Thesis for The Degree of Licentiate of Computer Science and Engineering

Predicting Workload Dependant and Independent Performance of Software Systems Using Machine Learning Based Approaches

Data-Centric AI For Performance Engineering

Peter Samoaa

Division of Data Science and AI
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2023

Predicting Workload Dependant and Independent Performance of Software Systems Using Machine Learning-Based Approaches
Data-Centric AI For Performance Engineering

Peter Samoaa

Department of Computer Science and Engineering
Division of Data Science and AI
Division of Interaction design and Software Engineering
ICET LAB
Chalmers University of Technology
SE-412 58 Göteborg,
Sweden
mail: samoaa@chalmers.se

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2023.

I want to dedicate this to my countries, England and Italy, where I always feel at home. These countries gave me the right to live. I will always be grateful for that.
I also want to thank the British Royal family institute, which supported me for 15 years through my school studies.
I would dedicate this to the memory of our gracious queen, her majesty queen Elizabeth II. There can simply be no finer example of dignified public duty and unstinting service. We all owe our sincere gratitude for her continued devotion, living every day by the pledge she made on her 21st birthday. Her dedication to our country has been incomparable, and, as such, she leaves an enduring legacy.
Special appreciation goes to my early line manager, Ivica Crnkovic, for the kindness, inspiration, unlimited support, and for being like a father, I will always remember you.
My parents, sisters, and brother, I am your permanent ambassador to the future and your hope that you will never be disappointed.

Abstract

Context: Machine learning (ML) approaches are widely employed in various software engineering (SE) tasks. Performance, however, is one of the most critical software quality requirements. Performance prediction is estimating the execution time of a software system prior to execution. The backbone of performance estimation is prediction models, in which machine learning (ML) is a common choice. Two settings are commonly considered for ML-based performance prediction: workload-dependent and workload-independent performance, depending on whether or not the specific usage of the system is fed as input to the ML estimator.

Problem: Developers usually understand the behaviour of performance with respect to the workload manually. This process consumes time, effort, and computational resources since the developer repeats the running of the same tested system (e.g. benchmark) many times, each with different workload values. In a workload-independent setting, predicting the scalar value of execution time based on the structure of the source code is challenging as it is a function of many factors, including the underlying architecture, the input parameters, and the application's interactions with the operating system. Consequently, works that have attempted to predict absolute execution time for arbitrary applications from source code generally report poor accuracy.

Goal: The thesis aims to present a modern machine learning-based approach for predicting the execution time from two angles: (a) workload-independent performance. (b) workload-dependent performance.

Solution Approaches and Research Methodologies: To achieve the goal and tackle the problems mentioned earlier, we conducted a systematic empirical study to fill the gap of workload-dependant performance across five well-known projects in JMH benchmarking (including RxJava, Log4J2, and the Eclipse Collections framework) and 126 concrete benchmarks. We generated a dataset of approximately 1.4 million measurements. As for the poor accuracy challenges, we aim to increase the quality of data which is the source code in this context. To that aim, we invest in Data-Centric AI. Thus, we conduct a systematic literature review and systematic mapping study about the different approaches of source code representation and the level of information each representation can hold. Then, based on that, we conduct an experimental study to increase the quality of source code representation by establishing a rich hybrid code representation Then marry this representation with a Graph Neural Network (GNN)- an ML approach to predict the scalar value of the functional test.

Results: Our results showed that by investing in classical ML approaches, we could predict the performance value of the benchmarks according to configuration workload. Moreover, with our proposed method, the developers can easily determine the impact of each workload on the performance measurement. On the other hand, by increasing the data quality through data-centric AI, we achieve very high and considerable accuracy in predicting the absolute execution time of software performance only according to the structure of the source code.

List of Publications

Appended publications

This thesis is based on the following publications:

[**Paper I**] **Peter Samoaa** and Philipp Leitner, *An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects*
Proceedings of the ACM/SPEC International Conference on Performance Engineering ICPE'21 (April 2021), 213–224.

[**Paper II**] **Peter Samoaa**, Firas Bayram, Pasquale Salza, and Philipp Leitner, *A systematic mapping study of source code representation for deep learning in software engineering*
IET Software Journal, 2022, 351-385.

[**Paper III**] **Peter Samoaa**, Antonio Longa, Mazen Mohamad, Morteza Haghir Chehreghani, and Philipp Leitner, *TEP-GNN: Accurate Execution Time Prediction of Functional Tests using Graph Neural Networks*
PROFES'22, the International Conference on Product-Focused Software Process Improvement (November 2022).

