*electronics*

MDPI

*Article*

# Code Smell Prioritization with Business Process Mining and Static Code Analysis: A Case Study

**Md Rofiqul Islam, Abdullah Al Maruf and Tomas Cerny ***

Computer Science, ECS, Baylor University, One Bear Place #97141, Waco, TX 76798, USA; rofiqul_islam1@baylor.edu (M.R.I.); maruf_maruf1@baylor.edu (A.A.M.)
* Correspondence: tomas_cerny@baylor.edu

**Abstract:** One of the most significant impediments to the long-term maintainability of software applications is code smells. Keeping up with the best coding practices can be difficult for software developers, which might lead to performance throttling or code maintenance concerns. As a result, it is imperative that large applications be regularly monitored for performance issues and code smells, so that these issues can be corrected promptly. Resolving code smells in software systems can be done in a variety of ways, but doing so all at once would be prohibitively expensive and can be out of budget. Prioritizing these solutions are therefore critical. The majority of current research prioritizes code smells according to the type of smell they cause. This method, however, is not sufficient because of a lack of knowledge regarding the frequency of code usage and code changeability behavior. Even the most complex programs have some components that are more important than others. Maintaining the functionality of certain parts is essential since they are often used. Identifying and correcting code smells in places that are frequently utilized and subject to rapid change should take precedence over other code smells. A novel strategy is proposed for finding frequently used and change-prone areas in a codebase by combining business logic, heat map information, and commit history analysis in this study. It examines the codebase, commits, and log files of Java applications to identify business processes, heat map graphs, and severity levels of various types of code smells and their commit history. This is done in order to present a comprehensive, efficient, and resource-friendly technique for identifying and prioritizing performance throttling with also handling code maintenance concerns.

**Keywords:** code smells; code-analysis; business process; heat map



**Citation:** Islam, M.R.; Maruf, A.A.; Cerny, T. Code Smell Prioritization with Business Process Mining and Static Code Analysis: A Case Study. *Electronics* **2022**, *11*, 1880. https://doi.org/10.3390/electronics11121880

Academic Editor: Anna Rita Fasolino

Received: 17 May 2022
Accepted: 10 June 2022
Published: 15 June 2022



## 1. Introduction

Developing large-scale projects involves long-term commitment. A single project's development strategy typically changes several times due to various known and unknown circumstances [1]. When constructing different portions of a project or making modifications, developers usually operate under strict time constraints. Under such intense and changeable conditions, software programmers can't always employ the greatest coding standards and most efficient algorithms. As a result, certain bad coding habits become ingrained in the project; these are referred to as code smells [2]. This can happen both deliberately and unintentionally. Developers may be aware of these issues, but lack time to determine proper solutions and implement them in the project [3]. Instead, they rely on the solutions they are familiar with. Such choices may lead to performance throttling or program management issues including poor reusability, testability, and maintainability over time. As a result, developers often provide post-development support to find and fix these issues to extend the app's life.

After an app goes live, developers must continuously monitor performance and repair bugs. However, to maintain software design quality and keep it compressible and evolvable at low costs, code smells should be avoided. Application architecture tends to degrade with evolution, and code smell often relates to the phenomena [4]. Almost 65% of all code

smells were correlated to 78% of all architecture problems [4]. This suggests determining and eliminating code smells will support the application's maintainability and evolvability. Detecting code smells can be done in a variety of ways [2,5]. Most use static code analysis, while others employ commit history, UML diagrams, and other methodologies. However, developers must not only identify code smells; they must also repair them in order to improve performance. The most difficult part of the development process occurs at this phase. One main drawback of current technologies is detecting too many code smells [6], overwhelming software engineers with the sheer volume of data that must be analyzed at once. Second, the developer's refactoring budget is generally too little for the time and effort required to eradicate all present code smells. Finally, not all code smells are of comparable importance to fix. An important part of a large project is determined by how frequently it is used. If a utility library's date and time format conversion function has faulty code, it may be utilized in multiple locations across the project. Significant portions of the project will be hampered by this code smell. On the other hand, consider a data verification function used by a single class. There are only a handful of instances of this class throughout the entire project. It's possible that this class has some code smells, but they may not need to be fixed immediately. To determine the significance of various code smells, it is necessary to focus on three different metrics: the importance of the affected code to the overall system, how frequently the affected code is run (typically determined by a frequency or heat map), and how rapidly a code block changes. Using these three metrics, we might be able to come up with a scale for how important code smells are.

In this study, we propose a novel approach for business process mining by using static code, log, and commit history analysis. The strategic importance of affected code can be determined by looking at the project's business logic. Business process logic shows how decisions are made and how project pieces are connected (variables, functions, classes, and so on). Moreover, combining heat maps (frequency graphs), commit information, and code smell counts can help fix code smells in large projects. To add commit information to the business process graph, we must first examine the commit history and determine how often a code unit has changed. To identify code smells, we have used traditional tool such as SonarQube [7], SpotBugs [8] and PMD [9]. To construct a heat map, we must first collect data on how each application module is used. If we monitor the program long enough, we may be able to determine how often particular methods are called in a large project. Application log files are great for creating a heat map since they capture almost all events. To build usage frequency tables for each module, we employed a pattern matching approach to examine application log files in this study. In the last part of this work, We completed a full-proof case study with performance improvement analysis on a medium-scale project with 8 microservices in a distributed environment.

To organize the paper, we have followed a standard format [10] as follows: Section 2 describes the background knowledge and associated works. The proposed technique is described in Section 3. The implementation approach is outlined in Section 4. Section 5 evaluates our case studies and identifies challenges to validity. Finally, in Section 6, there is a conclusion and discuss potential future work.

## 2. Background and Related Work

The word code smell was first defined by Fowler [11] as a harmful effect of bad coding practice and poor design decisions. However, in today's world of modern software engineering, we can describe code smells as software characteristics that make it difficult to evolve and maintain software due to code or design flaws [12]. Several tools are available on the market to identify code smells, such as SonarQube [7], Spotbugs [8], Arcade [13], Arcan [13], PMD [9], etc. These tools use static code analysis and software metrics to identify code smells. However, these tools usually do not prioritize the long list of smells by themselves. There are few works that actually work on prioritizing code smells. Their strategies will be discussed in this section.

### 2.1. Code Smell Detection and Prioritization

A systematic literature review (SLR) on prioritizing code smells in object-oriented software by Amandeep Kaur et al. [14] discussed roughly 23 available approaches and categorized them based on their nature. They have covered existing systematic literature reviews to collect the information and created a taxonomy table based on their findings of different strategies and available tools. Another important table given in this paper indexes many kinds of code smells and their abbreviations. Almost 36 types of code smells are mentioned in that list. Finally, they conducted the most important work, making a complete guidebook to clearly indicate which type of code smells are being explicitly considered by which tool for prioritization. They have observed that most code smell prioritization tools focus on code smell's nature, such as repeating code, long method body, etc. The majority of these works are Java language-oriented.

Verma et al. [15] did a study to find out which relevant parameters have the most impact on how code smells are prioritized. They paid special attention to software systems that are based on objects. Object-oriented software releases increase the quantity of code smells and they could harm program quality. This paper addresses the drawbacks of code smells and summarizes prioritization criteria, subject programs, performance measurements, and detection approaches.

F. A. Fontana et al. [16] presented a code smell intensity index that can be used as an estimate to find the most critical cases, prioritizing smell inspection and, potentially, eradication. They have computed the Intensity Index for six smells: God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, and Message Chain, which they identified by their proposed tool JCodeOdor [16]. In their work, they have created a matrix with different kinds of code smells based on the intensity, which was the main basis for prioritization. Around 74 systems were analyzed in this work to find out the intensity of six targeted code smells. The main goal of this work to identify and remove the most critical code smell first.

Another interesting work, written by Fabiano Pecorelli et al. [17], proposed a developer-driven code smell prioritization technique. The most remarkable part of their work is that they have also found the shortcomings of existing works, which mostly attempted to deliver solutions that can rank smell instances based on their severity. This paper introduced machine learning as a prospective approach that works on the developer's feedback data. For machine learning model training, they need sufficient data from various project. After checking more than two thousand projects, they found around 682 projects based on using filters of number of classes, change history, and number of commits.After that, they choose 9 projects randomly from them for evaluating their work. Developers play an important role since code smells are ranked based on the perceived criticality that developers assign to them. Since this work is heavily dependent on developers' reviews, its success rate is also dependent on the performance of developers. Generally, developers' group express their criticality review based on their experience, and there is no guarantee that it will work similarly for all applications.

Gupta et al. [18] presented a data sampling technique to improve code-smell prediction. This empirical study is targeted at imbalanced data. Using feature engineering and sampling techniques, this work tries to discover eight different types of code smells. The main contribution of this paper is using three different naive Bayes classifiers to find out code smells. They have conducted their study over 629 projects and found out that the Gaussian Naive Bayes classifier performed better than Bernoulli Naive Bayes and Multinomial Naive Bayes. Another recent machine learning based work [19] work had conducted for Java and Kotlin based applications. The experiment compares machine learning algorithms for Kotlin code smell detection. JRip's 10-fold cross-validation showed overall good precision and accuracy.

Santiago Vidal et al. [20] have introduced a new tool JSpIRIT which takes Java source code as input and produces as output a ranking of code smells. The key benefit of utilizing JSpIRIT is that developers may customize and extend the tool by offering multiple ways

to recognize and rank the code smells. The tool includes hooks for implementing smell identification algorithms and evaluation criteria. Furthermore, JSpIRIT enables combining several algorithms for detecting code smells and the design of variable prioritizing strategies for rating the code smells. The supported code smells by JSpIRIT are Brain Class, Brain Method, Data Class, Dispersed Coupling, Feature Envy, God Class, Intensive Coupling, Refused Parent Bequest, Shotgun Surgery, and Tradition Breaker. JSpIRIT core, Detection strategies, Mapping Data, Prioritization Criteria, and User Interface are the main components of JSpIRIT tool.

A recent work done by R Singh et al. [21] shows a remarkable approach for code smell identifying, prioritization, and fixing strategies to reduce maintenance costs. In the code smell identification part, it shows strategies to identify seven different types of code smells, such as feature envy, God class, long method, long parameter, refused request, shotgun surgery, and duplicate code detection. It considered three important criteria for prioritizing code smells, such as the severity of code smells, maintenance efforts, and user perception. Finally, they explored the most effective strategy to fix high-priority code smells with limited resources. This work is very admirable since it tried to find out the different factors that can play an important role in code smell prioritization. However, they missed some important information, such as code evolution history and code usage frequency and pattern.

Stedi and Eder [22] introduce a refactoring recommendation strategy to prioritize quality defects and code smells. This work focuses only on two specific types of code smells, code clones and long methods. To detect these two maintainability defects, they first used the ConQAT tool [23] and then identified refactoring opportunities based on developers' feedback. They came up with a number of heuristics algorithms for figuring out which code smells were the most important and tried them out on seven industrial and six open-source applications.

There are a wide range of tools and techniques available in the market to detect and prioritize code smells. A comparison table of these works given in Table 1. Most of these approaches prioritize code smells based on the type and severity of their impact. In the real world, however, we cannot conclude that this form of universal indexing will work for all projects, because the same type of code smells may not affect various applications in the same way. We seek to use business process mining and heat maps to gain a better understanding of this problem.

