a defect compared to a non-defect one.

2.6 Data Synthesis

Collecting and synthesizing the analysis results is crucial to interpreting data and discovering important findings. Before presenting all the results extracted from data analysis, an MSR researcher's primary focus is to remember the goal addressed in the previous steps. Following the GQM approach (presented in Section 2.1), all the plots and the results shown in the study aim to answer the research questions formulated. Therefore, before starting, it is crucial to keep the following points in mind:

- **Research Objectives as a Guide:** Using the formulated research goals helps to organize and prioritize your results. Therefore, considering a mapping link between the research question and the findings helps to find the best solutions to present results through discussion, plots, and tables.

- **Relevance to Research Questions:** Articulating how each result addresses specific research questions helps to check if the goal is accomplished. This approach ensures that the presentation of the results maintains a cohesive narrative and directly ties back to the study's purpose.
- **Embracing Unexpected Findings:** During an MSR study, it is possible that the results show something different from what is expected. The presentation and discussion of unexpected results could enlighten new implications and open the possibility of novel opportunities.

Once the goal is clearly assessed, it is necessary to format the data and the outcome of the analysis to obtain the information needed to answer our research questions. Therefore, summarizing and exploring the data is important to extract the veiled message from large collected data.

The main approaches to summarize the analysis results are based on descriptive metrics and visual representations.

2.6.1 Descriptive Metrics

Using properly descriptive metrics and presenting information is integral to conveying the study results. In summarizing metrics derived from MSR, descriptive statistics are pivotal in distilling complex datasets into meaningful and interpretable insights. The choice of descriptive statistics should be adopted, focusing on the goals of the analysis to find the most suitable answers from the data. Key descriptive statistics capture the central tendency, variability, and distribution of the metrics, including mean, median, mode, and standard deviation [KMB+17]. For instance, the mean serves as a central measure, offering an average value that summarizes the overall trend within a dataset. However, it is important to note that the mean can be influenced by extreme values, making it less representative of skewed distributions. In the context of software components, calculating the mean number of lines of code provides an overview of the project's codebase size, but it may not fully capture the central tendency if the distribution is highly skewed. The median, resistant to extreme values, provides a robust representation of the center of the data. In MSR, identifying the median response time for resolving issues can offer a more stable representation of the typical resolution time, even in the presence of outliers. Mode highlights the most frequently occurring values, emphasizing prevalent patterns. For instance, identifying the mode in a dataset of commit frequencies may highlight the most common development activity intervals. Standard deviation quantifies the dispersion of data points, offering insights into the dataset's variability. For example, examining the standard deviation of code churn rates can reveal the extent to which development activity is variable across different periods. Collectively, the combination of several descriptive statistics contributes to a comprehensive understanding of the metrics, aiding researchers and stakeholders in

uncovering patterns, trends, and central characteristics within the intricate landscape of software development data.

2.6.2 Visual Representations

Descriptive statistics offer a concise and straightforward way to present data analysis outcomes. However, relying solely on numerical summaries, such as means and medians, can obscure significant patterns and variations within data. This approach may also fail to capture the complexity of relationships or variations in multidimensional data. To overcome these shortcomings and improve the interpretability of the results, it is crucial to supplement numerical summaries with visual representations. Visualizations provide an interactive and intuitive method for discovering patterns, outliers, and correlations within the data. Employing various visual elements such as plots, tables, and representative graphs when synthesizing your data will increase the comprehension of the readers interested in the study and allow them to catch it.

As for the descriptive metrics, each plot type serves a unique purpose in conveying information effectively. Some of the most used visual representations to summarize the results in MSR studies are:

- **Line Charts**: This type of plot is helpful to visualize temporal aspects of software repositories, showcasing the evolution or the history of metrics over time. An example of this type of plot for MSR Studies can be related to analyzing the use of third-party libraries over time [SPDN+18].
- **Bar plots**: These plots show the frequency of a categorical variable using bars. The bars' height indicates the data's value in each category, such as the frequency, total count, sum, or average. An example of using bar plots for MSR studies is to analyze the number of occurrences of bug categories in software systems [CPZF19].
- **Box Plots**: These plots show the distribution of a numerical variable using five statistics: the minimum, the lower quartile, the median, the upper quartile, and the maximum. Box plots can be used to compare the central tendency and the variability of different groups or samples of data. An example of using box plots for MSR studies can be related to the extraction of the number of commits that lead to the appearance of a code smell [TPB+15].
- **Scatter Plots**: This type of plot is used to show the relationship between two numerical variables. The position of each dot on the horizontal and vertical axis indicates the values of the variables for each observation or unit of analysis. Scatter plots help explore the correlation or association between two variables or identify outliers or clusters in the data. Scatter plots are commonly used to analyze a specific factor's evolution to find tendencies. An example of using scatter plots is analyzing the adoption of reusability mechanisms in source code over time to find common patterns between projects [GFC+24].