[**Paper IV**] **Peter Samoaa** and Barbara Catania , *A Pipeline for Measuring Brand Loyalty Through Social Media Mining*
SOFSEM 2021: Theory and Practice of Computer Science. (January 2022).

Research Contribution

I (Peter Samoaa) was the main driver and contributor of the all papers. A summary of the contributions is presented in Table 1.

Role	**Paper I**	**Paper II**	**Paper III**	**Paper IV**
Conceptualization	X	X	X	X
Data curation		X		X
Formal Analysis	X	X	X	X
Investigation	X	X	X	X
Methodology		X	X	X
Software				X
Validation	X	X	X	X
Visualization	X	X	X	X
Writing - original draft	X	X	X	X

Table 1: The Individual Contributions of this thesis' author to the appended papers [1].

Acknowledgment

First and foremost, I would like to thank my supervisors, Philipp Leitner and Morteza Haghir Chehreghani, for the mentorship and support that they have given me. Your input and insights have been invaluable in shaping the research of this thesis. More importantly, your coaching helped develop my research acumen, and I will be forever grateful for that. Looking forward to continuing our research journey.

Next, I would like to thank my examiner, Prof. Miroslaw Staron, for the constructive feedback. I would also like to express my appreciation to the discussion leader of this licentiate, Prof. Görel Hedin, for accepting the invitation to discuss my research.

To my soul mate and fountain of cherish and support, Walaa, From the bottom of my heart, thank you for being the brightest part of my life.

Mazen & Reem, thank you for embracing me from the first day of my arrival in Sweden. You let me feel that I have a lovely family here in Sweden.

My acknowledgement and gratitude also go to my bro Khaled for all the funny and crazy special moments as well the food adventures. Your gentleness made a life here less difficult.

Special thanks to the Data Science and AI division for the supportive and inspirational environment. Special thanks also to my office mates Firooz and Mehrdad and my friend Arman for the friendly and colourful working atmosphere.

To Firas, we shared not only unforgettable memories in Italy but the same inspirational models and the same passion for AI. Thanks for helping me out through my SLR.

Big thanks to Antonio for the joyful collaborations. Looking forward to continuing our collaboration.

To the ICET-LAB research group members, Christopher, Linda, Joel and Hamdy, Thanks for the joyful discussion and help.

This work received financial support from the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineering).

Contents

Chapter 1

Introduction

Machine Learning (ML) approaches have been widely investigated in software engineering (SE). ML-based approaches deliver intelligent solutions for many challenging problems in SE, such as defect detection [2]–[4], automated program repair [5], and predicting the performance of software system before it executes [6]. The performance prediction problem is getting more attention in research due to the different dimensions of the complexity of the performance problem. Examples of such complexity are variability and uncertainty in the performance measures and the large space of workload configuration options in the large-scale workload-dependant systems [7]. These problems are complex because performance measures are a function of many factors at the hardware level, for example, computational resources (used memory, CPUs) and networks, especially in distributed or cloud-based systems. Therefore, predicting the performance of software systems has attracted the attention of software engineering researchers. Numerous research studies aim to develop novel methods for estimating the performance of software systems from a limited number of measured data [8]. A performance prediction model helps developers monitor the source code changes' impact on the performance in the workload-independent setting. In addition, it helps developers in testing the different configurations and their correlations with performance in the workload-dependent setting. This motivates analysing software systems' performance at the earlier life cycle phases by reasoning on quantitative predictive results to avoid expensive post-deployment rework.

1.1 Background

Software Performance Engineering (SPE) is the discipline that represents the entire collection of engineering activities used throughout the software development cycle and directed to meet performance requirements (such as throughput, latency, or memory usage) [9].

According to Woodside et al. [10], the elements of the SPE domain are

- System operations (Use Cases) with performance requirements, behaviour defined by scenarios (e.g., by UML behaviour specifications) and workloads (defining the frequency of initiation of different system operations).

- System structure, the software components, resources, hardware and software.

Besides these core elements of SPE, there are many objectives that drive performance prediction, as listed in the following examples.

- Grow business revenue by guaranteeing the system can process transactions within the requisite timeframe (see the Example Scenario 2: SE Product Performance below).
- Eliminate late system deployment, avoidable system rework, and avoidable system tuning efforts due to performance issues. Moreover, avoid additional and unnecessary hardware acquisition costs. In addition, predicting the performance reduces increased software maintenance costs due to performance problems in production and software impacted by ad hoc performance fixes. Predicting the performance helps reduce the additional operational overhead for handling system issues due to performance problems and identify potential future bottlenecks by simulation over the prototype.