**Table 1.** Code smell detection and prioritization result.

| Ref | Category | Language | Data Set | Used Tools | Approach |
|---|---|---|---|---|---|
| [14] | Systematic Literature Review | Java, C++, C, C# | Existing 23 works for code smell prioritization | N/A | Performance analysis of existing approach against different kind of code smells |
| [16] | Research Prototype | Java | Analyzed 74 systems to find out intensity of six specific code smells | JCodeOdor | Matrix based |
| [17] | Research Prototype | Java | Randomly selected 9 projects from 682 | Decor [24], Hist [25] | Machine learning based |
| [18] | Research Prototype | N/A | 629 open-source projects | N/A | Naive Bayes classifier |
| [20] | Research Prototype | Java | Logical assumption and testbed project | JSpIRIT | Works based on developer's own criteria |

**Table 1.** *Cont.*

| | | | | | |
|---|---|---|---|---|---|
| [21] | Research Prototype | Java | N/A | N/A | Works based on severity of code smells, maintenance efforts, and user perception |
| [22] | Research Prototype | Java | seven industrial and six open-source applications | ConQAT [23] | Refactoring recommendations using various heuristics algorithms |

### *2.2. Static Source Code Analysis*

Static code analysis can process the program's source code to predict all possible outcomes during run-time execution [26]. It cannot, however, predict how users will use the program. For various programming languages, there are a variety of tools available for static source code analysis [27]. For example, as recognized in literature, common source code analyzers for the Java programming language are Raxis, PVS-Studio, reshift, CodeSonar, etc. [28]. The two static code analysis approaches that result in a representation of the program are source code analysis and bytecode analysis, respectively. A number of techniques are used to accomplish this, including recognizing components (such as classes and methods), tokenization (such as fields and annotations), and parsing (which results in graph representations of the code). A few examples include Abstract Syntax Trees (AST), Control-Flow Graphs (CFG) [29–31], and Program Dependency Graphs (PDG) [32,33]. Bytecode analysis employs the application's compiled code to discover endpoints, components, classes, and functions. It can supplement or originate CFG or AST. However, not all languages have bytecodes. In source code analysis, we examine the application's source code without compiling it. There are numerous ways to accomplish this; nevertheless, the majority of tools pre-process the code and generate trees, such as AST, CFG, or PDG.

### *2.3. Log Analysis*

Log analysis is a prominent area of study, as logs typically record numerous types of information throughout application runtime. Developers depend heavily on logging mechanisms for finding performance abnormalities and bugs in the period of contemporary software engineering. The phrases static logs, runtime and traced logs will be used extensively in this research work. As a result, recognizing the differences between them is critical to comprehend the entire work. In this paper, the set of logs identified by static code analysis is referred to as static logs. During static analysis, we must loop over each line of source code to find logging statements. Following this, we must prune out the log message from these code statements and save them for further processing. Runtime logs, on the other hand, are the logs generated when the application is executing. The goal of logging is to keep track of error reporting and related data in one place. Logging should be utilized in large apps, but it can also be used in smaller apps, especially if they perform a critical role. Generally, this kind of application log is stored in log files. Log files can show any discrete event within an application or system, such as a failures, errors, or state transformations. Lastly, the traced logs encompass a much wider, continuous view of an application. In many cases, tracing illustrates a journey through a whole app stack. By tracing through a stack, developers can identify bottlenecks and focus on improving performance [34]. Multiple technologies are available for log tracing such as opentelemetry [35], Test-to-Code Traceability, Ref. [36] or profiling application. Jaeger (https://www.jaegertracing.io, accessed on 16 May 2022) and Zipkin (https://zipkin.io, accessed on 16 May 2022), can aid in the analysis and visualization of the life cycle of an HTTP call in a complex microservice architecture, which in turn aids in the analysis of system performance and more effective debugging of issues by identifying the source of the problem. As microservices are becoming the new standard for web applications, distributed

tracing technologies are increasingly important for troubleshooting. In addition to tracing, service mesh technologies, such as Istio (https://istio.io, accessed on 16 May 2022), can provide metrics for each service, and visualization tools that use Istio metrics, such as Kiali (https://kiali.io, accessed on 16 May 2022), can use these metrics to build a dependency graph of microservices. Such a graph can aid in detecting architectural smells, including cyclic microservice dependencies or unbalanced traffic in the design. However, distributed tracing does not easily identify service level events or business logic information. In order to make a relationship between an application's static and runtime characteristics, a new sort of log analysis must be performed.

### *2.4. Business Process Mining*

BPM stands for business process mining, and it is a technique for extracting business logic from data sources. [37]. From these event logs, BPM can extract several types of data, including process, control-logic, etc. [38]. Process mining is composed of several techniques such as process discovery and visualization, conformance checking, performance analysis, root-cause analysis, prediction analysis, process management life-cycle, process monitoring, etc. [39]. Various BPM tools are available, such as Celonis, ProcessMiner, QPR process analyzer, PROM tool, and more.

### *2.5. Commit History Analysis*

The evolution of large-scale applications necessitates numerous changes. Code repository tools, which are becoming increasingly popular in modern software engineering, are being used to keep track of these changes. There are numerous tools available for storing and managing application code, including SVN and Git, among others. Git is the most popular at this moment, and commit messages are required in Git for any codebase update that involves changing the code. For this reason, analyzing git commit history is a hot topic for research. Behnamghader et al. [40] conducted research on understanding software quality evolution through commit-impact analysis. They look at each contribution to see if it changes the source code or not. If each important commit can be compiled or not. How changes to the source code affect measures of software quality. How useful it is to use a certain metric as a measure of software quality, etc. Another important work done by Zanjani et al. [41], which works on impact analysis on source code due to change requests by analyzing commit histories. Additionally, in this research, we employed git commit history analysis to identify sections of the codebase that are prone to rapid change.

## 3. Proposed Approach

In this section, we give a high-level overview of our proposed approach, which will help the reader understand the implementation strategy. In this work, we plan to use business process logic to find out the strategic importance of each function in the whole project. To use business process logic as a basis for code smell prioritization, we need to build a tool for extracting business processes. Application logs are a valuable source for business process extraction since they record almost all application events. However, using run-time application logs can be very costly for multiple reasons. Many simultaneous actions make it difficult to correlate logs for numerous requests in a log file. If we wish to leverage the application's runtime logs for business process extraction, we must employ a machine learning model to find links between different log messages. On the other hand, using static log analysis can help us to identify logic behind log printing statements and also their execution order. For this reason, this paper uses static code analysis to generate a business process graph.

The business process graph gives us a high-level perspective of a large-scale project's operating mechanism, but we can't pinpoint the densely occupied zone. For this reason, after creating an application's business process graph, we produce a heat map to identify heavily used areas during runtime. A heat map graph is a representation of the utilization frequency of different modules in a project. There are many ways to generate a heat map

graph such as using open telemetry [35]. It helps you study the performance and behavior of your software by allowing you to the instrument, generate, collect, and export telemetry data (metrics, logs, and traces). There are, however, significant drawbacks to adopting open telemetry. Using the endpoint probing technique, Open telemetry technology was primarily utilized to study data flow between distinct microservices. As a result, it can track requests and responses across microservice architecture, but not service-level operation or business logic. However, we require that information in order to prioritize code smell in this project. For this reason, we have used manual runtime analysis of application logs and tracing events from that. Moreover, code maintenance concerns are intimately tied to the progress of the project; hence, it is crucial to correctly maintain significantly changeable code blocks. Due to this, we have additionally incorporated code evolution history and heatmap by analyzing commit history. After that, we'll use various tools to identify the application's code smells. Finally, we'll integrate the results of the business process graph, commit history, heat map, and code smells identification into a single graph. Figure 1 depicts the architecture of proposed method. In the following sections, a short summary of each component of the architecture will be provided. Following that, the implementation plan for the entire proposed method is briefly detailed.

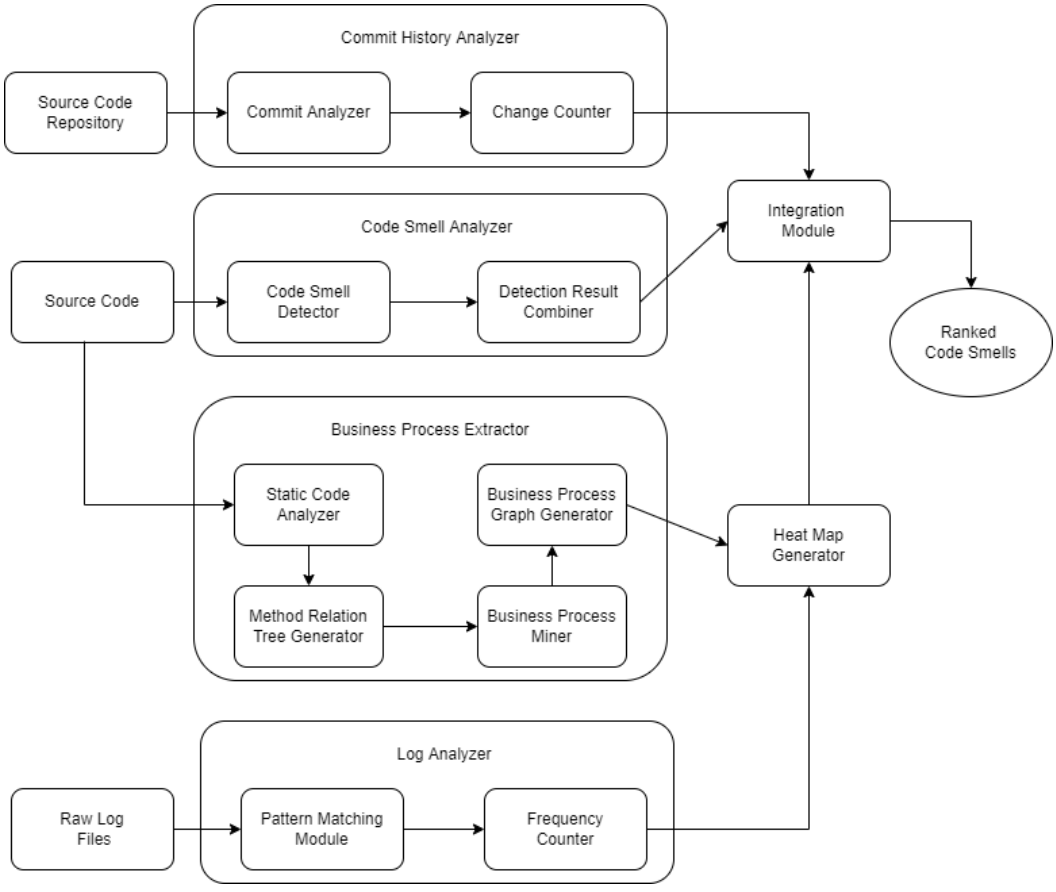


**Figure 1.** Architecture of Proposed Method.

### *3.1. Input*

*Source Code and Source Code Repository*: This study's static analysis was done on the application's source code. This prototype project is currently only compatible with Java. Any Java code base will work, but for the best results, we need a project that has a solid logging mechanism built in. This approach's success depends on the accuracy of business process mining data collected from log messages. The ideal logging method should be able to detect any sort of event that occurs during the application's operation. Various modern software engineering tools enable version control for storing and managing source code

and other data. Version control systems employ commit messages to maintain track of all changes made to a project. Currently, Git is the most extensively used code repository and version control system. This project only considers Java programs that use Git for source code version control.

*Log Files:* The second input is the application's runtime log files. We used dynamic analysis to examine large-scale applications' log files, which typically contain a record of each business logic execution. This project benefits from log files with a standard format. The static and dynamic elements of each log message must be identified during the dynamic analysis process.

### 3.2. Business Process Extractor

The most crucial step is to extract business processes. This module tracks code flow by identifying business process logic that underpins all logical judgments. This module is made up of sub-modules. These modules are described as follows:

*Static Code Analyzer:* A static code analyzer is required to parse source code. This module analyzes source code to find and classify classes, methods, variables, enumerations, if-else clauses, loops, annotations, and other data. Static code analyzers are of two types. Firstly, source code analyzers; secondly, byte code analyzers. Source code analyzers work with all compiled programming languages, whereas byte code analyzers only work with interpreted languages. The source code analyzer was utilized for this project, even though Java provides for both.

*Method Relation Tree Generator:* After reading the source code, static source code analyzers let us keep track of all the method details. There are if-else clauses, variables, and loops, as well as the origin class and argument. Then we used this module to attempt to connect them. This module receives source code analysis results and creates a graph showing how each method is related to the others.

*Business Process Miner:* Business processes underpin all logical decisions and events that occur during an application's runtime, and they should be logged. This module's main goal is to trace runtime code based on business processes. This module uses graph traversal to extract business processes from a method relation tree by arranging the log printing statements stored in each node.

*Business Process Graph Generator:* After mining business processes, our next objective is to portray them in a graph structure that is much more intuitive for consumers to understand and apply. This module does the conversion by taking the results of business process mining as input and displaying them graphically.