- **Network Graphs**: This type of plot is commonly used to model relationships between elements. It can be used to visualize clusters and patterns in the data and analyze complex systems' properties and behaviors. Network graphs in MSR studies are commonly used to explore collaborative networks between developers [GSP14].

2.6.3 Best Practices and Dos for Educators

The following best practices are collected to inform MSR researchers of the main requirements for reporting an MSR study.

Replicability

In MSR studies, it is crucial to document data sources, collection methods, preprocessing steps, and analysis techniques for easy replication by other researchers. MSR researchers should always provide all the material to reproduce the conducted studies, including code, scripts, or workflows. All this content should be in an accessible collection called *replication package*. Mahmood et al. [MBH+18] proposed using common online replication services, such as OpenML[12] and Zenodo[13], to replicate MSR studies effectively.

Use of Visual Elements

When incorporating visual elements in information representation, prioritize clarity, consistency, accessibility, and relevance. Visualizations should be designed with simplicity to ensure easy comprehension for a diverse audience. Consistent use of visual elements fosters a cohesive visual language, while accessibility features, such as color-blind-friendly palettes and alternative representations, promote inclusivity. Additionally, colors can be strategically used to represent additional information, adding depth to the visualization. Visualizations should serve a clear purpose and contribute meaningfully to the overall narrative, enhancing the interpretability of complex data.

Use of Relative Discussions to Findings

When presenting research findings, it is important to emphasize their significance and relevance to the research questions. Researchers should discuss existing literature, industry benchmarks, or expectations to comprehensively

[12] http://www.openml.org/
[13] https://zenodo.org/

understand the relative importance of the findings. By highlighting the connections between the findings and the external contexts discussed, researchers can offer richer insights and demonstrate the practical implications of their work. This approach ensures that the reported results are not isolated but instead contribute meaningfully to the ongoing goal of the study.

Use of a Systematic Quality Assessment Process

When all research components are in place, employing a systematic quality assessment process is prudent. This systematic approach ensures a thorough and rigorous evaluation of all artifacts generated throughout the study. The assessment process should encompass a comprehensive review of the dataset, code, visualizations, and other materials, verifying their accuracy, completeness, and adherence to established standards. MSR researchers can identify and rectify potential errors or inconsistencies before finalizing the study's outcomes. To have an effective systematic evaluation process of the MSR study, Chatterjee et al. [CSR22] proposed a series of standards that an MSR researcher should use to ensure high-quality artifacts.

Teaching Exercises. A good exercise is reproducing an existing study, including related synthesis, analysis, and interpretation of the results. Considering the outcome of the previous exercise that highlights the most used topics in defect-fixing commits, students will attempt to synthesize the data, collecting key elements and interesting findings. They will create plots to observe critical characteristics of the synthesized data and make interpretations of the results to explore the topics in defect-fixing commits.

2.7 Threats to Validity in MSR Studies

While often considered a mere afterthought of mining processes [VEL+23], Threats to Validity (TTV) play an essential role in empirical inquiries. Summarily, threats to validity could potentially affect the accuracy or credibility of a study or its results [WRH+12]. A transparent, comprehensive, and truthful documentation of the TTVs that may have influenced a study is essential. TTV considerations are crucial to let the reader accurately understand the mining process, interpret its results, and potentially build upon them. TTVs and related mitigation strategies should be considered throughout the entirety of the mining processes, starting from the earliest stages (e.g., by reflecting if the RQ is correctly formulated to achieve the research goal) till the concluding steps. Among different TTV categorizations, often MSR stud-

ies follow the four categories presented by Wohlin et al. [WRH+12], namely *conclusion validity*, *internal validity*, *construct validity*, and *external validity*.