These objectives will be justified in the following example scenarios.

Scenario 1: Self-driving vehicles: Imagine a self-driving vehicle where many sensors collect data from the car's surrounding environment. This data is the input of the online algorithms encapsulated in the back-end system. The online algorithm then decides the car's behaviour based on the sensor data. Suppose there is a delay in the data exchange between the sensor, back-end system, and the online algorithm. In that case, this latency will delay the vehicle's reaction, which might lead to a disaster if pedestrians cross the street and the vehicle keeps driving.

Scenario 2: Product performance in social business intelligence: In online systems such as social business intelligence (SBI, the process of combining the corporate data with user-generated content, or UGC, to make decision-makers aware of important brand-related trends, and improve decision making through timely feedback [11], see also **Paper IV**) we eventually want to deliver an online dashboard for the decision makers about different indicators related to a studied brand. The result must be updated and presented in an online manner so the decision maker can take a tactical decision accordingly. However, if there is a delay in the online dashboard because of the complexity of processing the collected data, training many ML models and then computing the KPIs, that will lead to wrong decisions taken by the decision-makers due to wrong indicators on the dashboard.

Scenario 3: E-Commerce business: Web applications are ubiquitous scenarios where performance is essential, and optimizing the performance is crucial. A slowdown in any E-Commerce system (e.g., online shopping) has a direct business impact [12].

Scenario 4: Middle-ware libraries: Suppose we are building a programming library (e.g., log4j[1]), and many systems use this library. In that case, the slowdown for log4j negatively impacts the entire ecosystem, not only the library itself.

In this thesis, we are looking at performance engineering from two different perspectives, while we are working on source code and while we are optimizing an application for different workloads [13]. Thus, if the developers need to change the code due to changes in customer requirements or to add a new service, then it is hard to express the impact of these changes on the execution time. Hence having a predictive performance model is crucial to predict the performance of the added code and check the effect of the added code on the performance metrics. On the other hand, testing the system with different parameters is important since most modern software systems can be customized by means of workload options to enable desired functionality or tweak non-functional aspects, such as performance or energy consumption. The relationship of workload choices and their influence on performance has been extensively studied in the literature [14], [15]. However, developers usually manually test the different values of the workloads and their contribution to the benchmark's performance measurement. The problem with manual testing and adjustment is that it takes significant time, effort, and computational resources, as executing some benchmarks might take hours. That is why having a predictive performance model that predicts the execution time based on workload parameter values will save substantial time and effort.

1.1.1 Types of Performance Considered in This Thesis

Performance testing is an umbrella term used for a wide variety of different approaches. In this thesis, we are studying different types of performance: (i) workload-dependant performance, where we assume that the software system is fixed and we vary the workload, (ii) workload-independent performance, where we keep the workload fixed and vary the software system (source code), and (iii) eventually the overall performance of software product.

1.1.1.1 Workload Dependant Performance

The performance that we obtain in this context is based on how the software is used. Thus, we invest in micro benchmarking to test how the system performs under different workloads, i.e., short-running benchmarks aiming to measure fine-grained performance metrics, such as method-level execution times, throughput, or heap utilization. Different from application benchmarks,

[1]https://logging.apache.org/log4j/2.x/

system tests, or load tests, the goal of micro benchmarking is not necessarily to enact the system under realistic, production-like conditions. Instead, microbenchmarks are often written specifically to test sensitive code elements for extreme conditions or to compare multiple implementation variants of the same feature (e.g., different data structures or external libraries).

By now, micro benchmarking frameworks are available for a wide range of programming languages, including C++, Java, and Go. We specifically explore the Java Microbenchmark Harness[2] (JMH). JMH is part of the OpenJDK ecosystem and allows users to specify benchmarks through Java annotations in a syntax clearly inspired by common unit testing frameworks such as JUnit. Laaber et al. [16] have found in 2020 that 753 significant open-source projects on GitHub are actively using JMH, showing that the framework is widely accepted in practice.

```java

@OutputTimeUnit(TimeUnit.NANOSECONDS)
@BenchmarkMode({ Mode.AverageTime })
@Warmup(iterations = 3, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS
    )
public class SetOps {

    @Param("1024")
    int size;
    @Param("512")
    int occupancy;
    @Param("2048")
    int keyBound;
    @Param({ "java.util.HashSet",
        "org.jctools.sets.OpenHashSet" })
    String type;
    private Set<Key> set;
    private Key key;

    @Setup(Level.Trial)
    public void prepare() throws Exception {
        // set up test data according to config / params
    }

    @Benchmark
    public boolean add() {
        return set.add(key);
    }

    @Benchmark
    public boolean remove() {...}

    @Benchmark
    public boolean contains() {..}

    @Benchmark
    public int sum() {...}