### 3.3. Log Analyzer

The log analyzer module looks at the app's runtime logs. We used log analysis to count the number of times each log printing statement was executed in this project.

*Pattern Matching Module:* To count each log printing statement, we must first locate them in the application's log files. This calls for a pattern matching algorithm. Log messages are usually split into two parts. In the first segment, the message statements are static and identical between executions. The second portion is dynamic and contains information about the program runtime. So we must focus on just the static component of each log report. The pattern matching module handles everything.

*Frequency Counter:* Depending on the pattern matching module's results, the frequency counter module keeps track of how many times each log printing command is run. To do this, it scans all available log files for patterns to match and increments the counter for each unique message.

### 3.4. Heatmap Generator

In the heatmap generator, business process extraction data are combined with a usage frequency counter. The business process graph and the frequency counter table are

compared for log messages. A heatmap is a specialized graph that contains data about business activities.

### 3.5. Code Smell Analyzer

The goal of this project is to rank code smells. This module finds code smells using several code smell detection methods and integrates the results using duplicate data resolution. *Code Smell Detector:* There are various code scent detectors on the market. They don't all work the same as others. Their code smell detection algorithms are also diverse. This project uses numerous code smell detection tools to reach the best results.

*Detection Result Combiner:* Because we used multiple techniques to find code smells, it's possible that the result contains duplicate data. To uniquely detect every code smell, we must remove redundant data. We must also categorize and store the code smell detection results based on their source (class and method).

### 3.6. Commit History Analyzer

With the aid of the commit history analyzer, we can determine which classes and methods experienced the most changes throughout program development. This module is responsible for assessing commit history and recording all types of project-wide changes.

*Commit Analyzer:* Git automatically creates change history for each file, but we need it for each method. So we have to carefully analyze each file's changes to see which methods are affected. This information is sent to the change counter sub-module, which generates the commit history table.

*Change Counter:* During this phase, a commit history table is constructed to record which methods have been modified and how frequently. This table's data will eventually be integrated with business process and heatmap data.

### 3.7. Integration Module

The integration module collects the results of business processes, heatmaps, commit history tables, and code smell detection. This will be used to prioritize code smells. This project presents a code smell ranking algorithm for prioritization and output production.

## 4. Prototype Implementation

In this section, we will discuss briefly the proposed solution's implementation procedure. We have broken this content into numerous smaller sections for clarity.

### 4.1. Business Process Mining by Static Analysis

To implement our BPM-SCA tool, we must first build a special static source code analyzer. We choose Java as our implementation language. There are many available parsers for static Java code parsing. In this work, we have used an open-source library for parsing source code [42]. It searches identified constructs, generally classes, for variables, dependencies, method bodies, if-else clauses, loops, annotations, and other items. The steps of a design are described in following sections.

#### 4.1.1. Class Relation Tree Construction

A class relation tree is a graph that shows the connections between multiple classes in a project. All classes are detected initially and then processed in the project directory. Classes in the Java programming language can be linked in two ways: via importing and annotating. Each class's import declarations and annotation declarations enable the construction of a relation tree graph between all classes in Java. As a result, it's possible to figure out which classes are linked to each other. Using an open-source parsing tool, we generate a class relation tree from imported and annotated classes. We produced a class id using the class name and class path to uniquely identify each class in a project. We have used a hash map to store the graphical representation of the class relation tree, which gives us the $O(1)$ solution for accessing and searching over the tree.

#### 4.1.2. Method Calling Graph Construction

To arrange the logs in run-time execution order, we must first identify the method execution order. One can utilize the method calling graph to determine the order of method execution. Method models are representations of methods in a project that include method id, name, class, in-method variable list, invoked method list, argument list, log list, and line number. This graph similarly uses a hash map. We first crawl through each class to locate all available methods using an open-source static source code parser. We gave them a unique id consisting of a class id, method name, and parameter list. Each method model class object's invoked method list can be utilized to build a method model tree. The Java language has difficulty determining the invoked method due to method overloading, overriding, and similar named methods in various classes. Parameter lists helped us handle method overloading. As a run-time process, method overriding is difficult to handle. In this example, we used our static source code parser to detect the inheritance declaration. Similarly, named methods from other classes are handled by their id. We parse each method body and invoke each method. We disregard methods that are either part of Java or third-party libraries because we are only looking at methods implemented in the project. We match called methods with methods in our method model list, which comprises every method in the project. If it matches, we add it to our caller method's invoked method hash map; otherwise, we ignore it. In Listing 1 we can see an example of a method model.

#### 4.1.3. Identifying Log Printing Statements

Log printing is a common strategy for keeping track of actions and events. There are many popular log printing strategies available on market. Java has a built-in logging framework, but there are also many popular third-party solutions like Log4j, Logback, Log4j2, SLF4J, etc. Each library has a different strategy to identifying log printing statements, making it tricky to implement for multiple. Thus, in this implementation, we are only considering the Log4j library, which has 7 different logging types—*INFO, DEBUG, WARN, ERROR, FATAL, TRACE, ALL*. When we see one of these tied to a logger object in a method body, we identify it as a log printing statement and build an object with the log message and line number. We then saved this object into the log-list which belongs to the method model. Traditional log printing statements are divided into two parts: static messages and variables. During static analysis, we don't have variable values, thus we replace variable names with a specific string.

#### 4.1.4. Merging Log Printing Statements with Invoked Methods

The method models' log reporting statements are now linked to the methods called. There are just two circumstances to consider when ordering the logs in runtime order. These two steps lead to a log reporting and method invocation statement. To complete the procedure, the called method must be accessed first before the prior method. When the first event occurs, the output receives the log print. The line number determines the order of these events. Traditional log messages have two parts: static and dynamic. Static strings are strings that do not alter. The dynamic parts of log messages contain the application's run-time generated data. Since we are static analysing the logs, we will perceive them as variables or objects. We don't need the dynamic part for now, thus we'll only save the static.

**Listing 1.** JSON Format of Sample Method Model Object.


```
"/src/main/java/edu/baylor/ems/service/ExamService.java_finishExam_1": {
"id": "/src/main/java/edu/baylor/ems/service/ExamService.java_finishExam_1",
"methodDefination": "public ResponseEntity<Object> finishExam(Integer id)",
"methodName": "finishExam",
"classPath": "/src/main/java/edu/baylor/ems/service/ExamService.java",
"methodBegin": 188,
"methodEnd": 230,
"modifiedVariableList": [
"same",
"correct"
],
"variableList": [
"exam",
"q",
"same",
"correct",
"ch",
"questions",
"optionalExam",
"currentDate"
],
"logList": [
{
"log": "Setting exam status as done",
"line": 7
},
{
"log": "Performing persist operation on updated exam",
"line": 20
},
{
"log": "Updating the exam as correct",
"line": 18
},
{
"log": "Returning the result",
"line": 22
},
{
"log": "Checking the validity of exam questions ",
"line": 13
},
{
"log": "Perform database query to find exam by id ",
"line": 1
},
{
"log": "Updating the data of found exam ",
"line": 6
}
],
"parameterList": [
"Integer"
],
"invokedMethodList": [
{
"methodId": "/src/main/java/edu/baylor/ems/model/Exam.java_getExamDate_0",
"calledLine": 11
},
{
"methodId": "/src/main/java/edu/baylor/ems/model/Exam.java_setStatus_1",
"calledLine": 13
},
{
"methodId": "/src/main/java/edu/baylor/ems/model/Exam.java_getId_0",
"calledLine": 15
},
{
"methodId": "/src/main/java/edu/baylor/ems/model/Exam.java_setSum_1",
"calledLine": 16
},
{
"methodId": "/src/main/java/edu/baylor/ems/model/Choice.java_isCorrect_0",
"calledLine": 22
},
{
"methodId": "/src/main/java/edu/baylor/ems/model/Choice.java_isChosen_0",
"calledLine": 22
},
{
"methodId": "/src/main/java/edu/baylor/ems/model/Exam.java_setCorrect_1",
"calledLine": 32
}
]
}
```


#### 4.1.5. Top-Level Method Identification

It is now possible to traverse our method calling tree because we have completed the construction process. The starting point (the root nodes) must be supplied first. Nodes designated as top-level methods are those that have no other methods calling them. DFS

(Depth First Search) [43] can be used to identify the nodes from the method tree, which can be determined using a topological sort [44] based on the traversal completion time of each node.

#### 4.1.6. Definition of Case and Events

Before commencing the traversal process, it is important to address two topics that are critical to the procedure: *cases* and *events*. A case is a method tree branch that is traversed from one top-level method to another and then returned to the top-level method. Events are nodes that exist in a branch of a traversal and are referred to as such.

#### 4.1.7. Traversing the Method Calling Graph to Arrange Log Printing Statements in Run-Time Execution Order

The traversal operation can begin at this point. A new case id will be generated for each top-level method as it is explored. Any log printing statement in the combined list of top-level logs and methods creates an event id and writes it to the output file. This method's processing will be halted, and work with the newly invoked method will begin by iterating through the combined list of the newly invoked method, as indicated above. This method of gathering logs is called "logs tracing mechanism" [45].

#### 4.1.8. Business Process Mining

The business process mining graph is generated based on the case-id and event-id of every log printing statement. From the output file, we will collect every log printing statement with the same case id, sort them based on the event id, and connect them all as branches of the business process mining graph. In Figure 2, there is a sample business process graph.

### *4.2. Commit History Table Generation*

This module builds a data table with update actions for each method in a large project. As we know, code smells are affects code maintainability most. For this reason, we must concentrate on maintaining the maintainability of the codebase's extremely change-prone portions. Because of the rapid evolution of code, these sections of code blocks are the most affected by code smells. As a result, it is very critical to find out the highly change-prone area in a code-base. The proposed heatmap graph is constructed using information about each method, and we intend to incorporate information on the methods' evolution as well. As a result, we must ascertain the evolution history of each method. As part of this step, we will develop a data table that will contain the evolution history of each method used in a large-scale project, which will be used later.

A Git commit history analysis was performed in this project in order to identify areas that were especially susceptible to change. Git is the version control system that is most extensively used in contemporary software engineering today. Using Git, it's easy to determine the evolution history of each file, but for each method, we must study its commits. Git has a very sophisticated CLI (Command Line Interface), which includes a large number of powerful commands. The "Git log -L" command is one of them, and it can be extremely beneficial for our purposes. Through the use of this command, we can track down which methods have been modified by supplying the class and method names as parameters into that command. We have stored these data in commit history table. Additionally, this information will be included in the business process graph.