2.7.1 Conclusion Validity

Threats to the conclusion validity refer to impediments that may affect the ability to draw the correct conclusion about relations in the collected data. Leaving aside conclusion threats related to statistical result analyses (e.g., low statistical power and violated statistical test assumptions), recurrent conclusion threats in MSR processes regard confounding factors, such as measure reliability (mainly if dynamic code analyses are used), random irrelevancies in the mining process (e.g., including extended periods of repository inactivity), and excessive heterogeneity of repositories (e.g., considering in the same MSR process both cloud-native and embedded contexts). Mitigation strategies for conclusion TTVs in MSR studies often entail the selection of repositories based on *a priori* defined criteria, the carefully motivated use of sound source code analysis tools, and *post hoc* scrutiny of results trends to spot potential conclusion pitfalls related to specific repositories.

2.7.2 Internal Validity

Threats to the internal validity regard unknown factors that can impact the study's relationship between the phenomenon considered and the observed results. Threats of this nature are often related in MSR processes with a summary repository selection, inaccurate mined data post-processing, unfitted statistical data analyses, and, when dynamic analyses are used, the influence of previous code executions on subsequent ones. Internal TTVs of MSR studies can often be tackled during the mining design phase by ensuring that a systematic, repeatable, and documented process is used to select repositories and manage the collected data, and "cooldown" and cache cleaning precautions are taken to start each dynamic measurement as a clean slate.

2.7.3 Construct Validity

Construct validity pertains to the representativeness of the designed mining process to accurately study the considered theoretical construct. Most commonly, construct threats in MSR processes relate to the design phase of the mining processes and may involve a vague or ill-suited definition of the construct (e.g., defining technical debt as maintainability issues), underrepresentation of the considered construct (e.g., considering a single type of test flakiness to study the topic), or the adoption of an overly-narrow analysis to examine a considerably broader construct (e.g., focusing solely

on number of functions to study software testability). During the design of MSR processes, the construct under study and how the mining considers such construct should be carefully discussed among miners to understand, mitigate, and document the potential construct TTVs entailed by the adopted methodology.

2.7.4 External Validity

External validity regards the extent to which the results obtained with the mining process can be translated into other contexts (e.g., industrial practice, another application domain, or a different programming language).

Among the external TTVs, one of the most common ones in MSR regards the under-representation of the entire set of software repositories and commits relevant to answering the RQs. A strategy to systematically sample repositories and commits is paramount to mitigate this threat by considering the threat such sampling entails. A sound sampling should carefully select both a fitting sampling technique (e.g., stratified sampling if repositories of different natures need to be represented) and a sample size. Regarding sample size, numerous tools can be used to calculate it based on confidence interval and margin of error, typically 95% and 5%, respectively, but these can vary according to the entire set of repositories considered.

Another common external TTV of MSR studies is adopting *ad hoc*, manual, or outdated processes to mine and analyze the source code. Conscious attention should be paid to ensure that systematic and repeatable state-of-the-art and practice processes are used, i.e., analysis tools and considered repositories, to mitigate this threat.

Additionally, depending on the data under study, other sampling techniques, such as stratified sampling, should be considered to ensure that the distribution of the sample data in each class or group is retained and similar to the distribution of the classes in the original data.

3 Complementing Software Repository Mining Studies

MSR studies might be combined by analyzing additional sources of information. From a methodological standpoint, this is the basis of *mixed-method* research [Cre99, SHMB24], a research approach combining qualitative and quantitative research methods elements within a single study or research program. This approach seeks to harness the strengths of both methodologies to provide a more comprehensive and nuanced understanding of a research question or phenomenon. Storey et al. [SHMB24] recently defined guidelines aiming to ease the application of mixed-method research in software engineering, as well as a catalog of best and bad practices to help apply it. In

the scope of MSR research, mixed-method research may be useful to enhance or confirm the findings coming from mining human-generated data but also enable asking different questions as they arise during the study [SHMB24]. In doing so, researchers may triangulate the findings by exploiting multiple research methods, increase the overall credibility of the findings, or even find contradictory or surprising results [SHMB24].

In the following section, we overview the potential methodologies that may be used to complement the results of MSR studies and provide exemplary articles that may be used to illustrate these methodologies in practice.

Complementing MSR Research with Qualitative Methods. For instance, imagine complementing an MSR exploration with qualitative research methods, such as *surveys*, *interview studies*, and *focus groups*. A survey represents a research method that involves collecting data from a sample of individuals by administering a structured set of questions. When combined with MSR studies, a survey could capture the subjective experiences and perceptions of developers, project managers, and other stakeholders, providing a human narrative to complement the quantitative trends identified in the MSR exploration. Literature has often relied on this combination of research methods. A notable example is the paper by Qiu et al. [QNB+19], where the authors effectively combined software repository analyses and insights from a survey study to investigate the impact of social capital on the sustainability of open-source projects.