```

[2]https://openjdk.java.net/projects/code-tools/jmh/

```
}
```

Listing 1.1: Example of a JMH benchmark (from the `JCTools` project)

An example of a real-life JMH benchmark class from the `JCTools` project is provided in Listing 1.1. The Java class `SetOps` defines four benchmarks, implemented in four methods carrying the `@Benchmark` annotation. In addition, four user-specified parameters are defined using `@Param`. The method annotated with `@Setup` is executed once at the beginning of the process to set up the necessary test data. In the remainder of this work, we refer to `SetOps` as the benchmark class and `SetOps.add`, `SetOps.remove`, and so on as the concrete benchmarks in this class.

The execution model of JMH is straightforward. For each benchmark, the framework generates the cartesian product of all combinations of parameter values and instantiates the benchmark once for each possible combination of parameters. Each of these benchmark instances will then be executed $n + m$ times, where n is the number of warmup iterations (executions where the measurement result is discarded), and m is the number of measurement iterations. Each iteration consists of the framework executing the benchmark method as often as possible in a loop until a configured timeout (commonly 1 or 10 seconds) is reached. The framework then records the measurements produced in the iteration (e.g., how often the method could be invoked or, as is the case of the example, the average method execution time in nanoseconds).

In the mentioned example, since `size`, `occupancy`, and `keybound` all only have a single value, only two distinct combinations of parameters need to be instantiated and executed by JMH. However, in the general (and more common) case where each parameter has multiple values, the presence of parameters leads to an explosion of the number of combinations that need to be run (and hence of the total time required to execute the benchmark suite of the project). This means that the innocent-looking action of adding a new parameter to a benchmark class with two values already doubles the time required for executing all benchmarks in this class – even if this new parameter is potentially not critical to the measurement result or does not provide many new insights. It is, therefore, not surprising that the benchmark suites of projects that routinely use parameters with five or more different values (e.g., `eclipse-collections`) are very time-consuming, often taking multiple days to run in their standard configuration.

Given how crucial it is to choose parameters strategically and parameter values to keep the execution time of benchmark suites manageable, despite the importance of parameters, they have not yet been explicitly studied in previous work on JMH. The goal of our study is to address this research gap.

1.1.1.2 Workload Independent Performance

Here we are looking at the performance independent of any specific usage. End-users like developers are often overwhelmed with the possibilities of workloads of a software system [17]. On the other hand, understanding the behaviour of performance with respect to workloads only will not give enough sign about the complexity of the source code, which is one crucial factor of performance optimization. Thus, in this setting, we mainly focus on the performance impact of source code changes under a fixed workload. In this thesis, we use Java functional test cases as a case study of source code to predict the performance.

1.1.1.3 Software Product Performance

Here we are measuring the performance of software products using social business intelligence (SBI). SBI combines corporate data with user-generated content (UGC) to make decision-makers aware of important brand-related trends and to improve decision-making through timely feedback [18]. Although not our primary focus, this adds a third orthogonal dimension to the thesis with regard to different measures of software system performance.

1.1.2 ML-Based Techniques

To contextualize the rest of this study, we present some background about ML-based approaches used in this study, such as representation learning of the source code and the commonly-used Graph Neural Network (GNN) models.

1.1.2.1 Code Representation

The first step in machine learning is proper data representation. So we have to represent the source code in a format suitable for the model and the task of interest. Thus, we introduce three possible forms of how source code can be represented for a machine learning model. In the literature, the code representation approaches are mainly classified into three categories: Token-based, Tree-based, and Graph-based. Every form maps different syntactical and semantic aspects of the source code to a specific data structure. These representations can then be embedded in a neural network so that they can use source code as input.

Source code is originally a text encoding representing a program. This can be processed and further transformed into different representations forms. In this section, we describe three well-known representations, each one mapping certain aspects of the original source code. We use the C snippet depicted in Listing 1.2 as a running example.

```c
void foo() {
  int x = source();
  if( x < MAX ) {
    int y = 2*x;
    sink(y);
  }
}
```

Listing 1.2: Example of C code.