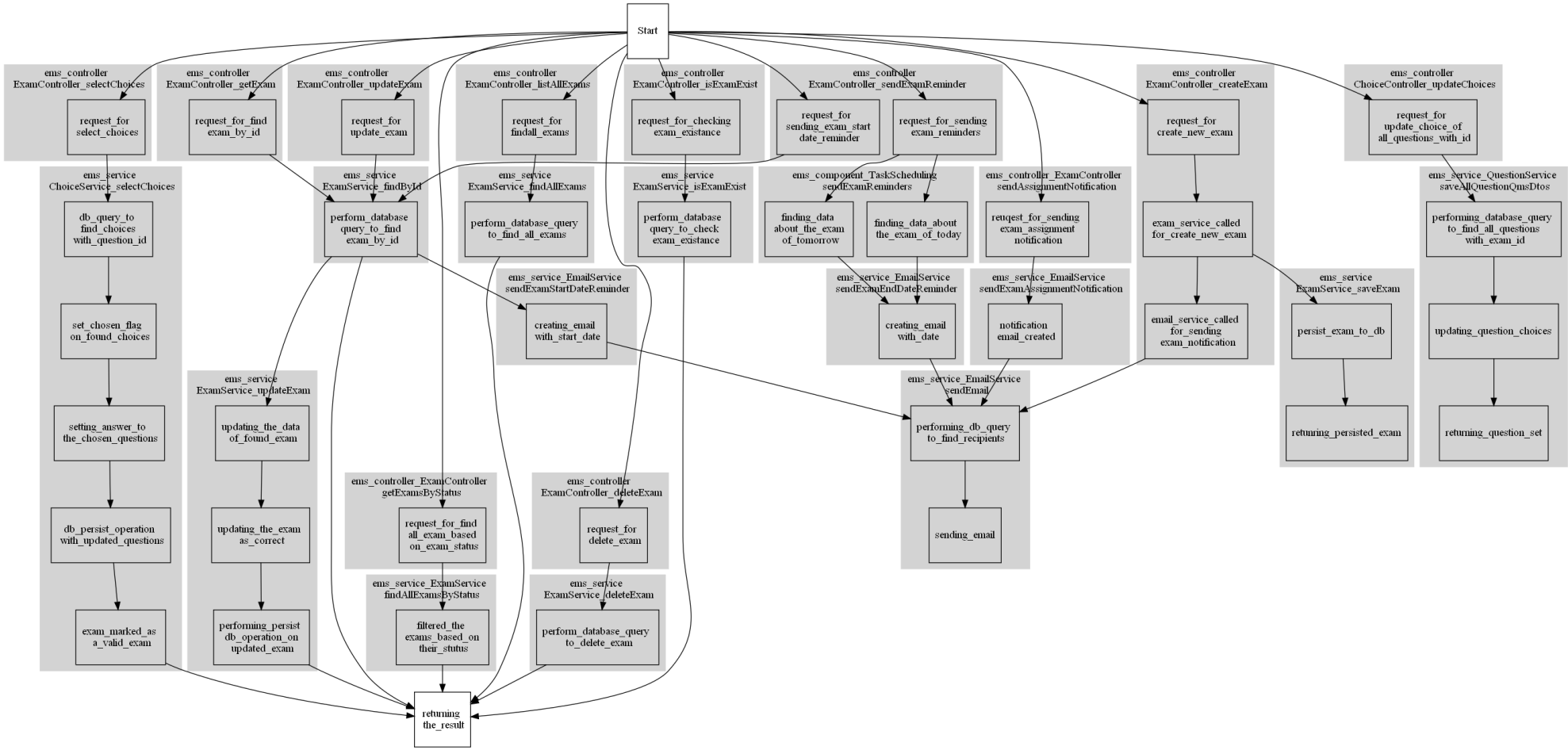


**Figure 2.** Business process mining sample graph.

### *4.3. Generation of Heat Map Graph*

The business process graph provides a high-level picture of a large-scale project's functioning mechanism, however, it cannot identify frequently used system zones. For this reason, after creating an application's business process graph, we develop a heatmap to identify heavily used parts. A heatmap graph is a representation of the utilization frequency of different modules in a project. There are many ways to generate a heatmap graph such as using distributed tracing with open telemetry [35], runtime log analysis, and runtime dynamic profiling through aspect-oriented programming (AOP) etc. The following section will explore various ways for collecting data to build heatmaps and their limitations.

#### 4.3.1. Distributed Tracing and OpenTelemetry

Our role as developers requires us to respond fast to production incidents and resolve them as quickly as feasible. To do so, we need to gather a lot of data quickly so we can understand what's going on in production and respond quickly to difficulties. Understanding and troubleshooting our system has become more difficult as distributed architecture and third-party services have grown. Metrics and logs are no longer able to give the required insight throughout our distributed system. In this case, OpenTelemetry and distributed tracing are both useful. Distribution tracing is becoming an increasingly important method for diagnosing and correcting performance and other issues in distributed systems. OpenTelemetry is now the fully open-source standard for obtaining distributed traces, which helps us identify and resolve system failures. Many commercial distributed tracing technologies exist, such as Jaeger tracing, Zipkin tracing, and Spring Cloud Sleuth.

Distributed tracing is a great tool for studying program performance and behavior by generating, collecting, and exporting telemetry data (metrics, logs, and traces). However, using distributed tracing has considerable limitations. Distributed tracing technology uses endpoint probing to investigate data flow across microservices. As a result, it can track requests and answers across microservice architectures, but not service-level operations or business logics. It combines runtime data with business operations, which distributed tracing cannot. So we couldn't use cutting-edge technologies in this project.

#### 4.3.2. Runtime Logs Analaysis

Another method to identify runtime information is analyzing the runtime log files. The ideal logging mechanism should detect any kind of events while the application is in running state. For this reason, log messages are ideal candidate for dynamic analysis. As with AOP, logging mechanisms are not uncommon. The majority of large-scale applications rely on logging to discover anomalies and evaluate performance. Since we do not have to add or modify the existing code in this approach, we have selected this way for our project.

To begin, we gathered application log files and attempted to decipher the static and dynamic components of each log message. In this stage, we focused on the static element of each log message since we plan to correlate it with the business process graph, a static action. Generally, a heatmap comprises information about the frequency of use of various modules. As a result, we must identify and count the executions of each log printing statement in log files. To do that, we used a pattern-matching program, Grok filter [46] to reconstruct the events. This third-party, open-source project can match patterns and count the occurrence of each pattern. Following that, we integrated the result of the occurrence count with the result of the business process mining by cross-checking the static section of each log printing statement. This new merged graph is referred to as a Heatmap.

#### 4.3.3. Generation of Usage Frequency Table

In our implementation, at first, we have to collect run time log files from the system by running the targeted project for a certain amount of time. We have used both automated tests with Selenium and manual tests with real users to ensure that the system gets enough requests to generate enough runtime logs for our approach to work with. When the

application reaches a log printing statement in the project at run time, it writes that log message to the log file. Since we are trying to count occurrences of this type of event in the run time log, we need to focus on the static part of each log message, as they remain the same in every occurrence of a single log printing statement.

Log messages can be pruned by using pattern matching. The static components of all log messages were previously kept in the business process graph generating section utilizing static analysis. A log pattern matching tool such as Grok exporter can be used to analyze the list by turning the static log messages into message patterns and feeding them into the tool. One of the most useful features of Grok [46] is its ability to transform log files into structured data. For Syslog, Apache, and web server logs, as well as MySQL and other database logs, Grok includes more than 120 predefined patterns. Custom patterns may be added quickly and easily to Grok. The Grok exporter can output a frequency countermeasure for each type of log via pattern matching. We utilized the open-source monitoring system Prometheus [47] to examine that measure. In Prometheus, each log printing statement is counted in a data table.

#### 4.3.4. Generation of Heatmap

The count result and the business process graph must be joined to create the heatmap graph. We can get the occurrence count of each log statement in the business process mining graph from the usage frequency database. Data is recorded in CSV format in the business process mining step, which can be readily transformed to a DOT file. The log messages and counting information were cross-checked against the business process data, and the counting information was appended to the CSV file. For converting a CSV file to a DOT file, we utilized a simple Java script, and then we used the Graphviz tool [48] to convert the DOT file to a graphical representation. A sample heatmap graph can be found in Figure 3.

### *4.4. Code Smell Relation to Business Process*

In this experiment, we have used SonarQube [7], Spotbugs [8] and PMD [9] as code smell detectors. All three of them use static code analysis to detect code smells. We gather all code smell information from these tools and merge them together, pruning out duplicate results. This phase is critical, since different tools can identify the same code smells, resulting in data duplication. We then sorted the code smells based on their source, which means combination of class and function.

### *4.5. Merging Code Smells with Heat Map (Integration Module)*

At this step, we need to add the resulting code smell detection to the heat map graph. After merging these two results, we can generate the combined graph of business processes, code smells, and utilization frequency. In Figure 4, we can see a sample of this combined graph. This graph is actually the basis of our prioritization operation since it holds all the data that is required to make an optimal decision. In this graph, we can see multiple nodes in a cluster. Each cluster indicates a method in the project, and their label represents the source class and method name. Moreover, highly-critical, semi-critical, and non-critical code smells related to a method are also given as additional information in the related cluster to that method. Each node in a cluster represents a single log printing statement in that function body.

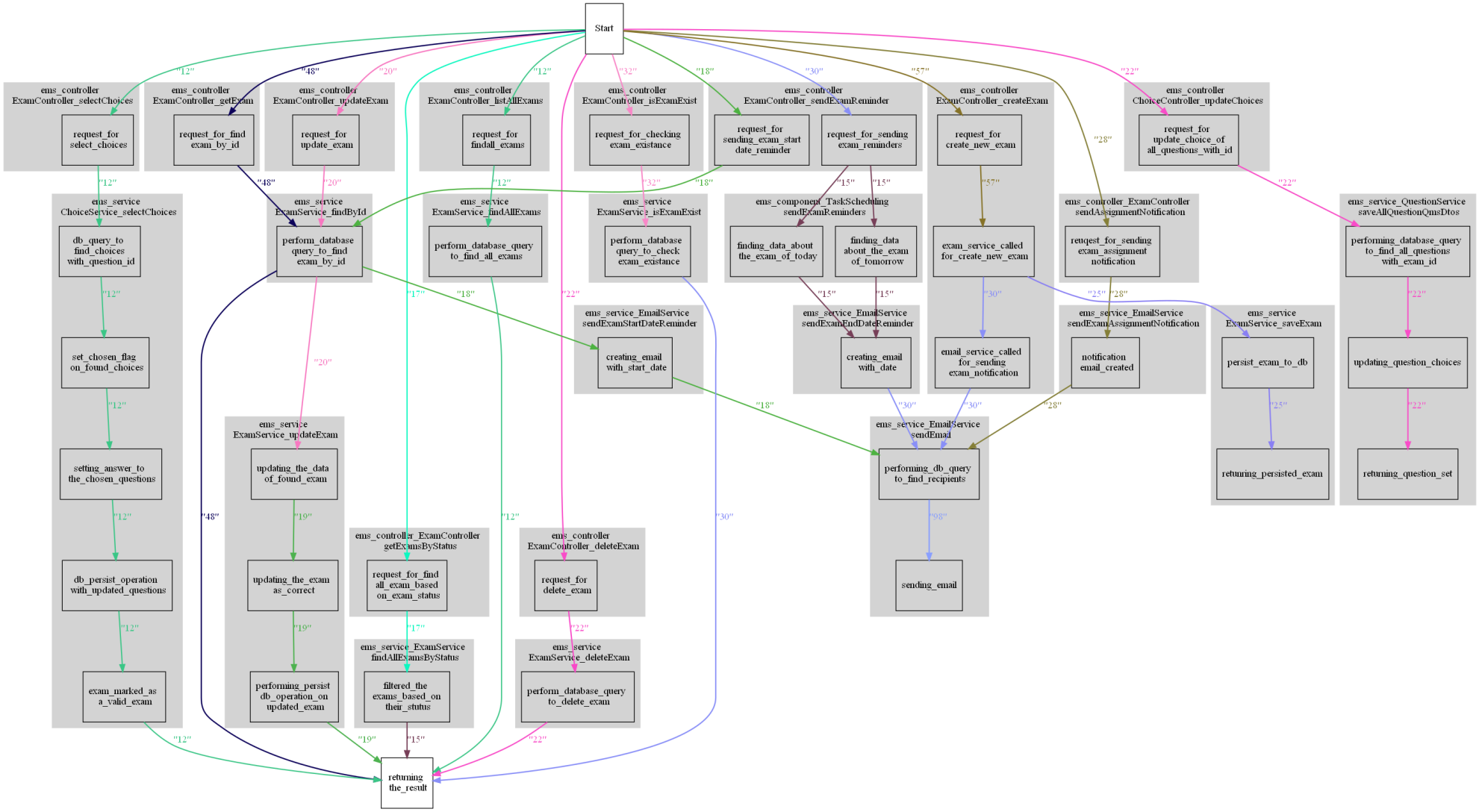


**Figure 3.** Utilization frequency sample graph (Heat-map).

### *4.6. Code Smell Prioritization Metric*

Our main goal is to prioritize the code smells in our system so we can correct the most significant ones to improve software performance with limited resources. We have five different independent variables in our hands to make this decision. Those variables are:

1. **The number of incoming and outgoing edges of all nodes in a cluster:** Each cluster represents a function in the application project, and nodes represent log printing statements in that method body. In Figure 4, we can see the combined graph of business processes, code smells, and heatmap. Our first independent variable is the summation of all incoming and outgoing edges of every node in a cluster in this graph. When a cluster's nodes have many incoming and outgoing edges, it means that many application modules or nodes use this module. Which makes the cluster particularly essential. We should give more priority to the code smells inside that module.
2. **Value of incoming edges for a particular node:** The value of the incoming edges of a node is the second independent variable in our model. The heatmap graph and business process graph are coupled to display the number of hits a node receives from another node. For a variety of reasons, we are only evaluating the incoming edges in this case, rather than both the incoming edges and the outgoing edges or only the outgoing edges. First and foremost, if we merge both, we may encounter a problem with data duplication. Furthermore, when a request hits an exception or data error in a function, we can only witness the arriving edges, not the outgoing edges. However, the module received hits for these types of instances, and we will need to account for that as well. Due to these reasons, we concluded that inbound edges are the best independent variable to express a node's usage frequency.
3. **Number of commits in each cluster:** Number of commits related to each method depicts the information about the evolution history of that method. This data will assist us in determining the degree to which each method is susceptible to change.
4. **Number of code smells in each cluster:** Using code smell detecting tools (Soanrqube, Spotbugs, and PMD), we have identified code smells in each function of our test-bed project and added them to their correspondent cluster in the combined graph of business processes, heatmap, and technical code smells (Figure 4). The count of code smells is the third independent variable for the prioritization metric.
5. **Code smell criticality:** We classified all identified code smells into three categories based on the results of the code smell detecting technology we employed in this project: highly critical, semi-critical, and non-critical. Highly critical code smells have a criticality level of 3, semi-critical code smells have a criticality level of 2, and non-critical code smells have a criticality level of 1.
6. **Code smell fixing cost:** For prioritization, the cost of fixing code smells is an important factor, since our main target is to maximize the performance using our limited resources. But code smell detecting tools do not give us the cost of fixing the smells, and it is a different research area. Maybe in the future, in our extending work, we will work with the exact value of repair cost. However, at this time, we count all types of code smells as requiring the same amount of resources to repair.