Interviews represent an alternative to survey studies. An interview is a qualitative research method that systematically collects and analyzes data from one-on-one interviews with participants. Unlike survey studies, interviews provide researchers with finer-grained insights from the interviewees' experiences. By nature, interview studies can only reach a small sample size and are typically limited to the analysis of a few practitioners. As a consequence, interview studies are particularly suitable when the aim is to gain a deep and nuanced understanding of participants' experiences, perspectives, or beliefs. For example, interview studies with key stakeholders offer a deeper understanding of individual experiences, motivations, and decision-making processes, adding a layer of context to the automated data. On the contrary, if the goal is generalizability, there might be better research instruments than an interview study. The interested reader might take the paper by Tao et al. [TDX+12] as a valuable example of how to make interviews instrumental for the goals of an MSR exploration.

Focus groups represent an additional alternative. These refer to a qualitative research method that involves a small, diverse group of participants who engage in an open and facilitated discussion about a specific topic under the guidance of a moderator. This method aims to gather insights, perceptions, opinions, and attitudes through group interaction, allowing participants to express their views and respond to each other in a dynamic setting. Like the interview studies, focus groups rarely have the power to generalize the insights that emerge and should be used to understand better the quantitative

findings obtained by an MSR study. An example of applying this combination can be found in the paper by Falessi et al. [FJW+18].

Complementing MSR Research with Quantitative Methods. Besides qualitative research methods, it is also worth reporting that an MSR study can be empowered using additional mining instruments beyond the scope of the traditional version control systems. For instance, *issue trackers*, where bugs and tasks are recorded and discussed, become a valuable trove of information. Mining these repositories unveils the challenges developers face, the evolution of bug resolution processes, and the collaborative dynamics surrounding issue resolution. At the same time, *code review repositories*, theaters where code changes are scrutinized, are an opportunity to explore the quality assurance practices within a project. By analyzing discussions, comments, and decisions made during code reviews, researchers gain insights into coding standards, knowledge transfer, and the social dimensions of code evaluation. Finally, *developer forums* like Stack Overflow become digital arenas where practitioners seek and provide solutions. Mining these forums provides a glimpse into the knowledge-sharing ecosystem, exposing common challenges developers face and the collaborative solutions the community offers. A notable example of the combination of multiple mining instruments can be found in the work by Ram et al. [RSCB18].

Key Advantages of Mixed-Method Research. In the scope of MSR, integrating qualitative research methods and exploring additional mining instruments offer a nuanced perspective. While MSR studies provide the backbone of empirical evidence, qualitative methods infuse a *human dimension*, unraveling the stories behind the code changes. Delving into issue trackers, code review repositories, and developer forums adds layers of context, portraying software development as *a collaborative, evolving journey rather than a mere compilation of code changes*. In essence, this integration transcends the boundaries of quantitative and qualitative methodologies, fostering a research approach that mirrors the complexity of the software development ecosystem. It is an opportunity to see automated analyses and human narratives converging. It offers researchers a holistic understanding of the intricate connection between code, collaboration, and the people who bring software projects to life. In Appendix 6, we report a non-exhaustive list of repositories that the reader may find helpful to running mixed-method MSR research.

A Do for Educators. MSR studies can be combined by leveraging other methods to obtain mixed-method research.

4 Ethical Mining

MSR research typically involves analyzing human-generated data, including developers' activities and interactions in repositories, as well as cultural and geographic data, such as racial and ethnic origin, which are necessary for studies on the geo-cultural dispersion of software communities. Although it is classified as a data-driven strategy rather than a respondent-driven one [SEWK20], ethical considerations may still be relevant. For example, consider the case of a mining study aiming to investigate PR acceptance rates in open-source repositories. The study may require ranking contributors based on their PR acceptance rate: while GitHub data may indeed be used to rank contributors and identify the most/least successful contributors, publishing the identity of those contributors would be unethical.

Despite ethics plays a role in MSR research, it is often overlooked. From 100+ papers on MSR mining challenges and data showcases from 2006 to 2021, only a few discussed ethics or data anonymization as part of the threats to validity [GK22], hence suggesting the need for educating the next generation of researchers to consider ethical aspects while mining software repository data. This is especially true when considering the potential impact of MSR research in practice: according to Feitelson [Fei23], open-source developers are indeed largely open to research, provided it is done transparently.