Token-Based Representation This representation treats code as free text. Thus, it converts the code into a list of tokens where each word (e.g., ”void”) is a token, but each special character (e.g., ’(’) is also a token (rather than considering it as part of a word). An example is given in Listing 1.3.

```
['void', 'foo', '(', ')', '{', 'int', 'x',
'=', 'source', '(', ')', ';', 'if', '(', 'x', '<', 'MAX', ')',
    '{', 'int', 'y', '=',
'2*x', ';', 'sink', '(', 'y', ')', ';', '}', '}']
```

Listing 1.3: Token Representation for the code in Listing 1.

Then, each token will be encoded into a numerical vector using different statistical language models, such as word embedding [19] or n-grams [20]. In principle, word embedding is a learned representation for text where words that have the same meaning get a similar representation. Technically, word embeddings are a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector, and the vector values are learned in a way that resembles a neural network. Hence the technique is often lumped into the field of deep learning (DL). N-grams are several words appearing together. They are useful abstractions for modelling sequential data such as text, where there are dependencies among the terms in a sequence. However, a corpus of code can be regarded as a sequence of sequences, and corpus-based models such as n-grams learn conditional probability distributions from the order of terms in a corpus. Corpus-based models can be used for many different types of tasks, such as discriminating data instances or generating new data characteristics of a domain. Embeddings can be considered a way to represent words and help the DL model learn the source code’s representation. An embedding can be trained to represent n-grams or just individual words.

Tree-Based Representation This representation captures the abstract syntactic structure of the source code. Abstract syntax trees (ASTs) are a kind of tree representation approach that is widely used by programming languages and SE tools.

Figure 1.1 shows an example of an AST representation. The nodes of the AST tree are related to constructs or symbols of the source code. In comparison to the token-based approach, AST representation is abstract and does not include all available details, such as punctuation and delimiters. Theoretically, ASTs can be used to illustrate the lexical information and the syntactic structure of source code, such as the function name and the flow of the instructions (for example, in an if or while construct). Recently, some approaches have combined neural networks and ASTs to constitute tree-based neural networks (TNNs) [21]. Given a tree, TNNs learn the vector representation by recursively computing node embeddings in a bottom-up way. Popular TNN models are the Recursive Neural Network (RvNN)) [22], Tree-based CNN (TBCNN) [23], and Tree-based Long Short-Term Memory (Tree-LSTM) [24].

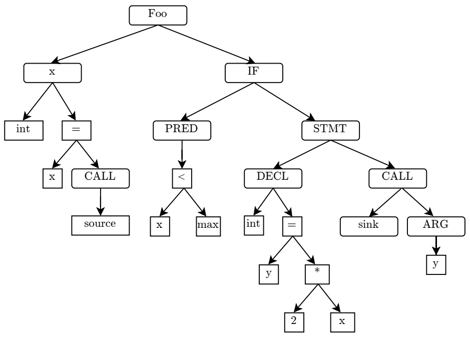

Figure 1.1: Abstract syntax tree (AST) for the code snippet in Listing 1.

Graph-Based Representation This approach represents source code as a graph at many different levels. Levels of representation define the type of the representation graph. Thus, a control flow graph (CFG, see 1.2 (a)) describes the sequence in which the instructions of a program will be executed. Thus, the graph is determined by conditional statements, e.g., if, for, and switch statements. In CFGs, nodes denote statements and conditions, and directed edges connect them to indicate the transfer of control.

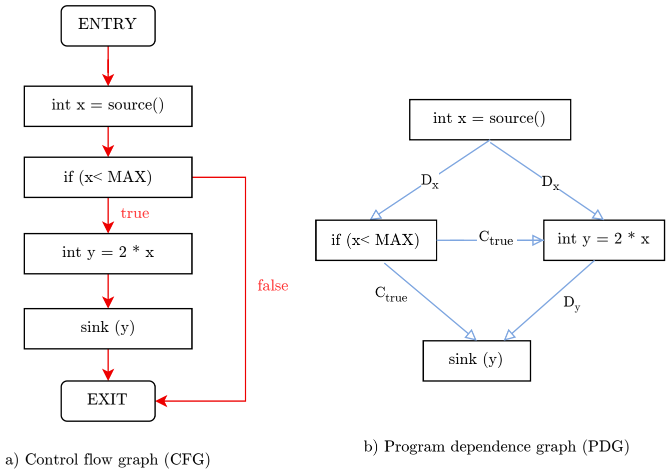

Figure 1.2: Graph-based representations for the code snippet in Listing 1.