By combining all six independent variables, we can generate a common equation for prioritization. Here are some important terms that we need to understand first to reach the prioritization equation:

$E_n$ = The number of incoming and outgoing edges that are connected to a particular cluster's nodes.
$H_n$ = Number of hits from incoming edges to a particular cluster's nodes.
$CM_n$ = Number of commits related each method.
$CM_{max}$ = Maximum number of commits.
$CL_n$ = Summation of criticality level of all code smells in a module.
$CL_{max}$ = Maximum criticality level.
$E_{max}$ = The number of edges of the cluster with the most edges.

$H_{max}$ = The number of hits of the cluster with the most hits from incoming edges.

$$\text{Prioritization factor } P_n = (\beta(\alpha(\frac{E_n}{E_{max}}) + (1-\alpha)(\frac{H_n}{H_{max}})) + (1-\beta)(\frac{CM_n}{CM_{max}}) \times \frac{CL_n}{CL_{max}}$$

Here $\alpha$ and $\beta$ are two special variables and their value can be 0 to 1. The first variable, $\alpha$ will help us to decide which will get priority between $E_n$ and $H_n$. In this equation, $E_n$ represents the strategic importance of a technique obtained through static analysis and the generation of a business process graph. On the other hand, $H_n$ is a measure of a method's runtime impact obtained by dynamic analysis and the generation of a heatmap graph. Both measures are significant, but their relative importance depends on the nature of the applications. For certain projects, runtime performance is more crucial than strategic significance, whereas, for others, the opposite is true. By including the $\alpha$ variable in the equation, we allow developers the freedom to utilize their preferred method. In the performance evaluation stage, we have found that its value should be in the range of 0.4–0.6 to get good prioritization results.

The second variable, $\beta$ is for prioritizing between code usage and code evolution. Depending on the project nature, the value of $\beta$ should be changed. Code usage and also code evolution both can be crucial factor for code smell prioritization. For this reason, developers need to take decision based on the project's state and nature. If the project in rapidly evolution stage then the commit history should get priority than code usage frequency. We haven't included the cost of resolving code smells in this equation because we don't have a standard measuring mechanism for it yet.

Let's clarify it with an example. Consider the following scenario: there are various functions in a large-scale project.On the other hand, when it comes to frequency of usage, the function that has the most connections is connected to 30 distinct functions. The function that has the highest frequency of use is employed 200 times. The function that has the most commits was modified for a total of 20 times, and the highest criticality level function has a criticality level of 200. Now let's assume, we have a function that is connected to 10 different methods, its usage count is 50, and the number of modifications is 5. The code smell severity level of that function is 70. Now we have to make a decision as to which should get a higher priority between strategic importance, usage count and commit history. Let's assume the value of both $\alpha$ and $\beta$ is 0.5 . So the prioritization factor of that function will be $(0.5 \times ((10/30) \times 0.5 + (50/200) \times 0.5) + 0.5 \times (5/20)) \times (70/200) = 0.0948$. The range of the prioritizing factor will be 0 to 1. In this way, we can calculate a priority for each method and sort them in descending order to get the functions with the highest priority.

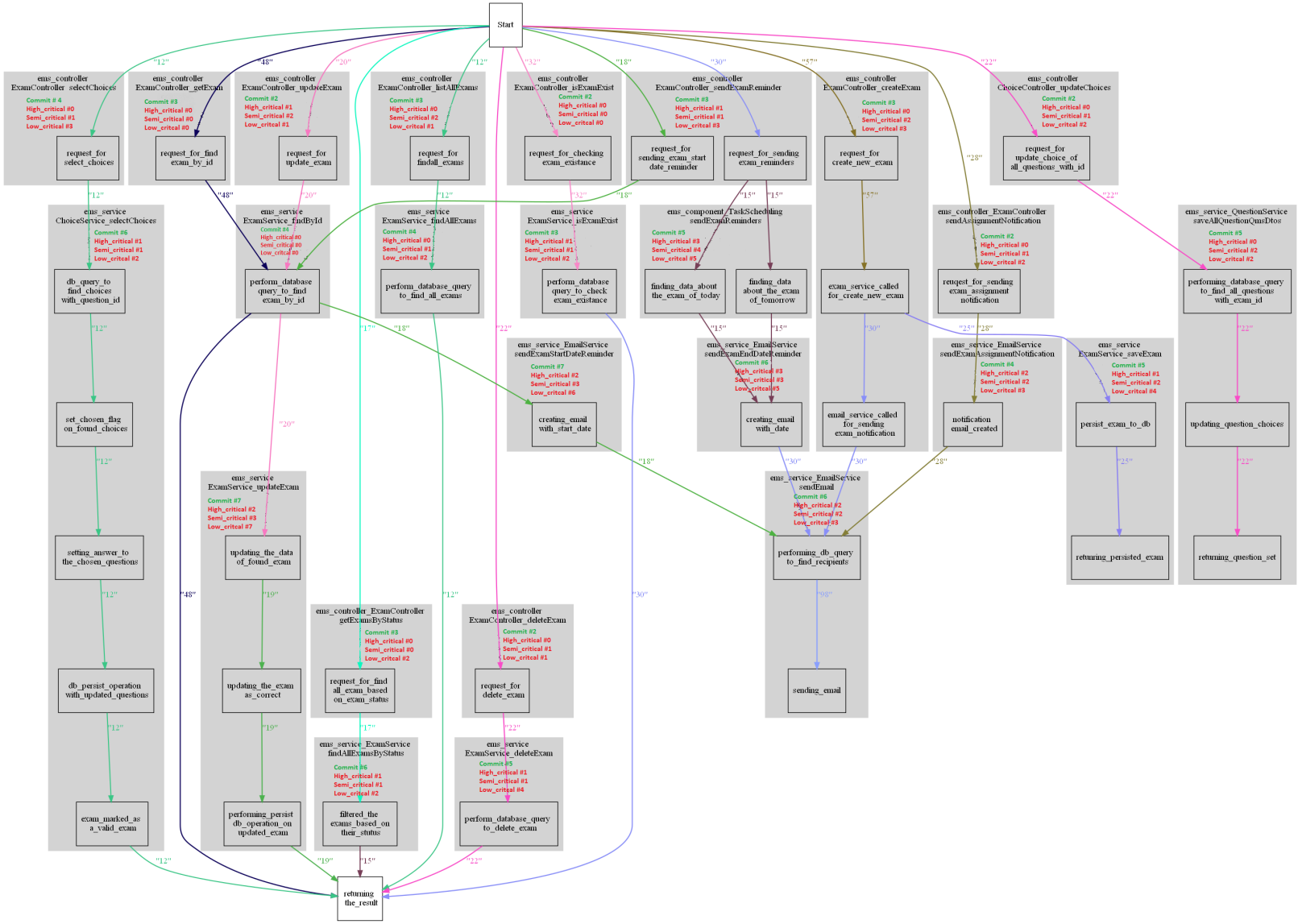


**Figure 4.** Combined graph of business process, code commits, code smells and heat map.

## 5. Case Study Evaluation

To evaluate the performance of our proposed approach, we ran a full-scale case study. In this case study, we tried to select multiple testbeds built upon microservice architecture to make it capable of running on distributed system. At first, we complete our static analysis part which means generating a business process graph by our BPM-SCA tool. Later, we find out code smells by using three tools- SonarQube, Spotbugs, and PMD. After that, we collect application logs from the distributed system by running target testbeds for a certain time duration for performance evaluation.

### *5.1. Testbed-1 (TMS)*

We employed Teacher Management System (TMS): a microservice project, as a testbed for performance evaluation. It is written in Java and uses the SpringBoot framework [49] with the log4j logging mechanism [50]. The repository pattern and service layer pattern are used in this testbed, which includes controllers, services, repositories, and models. All business logic in this design is stored in the service layer. It has eight different microservices, four of them built with pure Java programming language. Another four microservices are client applications built with JavaScript. We only test java files for this evaluation process. See Figure 5 for the testbed architecture.

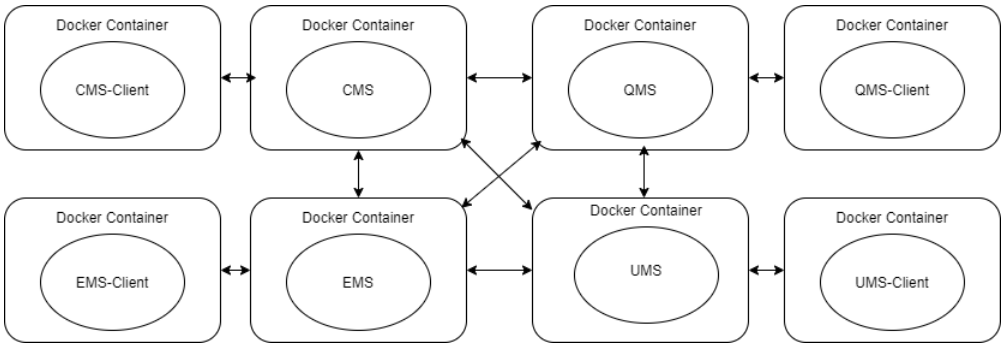


**Figure 5.** Microservice architecture of Teacher Management System (TMS).

To create a distributed test environment, we used docker [51] and Kubernetes [52] for managing docker instances. Eight different docker instances were run in parallel in 8 different kubelets, and they were connected to each other with k-proxy. The control plane of Kubernetes managed the performance and availability of those node instances. After successfully deploying the TMS project on the test environment, we kept it running for several days and asked test users to use the application. We also used the Selenium tool which is automated testing technology for multi-purpose objectives such as heavy load testing, performance stability testing, etc.

### *5.2. Testbed-2 (Train-Ticket)*

We selected to run our application on an existing microservices benchmark, the Train-Ticket Benchmark [53], in order to put it through its tests. As a result, this benchmark is ideal for a microservice application and will test all of our application circumstances. In this benchmark, real-world interaction between microservices in an industrial context was modeled after real-world interaction between microservices in the actual world. Following that, it is one of the most comprehensive microservice benchmarks available. Over 60,000 lines of code are contained within the 41 microservices that comprise this benchmark. In Figure 6 shows the architecture of Train-Ticket project. It deploys using either Docker (https://www.docker.com, accessed on 16 May 2022) or Kubernetes (https://kubernetes.io, accessed on 16 May 2022), and it routes traffic through either NGINX (https://www.nginx.com, accessed on 16 May 2022) or Ingress (https://kubernetes.io/docs/concepts/services-networking/ingress, accessed on 9 June 2022) depending on the environment.