Considering source code repositories as an example, it is essential to note that publishing source code under a license differs from publicly releasing a repository. Repository data are often not explicitly licensed for study and unrestricted use. Therefore, ethical concerns might arise, and ethical issues should be considered in such situations [GK22]. More particularly, when mining the human-generated data coming from repositories, some recommendations based on the Menlo report include [GK22]:

- *Stakeholder identification* - Consultation of all parties involved and impacted by the research (e.g., ICT researchers, human subjects, non-subjects, users, and platform owners) before using their data for research purposes. Adhering to this guideline when performing large-scale mining analyses may be challenging. Nonetheless, it is worth remarking that, according to the GitHub's acceptable use policy,[14] researchers *"may use public, non-personal information from the Service for research purposes, only if any publications resulting from that research are open access"*. This applies to both data directly extracted by GitHub, e.g., through GitHub's APIs, and data indirectly coming from GitHub, e.g., publicly available datasets that derive from GitHub data. As such, one approach is to focus on using non-personal data and ensuring that research findings are published as open access. Additionally, researchers may engage with repository maintainers and community members through public communication channels to inform them about the research and invite feedback. On the one hand, this may help ensure research transparency. On the other hand,

[14] The GitHub's acceptable use policy: `https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies`.

this may foster a collaborative environment, even if direct consultation with every individual contributor is impractical.

- *Informed consent* - ensuring that permission is sought, participation is voluntary, and data is anonymized, carefully processed, stored, and discarded are all essential aspects to be considered. When analyzing large projects with numerous contributors, obtaining informed consent from all participants can be impractical because many contributors might have left or lost interest in the project. In such cases, ethical considerations include respecting the contributors' privacy and ensuring that their data is used responsibly. On the one hand, GitHub's acceptable use policy solely allows the use of public, non-personal information for research purposes. On the other hand, one approach is to anonymize or use pseudonyms for contributors' names, as this protects their identities while still allowing for meaningful research.
- *Risks and benefits balancing* - the need to consider the potential harms, personal data protection, and the impact of the results.
- *Fairness and equity* - the need to consider the fair selection of subjects, data availability, and fair treatment of parties involved in the study.
- *Compliance, transparency, and accountability* - legal compliance should be considered as part of data handling. For instance, dealing with personal data may require the researcher to comply with the law of the country they are conducting the research.

Some points to consider when performing MSR studies include:

- A change of mindset from *"Here is a dataset, let us see what we can find"*, as this can be risky for the participants. So, the ethics considerations should assess the potential areas of harm, including the observations and judgments, and evaluate the impact the research results can have on the individuals/developers.
- MSR studies are often conducted on a sample of the data, which is random. Ethics could play a role in ensuring the sample represents the people involved and is inclusive regarding the questions and the process. The ethical consideration ensures that individuals and analyses are not excluded from the results.
- There are different laws and regulations regarding privacy, including the EU's General Data Protection Regulation (GDPR) and the California Consumer Privacy Act of 2018 (CCPA), in addition to the Intellectual Property (IP) laws. Various licenses also state different usage allowances. These laws sometimes restrict specific usages and research and should be considered when designing the study or before data collection. However, for the sake of simplicity, restricting research repositories with licenses that permit studies could affect the generalizability of the research findings. For instance, this aspect should be considered and actively discussed in the paper as a threat to validity.

In conclusion, educators should proactively integrate ethical MSR practices in their courses. First and foremost, it is paramount to highlight the significance of adhering to GitHub's acceptable use policy and safeguarding contributors' identities by refraining from disclosure without explicit consent. Additionally, educators can foster student comprehension of the legal and regulatory landscape governing data privacy and intellectual property rights through interactive discussions and hands-on learning experiences. Employing awareness-targeting teaching methods like quizzes and serious games may be an effective strategy to stimulate critical thinking on ethical challenges inherent in MSR research. An exemplary illustration of this pedagogical approach was provided by Teo et al. [TTA+23], who introduced an interactive, scenario-based ethical AI quiz for students to self-assess their awareness and perceptions regarding AI ethics.