Alternatively, the representation might be a data flow that is variable-oriented. Thus, a data flow graph (DFG) is used to follow and track the usage of the variables through the CFG. A DFG edge represents the subsequent access or modification of the same variables. The call flow graph (CallFG) captures the relation between a statement which calls a function and the called function [25]. Finally, the entire program can be represented as a graph using a program dependence graph (PDG, see 1.2 (b)), where the nodes can characterize statements and predicate expressions. In this study, we differentiate between the tree and graph-based approaches since each representation is used to retrieve a different level of information from the source code. Thus, the tree-based approach, such as using the AST, is used to extract the syntactical information from the source code, whereas graph-based approaches, such as CFG or DFG, extract semantic information.

1.1.2.2 Graph Neural Networks

If the source code is to be represented as a graph, then Graph Neural Networks (GNNs) are the right model to handle this type of representation.

Graphs are complex structures, and verifying if two graphs are identical (also known as the isomorphism test) is an important and difficult task. It is unknown if the problem can be solved in polynomial time or if it is computationally intractable for large graphs. A fast heuristic to verify if two graphs are the same is the k-Weisfeiler-Leman test [26]. The algorithm produces for each graph a canonical form. Then, if the canonical forms of two graphs are not equivalent, the graphs are not considered isomorphic. However, there is the possibility that two non-isomorphic graphs share a canonical form. Thereby, this test might not provide conclusive evidence that the two graphs are isomorphic. GNN network can be as powerful as the k-Weisfeiler-Leman test with k equal to 1, in which the canonical form propagates the information by nodes. With k greater than 1, the information is propagated among substructures of order k. Higher order graph convolution layer (k-GNN) is also proposed, wherein messages are exchanged among nodes, edges and substructure with tree nodes (triads). Once messages are exchanged among substructures, each node has a latent representation. In order to predict the property of the graphs (i.e., the execution time of a graph representing Java code), node embeddings are globally aggregated (pooling step) with an invariant ordering function (i.e. sum, max, mean). In particular, k-GNN is defined as: given is an integer k the k-element subset $[V(G)]^k$ over $V(G)$. Let $s = \{s_1, s_2, \dots\}$ be k-set in $[V(G)]^k$, then the *neighborhood* of s is defined as:

$$N(s) = \{t \in [V(G)]^k || s \cup t| = k - 1\} \tag{1.1}$$

In Equation 1.1, the neighbour of a k-set is defined as the set of k-set such that the intersection of their cardinality is equal to $k - 1$. The local neighbourhood is defined as:

$$N_L(s) = \{t \in N(s) | (u, w) \in E(G) \text{ with } u \in s/t \text{ and } w \in t/s\} \tag{1.2}$$

The local neighbourhood defined in Equation 1.2 is a subset of the neighbour (Equation. 1.1). Finally, the k-GNN is defined as:

$$f_{k,L}^{(l)}(s) = \sigma(f_{k,L}^{(l-1)}(s) \cdot W_1^{(t)} + \sum_{u \in N_L(s)} f_{k,L}^{(t-1)}(u) \cdot W_2^{(t)}) \tag{1.3}$$

The l-th layer of the k-GNN computes an embedding of s, using the non-linear activation function σ of the summation over the substructure itself in the previous layer (i.e., layer $l - 1$) and the summation over the previous layer embedding of each local neighbourhood of the substructure s.

1.1.2.3 Data-Centric AI

ML models have intensively invested in SE research and contributed to solving many SE problems on the source code level, like code clone detection [27], vulnerability detection[28], and fixing formatting [29]. In the context of performance prediction, Trials to predict performance based on source code using very efficient ML models tend to have very poor accuracy [30]. Thus, in the context of performance prediction, efficient Ml models can not deliver good accuracy in prediction. That opens the door to investing in the data rather than in the ML model by tuning the data instead of the model. That is the idea of data-centric AI. Andrew NG proposes this term in his MLOPs course. Improving the quality of the data can be done on many levels:

- Proactive dataset selection and curation: This process includes data forensics and provenance on the dataset, in addition to assessing the data for pertinence for the task and the ability to curate datasets continuously.

- Data preprocessing and cleaning: It is needed for data cleaning, preprocessing and handling the missing data.