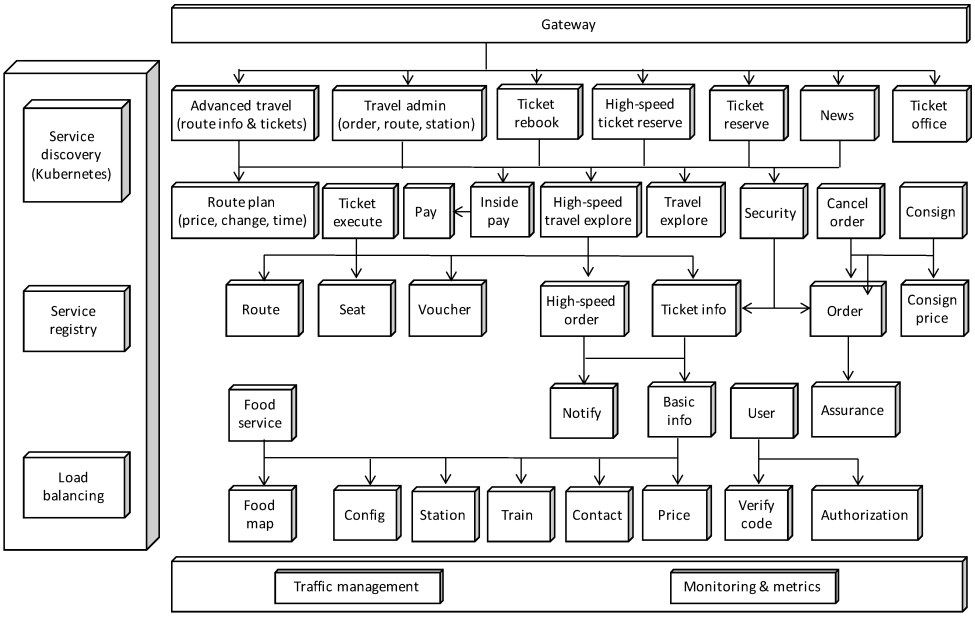


**Figure 6.** Microservice architecture of Train-Ticket project [54].

### *5.3. Using Business Process Mining Tool and Commit History Analyzer*

The proposed BPM-SCA tool is used on the TMS and Train-Ticket projects to generate business process graphs for each projects by static log analysis. We have run the BPM-SCA tool separately for each microservice and as a result, we got four business process graphs for TMS and around 40 for Train-Ticket. We also kept the log statements that we found in the static analysis step in a list for a heatmap generation. Look at Figure 2, it is a business process graph EMS microservice of TMS testbed. On the other hand, the commit history analyzer scans each testbed project's git repository and creates a unique commit history table. Table 2 is the commit history table for EMS microservice of TMS project. Additionally, the data from the commit history table will be integrated with the business process graph.

### *5.4. Using Code Smell Detection Tools*

After finding out the business process, we need to identify all code smells in the both testbeds separately. We have already mentioned that for detecting code smells, we used three popular tools - SonarQube, Spotbugs, and PMD. SonarQube and Spotbugs have a plugin for integrating it with different IDEs such as Eclipse and IntelliJ idea etc along with a standalone server. We have used the plugin for the Intellij idea. We could not find a plugin for PMD and for this reason we had used it as a standalone code smell detecting application. At first, we have to create a list of code smells along with their sources. Source means in microservice name, class name, function name. When the list is complete with the result from three different code smell detecting tools, we need to prune out duplicate results from our list. Since the same code smell can be detected by multiple tools, this step is critical for ensuring that data duplication is avoided. Otherwise, it will affect the performance evaluation accuracy. In Table 3, we can see the result of code smell detection for TMS and in Table 4 the result of code smell detection result is given for Train-Ticket project.

**Table 2.** Commit history table.

| Method ID | Number of Alteration |
|---|---|
| ems_service_EmailService_sendExamEndDateReminder_1 | 6 |
| ems_controller_ChoiceController_updateChoices_1 | 2 |
| ems_service_ExamService_updateExam_2 | 7 |
| ems_controller_ExamController_updateExam_2 | 2 |
| ems_service_ExamService_findAllExams_1 | 4 |
| ems_controller_ExamController_listAllExams_1 | 3 |
| ems_service_ExamService_findById_1 | 4 |
| ems_service_ExamService_findAllExamsByStatus_2 | 6 |
| ems_controller_ExamController_getExamsByStatus_2 | 3 |
| ems_service_ExamService_deleteExam_1 | 5 |
| ems_controller_ExamController_deleteExam_1 | 2 |
| ems_controller_ExamController_getExam_2 | 3 |
| ems_controller_ExamController_sendExamReminder_3 | 3 |
| ems_service_EmailService_sendExamStartDateReminder_3 | 7 |
| ems_component_TaskScheduling_sendExamReminders_3 | 5 |
| ems_controller_ExamController_sendAssignmentNotification_2 | 2 |
| ems_service_EmailService_sendEmail_3 | 7 |
| ems_service_EmailService_sendExamAssignmentNotification_4 | 6 |
| ems_service_ExamService_saveExam_2 | 5 |
| ems_controller_ExamController_createExam_2 | 3 |
| ems_controller_ExamController_selectChoices_3 | 4 |
| ems_service_ChoiceService_selectChoices_2 | 6 |
| ems_service_QuestionService_saveAllQuestionQmsDtos_3 | 5 |
| ems_controller_ExamController_isExamExist_1 | 2 |
| ems_service_ExamService_isExamExist_1 | 3 |

**Table 3.** Code Smell detection result for TMS.

| Tool's Name | Highly-Critical | Semi-Critical | Non-Critical |
|---|---|---|---|
| SonarQube | 87 | 63 | 76 |
| Spotbugs | 79 | 48 | 97 |
| PMD | 47 | 36 | 42 |
| Combined Result | 98 | 79 | 127 |

**Table 4.** Code Smell detection result for Train-Ticket.

| Tool's Name | Highly-Critical | Semi-Critical | Non-Critical |
|---|---|---|---|
| SonarQube | 238 | 563 | 1376 |
| Spotbugs | 179 | 648 | 997 |
| PMD | 124 | 436 | 942 |
| Combined Result | 323 | 879 | 1829 |

### *5.5. Application Run-Time Testing*

To use and test the application in run time we have selected around 20 volunteers, who will use different modules of the Teacher Management Application and give feedback about their experience. We also used the Selenium tool to check the performance of the system against a heavy load. We have done this testing process several times after fixing different code smell to check the performance difference.

### *5.6. Application Log Collection and Heat-Map Graph Creation*

We got the application run-time log from the log files while we were doing the application run-time test. This log will help us to generate a usage frequency table. To generate the usage frequency table, we have used Grok filter tool [46] on log files. Sample log files data of EMS microservice of TMS project is given in Figure 7. For analyzing the log file to generate the usage frequency table, we need to configure the Grok pattern matching configuration file. Grok filter project generates metric files as output which can be turned into a usage frequency table by using Prometheus [47] metric analyzer tool. In Figure 8 we give a sample output generated by Prometheus for the log files data of the EMS microservice.

### *5.7. Generation of Combined graph of Heat-map and Code Smells*

In Sections 4.3 and 4.4, we describe how to relate code smell to business process and heat-map graph. In this step, we are going to follow that technique to generate the combined graph of technical debt and heat-map for the TMS project. In Section 5.3, we have identified existing code smells in our testbed by using three popular tools in the market and stored them accordingly to their sources. In Section 5.6, we have generated the heat-map graph by merging the business process and utilization count of the different modules of the TMS project. In this section, we will merge technical debt analysis results with heat-map graphs to generate a combined graph of Heat-map and code smells for the TMS project. Check Figure 4, which is the combined graph for EMS microservice.

```
30.12.2021 04:33:03 userid=1 message="request for select choices" optional=""
30.12.2021 04:33:03 userid=1 message="db query to find choices with question id" optional="choiceid#3"
30.12.2021 04:33:04 userid=1 message="setting answer to the chosen questions" optional="questionid#14"
30.12.2021 04:33:04 userid=1 message="db persist operation with updated questions" optional="questionid#14"
30.12.2021 04:33:04 userid=1 message="exam marked as a valid exam" optional="questionid#14"
30.12.2021 04:33:04 userid=1 message="persist exam to db" optional="questionid#14"
30.12.2021 04:33:05 userid=1 message="retunring persisted exam" optional="questionid#14"
30.12.2021 04:33:05 userid=1 message="set chosen flag on found choices" optional="1"
30.12.2021 04:33:08 userid=2 message="request for find exam by id" optional="examid#23"
30.12.2021 04:34:12 userid=2 message="request for update exam" optional="examid#23"
30.12.2021 04:34:30 userid=5 message="request for update exam" optional="examid#24"
30.12.2021 04:34:35 userid=2 message="perform database query to find exam by id" optional="examid#23"
30.12.2021 04:34:50 userid=2 message="updating the data of found exam" optional="examid#23"
30.12.2021 04:34:51 userid=2 message="updating the exam as correct" optional="examid#23"
30.12.2021 04:34:51 userid=2 message="performing persist db operation on updated exam" optional="examid#23"
30.12.2021 04:34:55 userid=2 message="returning the result" optional=""
30.12.2021 04:34:59 userid=5 message="request for find exam by id" optional="examid#24"
30.12.2021 04:35:01 userid=3 message="request for find all exam based on exam status" optional="status#1"
30.12.2021 04:35:09 userid=3 message="filtered the exams based on their stutus" optional="status#1"
30.12.2021 04:35:10 userid=4 message="request for findall exams" optional=""
30.12.2021 04:35:10 userid=4 message="perform database query to find all exams" optional=""
30.12.2021 04:34:11 userid=2 message="request for sending exam start date reminder" optional=""
30.12.2021 04:34:12 userid=5 message="request for sending exam reminders" optional=""
30.12.2021 04:34:13 userid=5 message="creating email with start date" optional="05.04.22"
30.12.2021 04:34:13 userid=5 message="performing db query to find recipients" optional="id#15"
30.12.2021 04:34:14 userid=5 message="performing db query to find recipients" optional="id#3"
30.12.2021 04:34:14 userid=5 message="performing db query to find recipients" optional="id#8"
30.12.2021 04:34:15 userid=5 message="performing db query to find recipients" optional="id#5"
30.12.2021 04:34:16 userid=5 message="performing db query to find recipients" optional="id#4"
30.12.2021 04:34:16 userid=5 message="performing db query to find recipients" optional="id#9"
30.12.2021 04:34:20 userid=5 message="sending email" optional="id#15"
30.12.2021 04:34:21 userid=5 message="sending email" optional="id#3"
30.12.2021 04:34:22 userid=5 message="sending email" optional="id#8"
30.12.2021 04:34:22 userid=5 message="sending email" optional="id#5"
30.12.2021 04:34:23 userid=5 message="sending email" optional="id#4"
30.12.2021 04:37:27 userid=3 message="finding data about the exam of today" optional="12.31.21"
30.12.2021 04:37:28 userid=3 message="creating email with date" optional="12.31.21"
30.12.2021 04:37:28 userid=3 message="finding data about the exam of tomorrow" optional="01.01.22"
30.12.2021 04:37:30 userid=3 message="reuqest for sending exam_assignment notification" optional="12.31.21"
30.12.2021 04:38:23 userid=3 message="notification email created" optional="12.31.21"
30.12.2021 04:38:25 userid=1 message="request for create new exam" optional=""
30.12.2021 04:38:29 userid=1 message="exam service called for create new exam" optional=""
30.12.2021 04:38:30 userid=1 message="email service called for sending exam notification" optional="examid#5"
30.12.2021 04:38:35 userid=2 message="request for delete_exam" optional="examid#2"
30.12.2021 04:38:38 userid=2 message="perform database query to delete exam" optional="examid#2"
30.12.2021 04:38:39 userid=7 message="request for update choice of all questions with id" optional="choiceid#2"
30.12.2021 04:38:42 userid=7 message="performing database query to find all questions with exam id" optional="examid#3"
30.12.2021 04:38:43 userid=7 message="updating question choices" optional="choiceid#2"
```

**Figure 7.** Sample log file's data.

### *5.8. Performance Evaluation*

For performance evaluation of code smell prioritization, we have selected JSpIRIT [6] as our comparison basis. Using this prototype, we may sort code smells according to the severity they cause. There are about 250 code smells in TMS project and around 3000 code smells in Train-Ticket project. We have conducted two different studies for measuring the code smell prioritization performance.