Concrete examples. To exemplify how ethical concerns may be effectively addressed in the context of MSR research, we briefly discuss the strategies employed by two relevant articles published at the Mining Software Repositories Conference. The first, authored by Yamashita et al. [YAKG17], collected evolutionary data concerned with the programming skills of practitioners to publicly release an open dataset. Before releasing the data, the authors rewrote the whole change history of the Git repositories so that sensitive information, e.g., developer names and contact details, were removed or changed consistently to protect privacy and confidentiality. The second article, authored by Gonzalez-Barahona et al. [GBRIC15], released a suite of tools to extract data from software repositories, also contributing a database of human-generated data coming from open-source communities. In their article, the authors explicitly mentioned that the use of data was allowed by the organization providing those data. These two articles represent two valuable examples of how ethics should be preserved while mining software repositories and may be used by educators as case studies.

A Do for Educators. MSR studies must be compliant to ethical aspects and regulations enforced by policymakers. Educators should proactively introduce ethical MSR practices in their courses, for instance, by employing awareness-targeting teaching strategies.

5 Recent Trends and Future Outlook for Educators Leveraging Mining Software Repositories

As a last part of this book chapter, let us reflect on the recent trends in MSR research, with an outlook on the future development of the field.

The most recent advances made by the research community over the last few years reflect the dynamic nature of software development practices and

the increasing integration of advanced technologies. One prominent trend is the growing emphasis on leveraging artificial intelligence (AI) techniques to extract insights from vast repositories of software-related data. For instance, the 2024 Mining Challenge Track of the 21^{th} International Conference on Mining Software Repositories (MSR 2024) has featured a challenge on developers' ChatGPT conversations.[15] Such an example underscores the relevance of AI-driven approaches in understanding developer interactions, collaboration patterns, and decision-making processes. Educators responsible for mining software repository courses can leverage this trend to provide students with hands-on experience in applying AI techniques to analyze and interpret software-related data, fostering a deeper understanding of the complexities of modern software development processes.

Furthermore, there is a notable shift towards exploring the dynamics of software ecosystems within MSR education. This trend reflects the recognition of software systems as complex socio-technical ecosystems comprising diverse stakeholders, technologies, and dependencies. Educators are called to increasingly incorporate modules on software ecosystems into mining software repository courses, enabling students to gain insights into the interconnectedness of software projects, the evolution of software ecosystems over time, and the impact of ecosystem characteristics on software quality and maintainability. By incorporating software ecosystem analysis into their curriculum, educators can empower students to navigate the intricacies of real-world software development scenarios and equip them with the skills necessary to contribute meaningfully to software projects within diverse ecosystem contexts.

Finally, MSR education is increasingly emphasizing interdisciplinary collaborations and integrating diverse data sources and methodologies. Educators recognize the value of combining traditional MSR techniques with insights from machine learning, natural language processing, and social network analysis to address complex research questions and emerging challenges in software engineering. By fostering interdisciplinary collaboration and exposing students to various methods and tools, educators can prepare them to tackle real-world software engineering problems effectively and drive innovation in mining software repositories.

6 Sample List of Repositories

- Bugzilla: https://www.bugzilla.org/
- Bitbucket: https://bitbucket.org/
- GitLab: https://gitlab.com/
- Azure Repos: https://azure.microsoft.com/en-us/services/devops/repos/
- Google Code: https://code.google.com/archive/

[15] MSR 2024 Mining Challenge: `https://2024.msrconf.org/track/msr-2024-mining-challenge?`

- Jira: https://www.atlassian.com/software/jira
- ProjectLocker: https://www.projectlocker.com/
- CloudForge: https://cloudforge.com/
- Zenodo: https://zenodo.org/
- Awesome GPT: https://gpt4.tools/
- Docker Hub: https://hub.docker.com/
- Kaggle: https://www.kaggle.com/

Discussion forums
- Reverse Engineering: https://reverseengineering.stackexchange.com/
- Software Engineering: https://softwareengineering.stackexchange.com/
- Software Quality Assurance and Test: https://sqa.stackexchange.com/
- GenAI: https://genai.stackexchange.com/ - DevOps: https://devops.stackexchange.com/
- Hash Node: https://hashnode.com/
- Dev: https://dev.to/p/editor_guide
- Code Project: https://www.codeproject.com/

Online coding platforms used by developers:
- Code Pen: https://codepen.io/
- Replit: https://replit.com/

References

AEG+23. Aly Abdelrazek, Yomna Eid, Eman Gawish, Walaa Medhat, and Ahmed Hassan. Topic modeling algorithms and applications: A survey. *Information Systems*, 112:102131, 2023.