- Data quality evaluation: This process includes the assessment of sample-level quality and ambiguity, besides data imbalances and consistency checks.

- Synthetic data improvement: This process contains the synthetic data to improve dataset, diversity and coverage in addition to increasing sample size as in annotating more data using active learning.

In this thesis, we manipulate the data-centric AI by synthetically improving the data, which is, in our case, the source code. More specifically, we improve the intermediate representation of the source code to compress both syntactical and semantic information within one intermediate representation. The approach that combines this representation with an efficient ML model can overcome the poor performance prediction accuracy based on the source code.

1.2 Research Statement

This thesis aims to investigate ML-based approaches to predicting software execution times. The ultimate goal of the thesis is to present a modern machine learning-based approach for predicting the execution time from three angles: (a) workload-independent performance, where we predict the execution time based on the source code of the system (b) workload-dependent performance; where we mainly focus on the system's workload parameters and their impacts on the performance prediction (c) software product performance, where we use many KPIs measurements in the context of SBI.

1.3 Challenges

So far, the reader already knows that performance is essential in many different domains, and predicting performance is currently challenging. In this section, we will describe four challenges that motivate our research based on the gaps in the literature.

Challenge 1 (C1): Existing approaches to predict execution time based on source code suffers from poor accuracy.

Predicting the absolute execution time of applications based on code structure is challenging as it is a function of many factors, including the underlying hardware architecture, the input parameters, and the application's interactions with the operating system. Consequently, studies that attempted to predict absolute performance counters (e.g., execution time) for arbitrary applications from source code generally report poor accuracy [31], [32].

Challenges 2 (C2): It is unclear what is a good representation for the source code to predict its execution time.

In literature, there are many ways to represent the source code to the ML model, depending on what kind of information we want to pass on to the model. As described in Section 1.1.2.1, each representation approach delivers a particular type of information. However, for a complex performance prediction problem, more information is needed to deliver to the ML model throughout the source code representation. Thus, the already existing approaches in the literature are not yet enough to combine all different types of information (syntactical, semantic, and lexical) derived from the source code.

Challenges 3 (C3): Manually testing different workloads is cumbersome.

As discussed in Section 1.1.1.1, developers usually benchmark the application based on a workload configuration to detect the system's performance behaviour under different workloads. Usually, the developer does this testing process manually. They set initial values for the workload parameters, then run the benchmark and observe the performance value. Then they update the values of workload parameters, rerun the benchmark, and observe how the performance value changed accordingly. This process consumes a lot of computational resources since the same benchmarks run many times according to the number of testing scenarios. Some benchmarks might take a long time in each run (e.g., half a day), which consumes a lot of time and effort by the developers. Moreover, most existing approaches in this area rely on software performance measurements conducted with a single workload (i.e., input fed to a system). This single workload, however, is often not representative of a software system's real-world application scenarios. So far, no research has investigated how parameterization in benchmarking is used in practice and how different important parameters are to benchmarking results.

Challenges 4 (C4): There is no good way to predict a software system's execution time before implementing and benchmarking them using a specific workload.

It is challenging and exhausting to test all scenarios of workload-dependent performance. The relationship between workload choices and their influence on performance has been extensively raised as a concern that needs to be addressed. Thus, one still needs to investigate the mapping between a given workload and the estimated performance value.

Challenges 5 (C5): There is no clear consensus about measuring the performance of the software product based on Social Business Intelligence (SBI). In the context of SBI, the most challenging issue is the lack of standard key performance indicators (KPIs) to measure product performance. Moreover, SBI research is often constrained by the lack of publicly-available, real-world data for experimental activities.

1.4 Research Questions

To address the goal of this thesis, we formulate the following research questions (RQs) motivated by the four challenges raised in the previous section:

RQ1 *What are the proper source code representations for an ML-based approach? Is it feasible to combine more than one representation?*

A solid understanding of each source code representation and corresponding representation learning is needed to represent the source code for a complex task such as performance prediction (C2). To address this gap, we conduct a systematic literature review and mapping study to study the source code representation and how each representation is invested in an ML-based approach for a particular SE task. Thus this research question aims to systematically map the landscape of existing work on source code representation, ML-based approach used for representation learning and corresponding SE task to identify each representation's characteristics and how the trials of combining many representation approaches are invested in different SE research.