5.8.1. Runtime Performance Analysis

It is extremely difficult and time-consuming to remove every code smells. Our goal is to fix 20 code smells for each testbeds and measure the performance gain. At first we started work with TMS project and then repeat the same process for Train-Ticket as well. For starters, we'll collect code smells from all microservices, use both tools to prioritize them, and then fix 20 code smells for each to see if performance has improved. Fixing every code smell may not lead to an increase in run-time performance, as should be noted in this context. Majority of code smells actually increase code reusability, maintainability, and testability issues. Manual testing and feedback from expert developers are required to accurately measure such a feature.

| Element | Value |
|---|---|
| usage_frequency{instance="localhost:9144",job="grok",message="creating email with date}"} | 30 |
| usage_frequency{instance="localhost:9144",job="grok",message="creating email with start date}"} | 18 |
| usage_frequency{instance="localhost:9144",job="grok",message="db persist operation with updated questions}"} | 12 |
| usage_frequency{instance="localhost:9144",job="grok",message="db query to find choices with question id}"} | 12 |
| usage_frequency{instance="localhost:9144",job="grok",message="email service called for sending exam notification}"} | 30 |
| usage_frequency{instance="localhost:9144",job="grok",message="exam marked as a valid exam}"} | 12 |
| usage_frequency{instance="localhost:9144",job="grok",message="exam service called for create new exam}"} | 25 |
| usage_frequency{instance="localhost:9144",job="grok",message="filtered the exams based on their stutus}"} | 16 |
| usage_frequency{instance="localhost:9144",job="grok",message="finding data about the exam of today}"} | 15 |
| usage_frequency{instance="localhost:9144",job="grok",message="finding data about the exam of tomorrow}"} | 19 |
| usage_frequency{instance="localhost:9144",job="grok",message="notification email created}"} | 28 |
| usage_frequency{instance="localhost:9144",job="grok",message="perform database query to check exam existance}"} | 32 |
| usage_frequency{instance="localhost:9144",job="grok",message="perform database query to delete exam}"} | 22 |
| usage_frequency{instance="localhost:9144",job="grok",message="perform database query to find all exams}"} | 12 |
| usage_frequency{instance="localhost:9144",job="grok",message="perform database query to find exam by id}"} | 20 |
| usage_frequency{instance="localhost:9144",job="grok",message="performing database query to find all questions with exam id}"} | 22 |
| usage_frequency{instance="localhost:9144",job="grok",message="performing db query to find recipients}"} | 98 |
| usage_frequency{instance="localhost:9144",job="grok",message="performing persist db operation on updated exam}"} | 19 |
| usage_frequency{instance="localhost:9144",job="grok",message="persist exam to db}"} | 25 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for checking exam existance}"} | 32 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for create new exam}"} | 27 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for delete_exam}"} | 22 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for find all exam based on exam status}"} | 17 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for find exam by id}"} | 48 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for findall exams}"} | 12 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for select choices}"} | 12 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for sending exam reminders}"} | 14 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for sending exam start date reminder}"} | 18 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for update choice of all questions with id}"} | 22 |
| usage_frequency{instance="localhost:9144",job="grok",message="request for update exam}"} | 20 |
| usage_frequency{instance="localhost:9144",job="grok",message="retunring persisted exam}"} | 27 |

**Figure 8.** Grok filter result generated by Prometheus.

The proposed approach will be referred to as CSP (Code Smell Prioritizer). Our testbed has been compared to three different versions. In the first release, there is no code smell remedy. We used the JSpIRIT tool to correct 20 code smells that were deemed critical in the second version. We utilized the CSP tool to prioritize all code smells in the third version, and then solved the 20 that received the highest priority. We now have three versions that can be compared side by side. Each of these three iterations has been put through rigorous testing and analysis to determine how well they function under varied load conditions. See Figure 9, for their performance differences. Smaller requests don't make a big difference, but if we look at a huge amount of requests, the version patched with JSpIRIT and CSP performs a little better than the original one. Our understanding is that code smells generally do not have a significant effect on run-time performance.

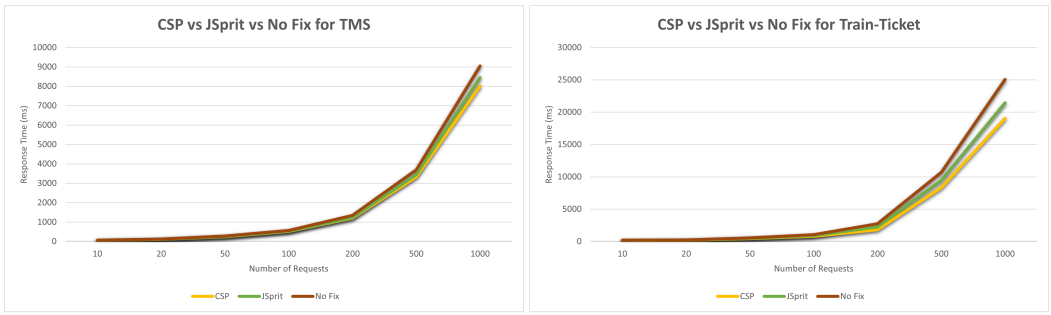


**Figure 9.** Response Time Comparison of CSP, JSpirit, and Original Project with No-Fix for TMS and Train-Ticket.

We had conducted same test for the Train-Ticket as well. The result was almost similar to Train-Ticket also. In Figure 9, we shows runtime comparison for train-ticket project.

5.8.2. Developers Oriented Maintainability Testing

Because there is no direct way to evaluate the maintainability or reusability of a codebase, we have done a specific type of testing with the assistance of developers who are directly involved in the development of the testbed projects. We asked them to choose couple of code sections based on their experience where the presence of code smell will have the most impact on the overall maintainability and testability of the project. After that, the developers were asked to confidentially add selected code smells to the source code to deteriorate it, without specifying these smells and locations and origins. Consequently, we used our proposed tool to generate the ranking of all code smells that were identified in the testbed project. Next, we forwarded the ranking results to individual developers, and instructed them to determine how many of the newly added code smells appeared in the top 5 percent, 10 percent, 20 percent, 30 percent, and 50 percent of the rankings. We also ran this check using JSpIRIT to see how well it performed in order to compare it. Finally, we polled the developers to find out which outcome they were the most pleased with. In Figure 10 we showed the developers oriented maintainability test result for TMS project. Using this graph, the horizontal axis represents the independent variable, which is the percentage of top-ranked code smells, and the vertical axis indicates the proportion of developer-added code smells in the top-ranked code smells. For example, if we take the top 20 percent of code smells from both the CSP and JsPIRIT priority ranking lists, we find that 48 percent of them are developer-added in the result of the CSP ranking and 25 percent are developer-added in the result of the JsPIRIT ranking. Figure 10 shows the result of similar analysis on the train-ticket project as well.

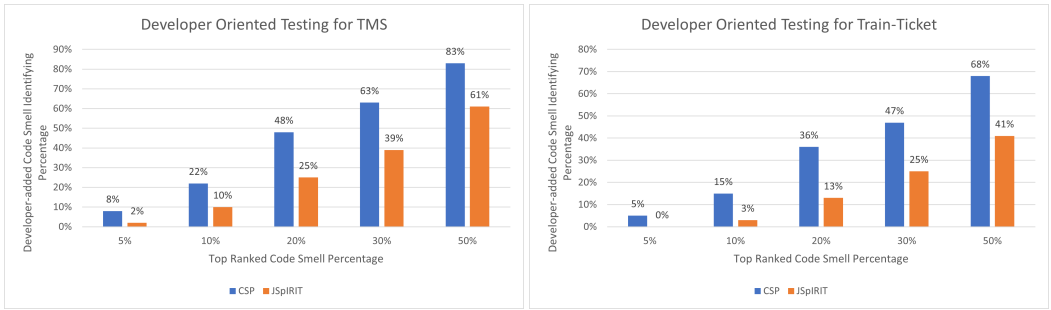


**Figure 10.** Developers Oriented Maintainability Comparison Between Code Smell Prioritizer (CSP) and JsPIRIT for TMS and Train-Ticket.

*5.9. Threats to Validity*

Multiple validity threats were discovered in this study, including the accuracy of the business process extracting tool, the cost of resolving code smells, and the improvement assessment after fixing code smells, among others. Since there is no standard resource on

the cost of fixing code smells, we are presuming that the cost of fixing all code smells is the same at this time. However, we are aware that this is incorrect. We aim to be able to improve our work once there is a uniform cost scale for addressing code smells based on their type. To measure the performance improvement after fixing code smells, we have used runtime performance comparison and developer oriented tests for maintainability, reusabilitiy and testability. We are aware, however, that this is not a conventional way for evaluating these benefits, and a standard evaluation strategy has not yet been developed. Below we will also address the internal and external threats to validity.

#### 5.9.1. Internal Validity

We ran our application ten times to validate that the times we recorded in Tables 3 and 4 for our system's running time avoid an unusual system deviation. Our solution is put to the test against both automated and manually collected data. To reduce the risk of errors when collecting the data, we had different researchers collect the data and compare it. These results were utilized to verify that our system was working properly.

#### 5.9.2. External Validity

To make our test as practical as feasible, we used open-source benchmark programs as a testbed that were similar to real-world settings. Our research is based on industry best practices. As a result, our system can analyze apps that follow best practice criteria.

The case study's benchmark application is mostly written in Java. We employed static code analysis to extract business process logic in this study. We used the Java parser to analyze Java source code for static analysis. Other current languages offer similar parsing capabilities; for example, Python and Golang contain built-in parser packages for obtaining AST from source code. Our tool may be modified to support a wide range of languages as long as they have an adequate parser.

## 6. Conclusions and Future Work

In this paper, we have presented a novel approach to prioritize code smells to achieve the best performance improvement using the limited available resources. When other related works tried to rank code smells, they usually only looked at the type of code smell. Even though it is important, it is not the only thing we should consider about when setting priorities. Our proposed solution considers how the code is utilized as well as how it has been changed in the past, as well as code smell categories.

In the section on performance evaluation, we compare our proposed approach with a similar kind of work and show the performance gain measure based on runtime and developer-oriented testing. During the tests, we noticed that code smells don't have much of an effect on runtime performance, especially for Java-based projects. Because of this, we didn't notice a big difference in performance between the original version and the version with code smells fixed. However, we detected a considerable performance difference between the suggested approach and related work during the developer-oriented testing phase. This developer-oriented testing was designed to assess the project's maintainability, testability, and reusability from the perspective of developers. Our proposed approach shows remarkable performance in this testing phase.

This novel approach has many factors that are not fully discovered in this research. For those factors, we have used some anticipated values, but there is not any valid proof that they are correct.

The first important factor is the cost of fixing a specific kind of code smell. To fix a code smell, we need to utilize different kinds of resources, such as manpower, time, money, etc. But there is no certain catalog for the cost to repair based on their types. In this work, we are just assuming that critical code smell's cost to repair is higher than a non-critical code smell's. However, there is no guarantee that this assumption will be correct in the real world.

The second important factor is that there is no standard tool to measure code maintainability, reusability, and testability. But we know that code smells do not only affect the run-time performance of a project, but also affect these features of the project's code. So in our performance evaluation, we could not quantify how much improvement our tool can offer.

These are some research areas that need additional work to make this novel approach more accurate in prioritizing code smells. In addition, we have tested our tool on two testbed projects with few micro-services, for simplicity. But in the real world, we typically see very big projects running on the server which are dependent on a huge number of micro-services. For better calibration, we need to use our tools on this kind of project also.

Moreover, we have plans to ask some large research groups or companies to use this tool and give us feedback, since we do not have any concrete measuring scale for code maintainability, reusability, and testability. Additionally, in this work, we discovered an intriguing way to quantify the importance of a code segment within a large codebase. We intend to use this information in a variety of ways such as resource distribution, program performance improvement, and more.