APSW19. Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

Bas94. Victor R Basili. Goal, question, metric paradigm. *Encyclopedia of software engineering*, 1:528–532, 1994.

BHWS21. Daniel Barros, Flavio Horita, Igor Wiese, and Kanan Silva. A mining software repository extended cookbook: Lessons learned from a literature review. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pages 1–10, 2021.

Bin07. David Binkley. Source code analysis: A road map. *Future of Software Engineering (FOSE'07)*, pages 104–119, 2007.

BT18. Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

CK94. Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

CPZF19. Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 152:165–181, 2019.

Cre99. John W Creswell. Mixed-method research: Introduction and application. In *Handbook of educational policy*, pages 455–472. Elsevier, 1999.

CSR22. Preetha Chatterjee, Tushar Sharma, and Paul Ralph. Empirical standards for repository mining. MSR '22, page 142–143, New York, NY, USA, 2022. Association for Computing Machinery.

CTH16. Tse-Hsun Chen, Stephen W Thomas, and Ahmed E Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21:1843–1919, 2016.

dBRBvD19. Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. The delta maintainability model: Measuring maintainability of fine-grained code changes. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 113–122. IEEE, 2019.

DMP+20. Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. Detecting and characterizing bots that commit code. In *Proceedings of the 17th international conference on mining software repositories*, pages 209–219, 2020.

dONTF+19. Francisco Gomes de Oliveira Neto, Richard Torkar, Robert Feldt, Lucas Gren, Carlo A Furia, and Ziwei Huang. Evolution of statistical analysis in empirical software engineering research: Current state and steps forward. *Journal of Systems and Software*, 156:246–267, 2019.

DPDNPT21. Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A Tamburri. Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Transactions on Software Engineering*, 48(6):2086–2104, 2021.

DRGP13. Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95, 2013.

DVM20. Tapajit Dey, Bogdan Vasilescu, and Audris Mockus. An exploratory study of bot commits. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 61–65, New York, NY, USA, 2020. Association for Computing Machinery.

EN08. Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

Fei23. Dror G Feitelson. "we do not appreciate being experimented on": Developer and researcher views on the ethics of experiments on open-source projects. *Journal of Systems and Software*, 204:111774, 2023.

FHN+20. Davide Falessi, Jacky Huang, Likhita Narayana, Jennifer Fong Thai, and Burak Turhan. On the need of preserving order of data when validating within-project defect classifiers. *Empirical Software Engineering*, 25:4805–4830, 2020.

FJW+18. Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23:452–489, 2018.

GBRIC15. Jesus M Gonzalez-Barahona, Gregorio Robles, and Daniel Izquierdo-Cortazar. The metricsgrimoire database collection. In

2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pages 478–481. IEEE, 2015.

GFC+24. Giammaria Giordano, Gerardo Festa, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. On the adoption and effects of source code reuse on defect proneness and maintenance effort. *Empirical Software Engineering*, 29(1):20, 2024.

GK22. Nicolas E Gold and Jens Krinke. Ethics in the mining of software repositories. *Empirical Software Engineering*, 27(1):17, 2022.

GKS08. Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, 2008.

GPLNMSMR18. Diego Güemes-Peña, Carlos López-Nozal, Raúl Marticorena-Sánchez, and Jesús Maudes-Raedo. Emerging topics in mining software repositories: Machine learning in software repositories and datasets. *Progress in Artificial Intelligence*, 7:237–247, 2018.

GSP14. Monika Gupta, Ashish Sureka, and Srinivas Padmanabhuni. Process mining multiple repositories for software defect resolution from control and organizational perspective. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 122–131, New York, NY, USA, 2014. Association for Computing Machinery.

Has08. Ahmed E Hassan. The road ahead for mining software repositories. In *2008 frontiers of software maintenance*, pages 48–57. IEEE, 2008.

HJZ16. Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21:303–336, 2016.

Hod21. Rashina Hoda. Socio-technical grounded theory for software engineering. *IEEE Transactions on Software Engineering*, 48(10):3808–3832, 2021.

HZ13. Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130. IEEE, 2013.

KCM07. Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

KGB+14. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

KMB+17. Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22:579–630, 2017.

KPB18. Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. Mining file histories: should we consider branches? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 202–213, New York, NY, USA, 2018. Association for Computing Machinery.

KSA+13. Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

LXH+18. Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 373–384, New York, NY, USA, 2018. Association for Computing Machinery.