**Author Contributions:** Conceptualization, M.R.I.; Formal analysis, M.R.I. and A.A.M.; Funding acquisition, T.C.; Methodology, M.R.I.; Project administration, T.C.; Software, M.R.I.; Validation, A.A.M. and T.C.; Visualization, M.R.I.; Writing—original draft, M.R.I.; Writing—review & editing, A.A.M. and T.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** NSF OISE-1854049, Red Hat Research.

**Acknowledgments:** This research was funded by National Science Foundation grant number 1854049 and a grant from Red Hat Research https://research.redhat.com (accessed on 9 June 2022).

**Conflicts of Interest:** The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

## References

1. Gupta, A.; Suri, B.; Kumar, V.; Misra, S.; Blažauskas, T.; Damaševičius, R. Software code smell prediction model using Shannon, Rényi and Tsallis entropies. *Entropy* **2018**, *20*, 372. [CrossRef]
2. Fontana, F.A.; Ferme, V.; Spinelli, S. Investigating the impact of code smells debt on quality code evaluation. In Proceedings of the 2012 Third International Workshop on Managing Technical Debt (MTD), Zurich, Switzerland, 5 June 2012; pp. 15–22. [CrossRef]
3. Arogundade, O.T.; Onilede, O.; Misra, S.; Abayomi-Alli, O.; Odusami, M.; Oluranti, J. From Modeling to Code Generation: An Enhanced and Integrated Approach. In *Innovations in Information and Communication Technologies (IICT-2020)*; Springer: Cham, Switzerland, 2021; pp. 421–427.
4. Baabad, A.; Zulzalil, H.B.; Hassan, S.; Baharom, S.B. Software Architecture Degradation in Open Source Software: A Systematic Literature Review. *IEEE Access* **2020**, *8*, 173681–173709. [CrossRef]
5. Gupta, A.; Suri, B.; Misra, S. A systematic literature review: Code bad smells in java source code. In *International Conference on Computational Science and Its Applications*; Springer: Cham, Switzerland, 2017; pp. 665–682.
6. Vidal, S.A.; Marcos, C.; Díaz-Pace, J.A. An approach to prioritize code smells for refactoring. *Autom. Softw. Eng.* **2016**, *23*, 501–532. [CrossRef]
7. Campbell, G.A.; Papapetrou, P.P. *SonarQube in Action*; Manning Publications Co.: Shelter Island, NY, USA, 2013.
8. SpotBugs Manual—Spotbugs 4.5.3 Documentation. Available online: https://spotbugs.readthedocs.io/en/latest/ (accessed on 16 November 2021).
9. Copeland, T. *PMD Applied*; Centennial Books: Arexandria, VA, USA, 2005; Volume 10.
10. Misra, S. A step by step guide for choosing project topics and writing research papers in ICT related disciplines. In *International Conference on Information and Communication Technology and Applications*; Springer: Cham, Switzerland, 2020; pp. 727–744.
11. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2018.
12. Fontana, F.A.; Zanoni, M. On Investigating Code Smells Correlations. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011; pp. 474–475.
13. Roveda, R.; Arcelli Fontana, F.; Pigazzini, I.; Zanoni, M. Towards an Architectural Debt Index. In Proceedings of the 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Prague, Czech Republic, 29–31 August 2018 ; pp. 408–416. [CrossRef]

14. Kaur, A.; Jain, S.; Goel, S.; Dhiman, G. Prioritization of code smells in object-oriented software: A review. *Mater. Today Proc.* **2021**. [CrossRef]
15. Verma, R.; Kumar, K.; Verma, H.K. A Study of Relevant Parameters Influencing Code Smell Prioritization in Object-Oriented Software Systems. In Proceedings of the 2021 6th International Conference on Signal Processing, Computing and Control (ISPCC), Solan, India, 7–9 October 2021; pp. 150–154. [CrossRef]
16. Fontana, F.A.; Ferme, V.; Zanoni, M.; Roveda, R. Towards a prioritization of code debt: A code smell Intensity Index. In Proceedings of the 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), Bremen, Germany, 2 October 2015; pp. 16–24. [CrossRef]
17. Pecorelli, F.; Palomba, F.; Khomh, F.; De Lucia, A. Developer-driven code smell prioritization. In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Korea, 29–30 June 2020; pp. 220–231.
18. Gupta, H.; Misra, S.; Kumar, L.; Murthy, N. An Empirical Study to Investigate Data Sampling Techniques for Improving Code-Smell Prediction Using Imbalanced Data. In *International Conference on Information and Communication Technology and Applications*; Springer: Cham, Switzerland, 2020; pp. 220–233.
19. Gupta, A.; Chauhan, N.K. A severity-based classification assessment of code smells in Kotlin and Java application. *Arab. J. Sci. Eng.* **2022**, *47*, 1831–1848. [CrossRef]
20. Vidal, S.; Vazquez, H.; Diaz-Pace, J.A.; Marcos, C.; Garcia, A.; Oizumi, W. JSpIRIT: A flexible tool for the analysis of code smells. In Proceedings of the 2015 34th International Conference of the Chilean Computer Science Society (SCCC), Santiago, Chile, 9–13 November 2015; pp. 1–6. [CrossRef]
21. Singh, R.; Bindal, A.; Kumar, A. 3 Software Engineering Paradigm for Real-Time Accurate Decision Making for Code Smell Prioritization. In *Data Science and Innovations for Intelligent Systems: Computational Excellence and Society 5.0*; CRC Press: Boca Raton, FL, USA, 2021; p. 67. [CrossRef]
22. Steidl, D.; Eder, S. Prioritizing maintainability defects based on refactoring recommendations. In Proceedings of the 22nd International Conference on Program Comprehension, New York, NY, USA, 2–3 June 2014; pp. 168–176.
23. Deissenboeck, F.; Heinemann, L.; Hummel, B.; Juergens, E. Flexible architecture conformance assessment with ConQAT. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 2–8 May 2010; Volume 2, pp. 247–250.
24. Moha, N.; Gueheneuc, Y.G.; Duchien, L.; Le Meur, A.F. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.* **2010**, *36*, 20–36. [CrossRef]
25. Palomba, F.; Bavota, G.; Penta, M.D.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A. Mining Version Histories for Detecting Code Smells. *IEEE Trans. Softw. Eng.* **2015**, *41*, 462–489. [CrossRef]
26. Gomes, I.; Morgado, P.; Gomes, T.; Moreira, R. An overview on the static code analysis approach in software development. Faculdade de Engenharia da Universidade do Porto, Portugal. 2009. Available online: https://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf (accessed on 16 May 2022).
27. Novak, J.; Krajnc, A.; Žontar, R. Taxonomy of static code analysis tools. In Proceedings of the 33rd International Convention MIPRO, Opatija, Croatia, 24–28 May 2010; pp. 418–422.
28. TOP 40 Static Code Analysis Tools (Best Source Code Analysis Tools). Softwaretestinghelp. 2021. Available online: https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/ (accessed on 9 June 2022).
29. Kumar, K.S.; Malathi, D. A novel method to find time complexity of an algorithm by using control flow graph. In Proceedings of the 2017 International Conference on Technical Advancements in Computers and Communications (ICTACC), Melmaurvathur, India, 10–11 April 2017; pp. 66–68.
30. Ribeiro, J.C.B.; Zenha-Rela, M.A.; Fernandéz de Vega, F. Using dynamic analysis of java bytecode for evolutionary object-oriented unit testing. In Proceedings of the 8th Workshop on Testing and Fault Tolerance, Beijing, China, 12–13 October 2007; pp. 143–156. Available online:https://iconline.ipleiria.pt/handle/10400.8/134 (accessed on 16 May 2022).
31. Syaikhuddin, M.M.; Anam, C.; Rinaldi, A.R.; Conoras, M.E.B. Conventional software testing using white box method. In *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*; 2018; pp. 65–72. Available online: http://download.garuda.kemdikbud.go.id/article.php?article=1620790&val=11237&title=Conventional%20Software%20Testing%20Using%20White%20Box%20Method (accessed on 9 June 2022).
32. Roy, C.K.; Cordy, J.R.; Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* **2009**, *74*, 470–495. [CrossRef]
33. Selim, G.M.; Foo, K.C.; Zou, Y. Enhancing source-based clone detection using intermediate representation. In Proceedings of the 2010 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010; pp. 227–236.
34. Kidd, C. Tracing vs Logging vs Monitoring: What's the Difference? 2019. Available online: https://www.bmc.com/blogs/monitoring-logging-tracing (accessed on 12 January 2022).
35. Chakraborty, M.; Kundan, A.P. Architecture of a Modern Monitoring System. In *Monitoring Cloud-Native Applications*; Springer: Berkeley, CA, USA, 2021; pp. 55–96.
36. Aljawabrah, N.; Gergely, T.; Misra, S.; Fernandez-Sanz, L. Automated Recovery and Visualization of Test-to-Code Traceability (TCT) Links: An Evaluation. *IEEE Access* **2021**, *9*, 40111–40123. [CrossRef]
37. Van Der Aalst, W. Process Mining. *Commun. ACM* **2012**, *55*, 76–83. [CrossRef]

38. van der Aalst, W.; Reijers, H.; Weijters, A.; van Dongen, B.; Alves de Medeiros, A.; Song, M.; Verbeek, H. Business process mining: An industrial application. *Inf. Syst.* **2007**, *32*, 713–732. [CrossRef]
39. Process Mining-What Is Process Mining? | Appian. Available online: https://appian.com/bpm/what-is-process-mining.html (accessed on 20 November 2021).
40. Behnamghader, P.; Alfayez, R.; Srisopha, K.; Boehm, B. Towards Better Understanding of Software Quality Evolution through Commit-Impact Analysis. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 2017; pp. 251–262. [CrossRef]
41. Zanjani, M.B.; Swartzendruber, G.; Kagdi, H. Impact analysis of change requests on source code based on interaction and commit histories. In Proceedings of the 11th Working Conference on Mining Software Repositories, New York, NY, USA, 31 May–1 June 2014; pp. 162–171.
42. VoidVisitorAdapter-Javaparser-Core 3.3.1 Javadoc. Available online: https://javadoc.io/doc/com.github.javaparser/javaparser-core/3.3.1/com/github/javaparser/ast/visitor/VoidVisitorAdapter.html (accessed on 10 January 2022).
43. Tarjan, R. Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1972**, *1*, 146–160. [CrossRef]
44. Pang, C.; Wang, J.; Cheng, Y.; Zhang, H.; Li, T. Topological sorts on DAGs. *Inf. Process. Lett.* **2015**, *115*, 298–301. [CrossRef]
45. Cinque, M.; Della Corte, R.; Pecchia, A. Microservices monitoring with event logs and black box execution tracing. *IEEE Trans. Serv. Comput.* **2019**, *15*, 294–307. [CrossRef]
46. Grok Exporter. 2020. Available online: https://github.com/fstab/grok_exporter (accessed on 16 January 2022).
47. Prometheus. Prometheus-Monitoring System & Time Series Database. Available online: https://prometheus.io/ (accessed on 17 January 2022).
48. Graphviz. Graphviz. Graph Visualization Software. Available online: https://graphviz.org/ (accessed on 12 December 2021).
49. Webb, P.; Syer, D.; Long, J.; Nicoll, S.; Winch, R.; Wilkinson, A.; Overdijk, M.; Dupuis, C.; Deleuze, S. Spring boot reference guide. In *Part IV. Spring Boot Features*; 2013; Volume 24. Available online: https://www.baidasteel.com/spring-boot/docs/2.1.8.BUILD-SNAPSHOT/reference/pdf/spring-boot-reference.pdf (accessed on 9 June 2022).
50. Gupta, S. Log4j and J2EE. In *Pro Apache Log4j*; A-Press: Berkeley, CA, USA, 2005; pp. 157–161.
51. Anderson, C. Docker [software engineering]. *IEEE Softw.* **2015**, *32*, 102-c3. [CrossRef]
52. Vohra, D. *Kubernetes Microservices with Docker*; Apress: New York, NY, USA, 2016.
53. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Xu, C.; Ji, C.; Zhao, W. Poster: Benchmarking microservice systems for software engineering research. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May–3 June 2018; pp. 323–324.
54. Walker, A.; Das, D.; Cerny, T. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Appl. Sci.* **2020**, *10*, 7800. [CrossRef]