MBH+18. Zaheed Mahmood, David Bowes, Tracy Hall, Peter C.R. Lane, and Jean Petrić. Reproducibility and replicability of software defect prediction studies. *Information and Software Technology*, 99:148–163, 2018.

Men16. Tom Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–8. IEEE, 2016.

MKCN17. Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22:3219–3253, 2017.

MPS08. Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

MSRM04. Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic. An information retrieval approach to concept location in source code. In *11th working conference on reverse engineering*, pages 214–223. IEEE, 2004.

NN22. Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 1–5, New York, NY, USA, 2022. Association for Computing Machinery.

NNN13. Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 138–147. IEEE, 2013.

PSVDB11. Wouter Poncin, Alexander Serebrenik, and Mark Van Den Brand. Process mining software repositories. In *2011 15th European conference on software maintenance and reengineering*, pages 5–14. IEEE, 2011.

QNB+19. Huilian Sophie Qiu, Alexander Nolte, Anita Brown, Alexander Serebrenik, and Bogdan Vasilescu. Going farther together: The impact of social capital on sustained participation in open source. In *2019 ieee/acm 41st international conference on software engineering (icse)*, pages 688–699. IEEE, 2019.

RK11. Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52, 2011.

RPS+21. Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. Evaluating szz implementations through a developer-informed oracle. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 436–447. IEEE, 2021.

RSCB18. Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: an

empirical investigation on code change reviewability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 201–212, 2018.

SAB18. Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 908–911, 2018.

SEWK20. Margaret-Anne Storey, Neil A Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, 25:4097–4129, 2020.

SF21. GM Sullivan and R Feinn. Using effect size—or why the p value is not enough. j grad med educ. 2012; 4 (3): 279–282. doi: 10.4300. Technical report, JGME-D-12-00156.1.[Europe PMC free article][Abstract][CrossRef][Google Scholar], 2021.

SGG21. Camila Costa Silva, Matthias Galster, and Fabian Gilson. Topic modeling in software engineering research. *Empirical Software Engineering*, 26(6):120, 2021.

SHMB24. Margaret-Anne Storey, Rashina Hoda, Alessandra Maciel Paz Milani, and Maria Teresa Baldassarre. Guidelines for using mixed and multi methods research in software engineering. *arXiv preprint arXiv:2404.06011*, 2024.

SPDN+18. Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. Do developers update third-party libraries in mobile apps? In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 255–265, New York, NY, USA, 2018. Association for Computing Machinery.

SRN+24. Margaret-Anne Storey, Daniel Russo, Nicole Novielli, Takashi Kobayashi, and Dong Wang. A disruptive research playbook for studying disruptive innovations. *arXiv preprint arXiv:2402.13329*, 2024.

ŚZZ05. Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

TDX+12. Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International symposium on the foundations of software engineering*, pages 1–11, 2012.

TPB+15. Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015.

TTA+23. Wei Teo, Ze Teoh, Dayang Abang Arabi, Morad Aboushadi, Khairenn Lai, Zhe Ng, Aastha Pant, Rashina Hoda, Chakkrit Tantithamthavorn, and Burak Turhan. What would you do? an ethical ai quiz. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 112–116. IEEE, 2023.

VEL+23. Roberto Verdecchia, Emelie Engström, Patricia Lago, Per Runeson, and Qunying Song. Threats to validity in software engineering re-

search: A critical reflection. *Information and Software Technology*, 164:107329, 2023.

WNLB20. Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. An empirical study of quick remedy commits. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 60–71, New York, NY, USA, 2020. Association for Computing Machinery.

WNLB22. Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Quick remedy commits and their impact on mining software repositories. *Empirical Software Engineering*, 27:1–31, 2022.

WRH+12. Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

WRK22. Mayur Wankhade, Annavarapu Chandra Sekhara Rao, and Chaitanya Kulkarni. A survey on sentiment analysis methods, applications, and challenges. *Artificial Intelligence Review*, 55(7):5731–5780, 2022.

YAKG17. Aiko Yamashita, S Amirhossein Abtahizadeh, Foutse Khomh, and Yann-Gaël Guéhéneuc. Software evolution and quality data from controlled, multiple, industrial case studies. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 507–510. IEEE, 2017.

YR+11. Fabian Yamaguchi, Konrad Rieck, et al. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011.

YXLG22. Yanming Yang, Xin Xia, David Lo, and John Grundy. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)*, 54(10s):1–73, 2022.