RQ2 *How well can a hybrid representation approach that combines tree and graph-based approaches predict workload-independent execution times?*

All research that tried to predict the scalar value of execution time based on static analysis and traditional source code representation tends to have poor accuracy (C1). To tackle this challenge, **RQ2** aims to propose a novel method of augmenting the source code representation by combining both syntactical and semantic information in one rich representation and then applying a GNN as an ML-based model to learn this representation and predict the scalar value of corresponding source code.

RQ3 *What is the impact of the workload on execution time, both individually and in the interaction of multiple workload parameters?*

The single workload usually used in most existing approach workload-dependant performance is often not representative of a software system's real-world application scenarios (C3). To fill this gap, we conducted a systematic empirical study to understand how workload—individually and combined—affects a software system's performance. Hence, we studied the relationship between workload choices and their influence on performance.

RQ4 *How accurately can the execution time of benchmarks for specific workload parameters be predicted?*

The manual testing of the workload parameters (C3), as well as testing all scenarios of workload-dependent performance (C4), are both exhausting processes. We propose a predictive performance model based on a workload-dependent setting to address this challenge. The backbone of performance estimation is prediction models that map a given workload to an estimated performance. Learning performance models relies on a training set of workload-specific performance measurements. That is, workload-dependent performance can be highly sensitive to workload variation, and the performance behaviour under different workloads can change in unforeseeable ways. That is key to understanding whether performance models are generalizable across different workloads.

RQ5 *What is the efficiency of the SBI software product in the context of brand analysis?*

This research question is proposed to address Challenge (C5). Hence, we design and implement an ML pipeline for measuring the performance of an SBI software product for brand analysis. The metrics are extracted from one theoretical study [33]. The ML approach employed within the SBI is trying to extract the attributes to compute the KPIs in the context of brand analysis. Since there is no real data in the context of SBI for brand analysis, we follow an empirical research methodology to collect the required data from different social media platforms.

1.5 Research Methodology

This section summarizes the research methodology used to answer the research questions of this thesis. The terminology is based on the framework "ABC of Software Engineering Research" [34] for knowledge-seeking primary studies and complement with the "ACM SIGSOFT Empirical Standards" [35] and established research guidelines for solution-seeking [36] and secondary research studies [37], [38].

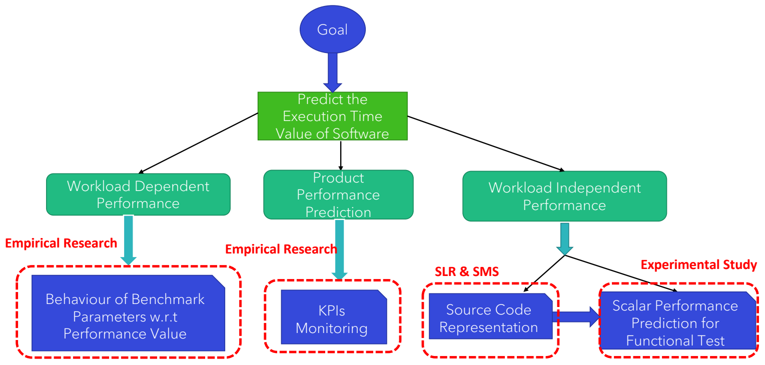
Goal
Predict the Execution Time Value of Software
Workload Dependent Performance
Product Performance Prediction
Workload Independent Performance
Empirical Research
Empirical Research
SLR & SMS
Experimental Study
Behaviour of Benchmark Parameters w.r.t Performance Value
KPIs Monitoring
Source Code Representation
Scalar Performance Prediction for Functional Test

Figure 1.3: Overall Map of the Types of Performance Studies in This Thesis and Research Methodology

Table 1.1: Mapping of research methodologies to research questions.

Research Methodology	**Section**	**RQ**
Literature Review & Mapping Study	1.5.1	**RQ1**
Empirical Study	1.5.2	**RQ3** + **RQ4**+ **RQ5**
Experimental Study	1.5.3	**RQ2**

Table 1.1 summarizes the mapping of research methodologies (this section) to the research questions of this thesis (Section1.4). For **RQ1**, a literature review (SLR) and systematic mapping study (SMS) (i.e., secondary research) were selected to address the lack of consolidated view on source code representation approaches and usage of hybrid representation combined with many ML-based approaches depending on the nature of SE tasks and the complexity of each task. The SLR and SMS were suitable because many individual studies existed. Still, the systematic mapping between representations, ML-based approaches, and the main categories of SE tasks and evidence synthesis, such as the discussion of the information that each representation can deliver as well as the pros and cons of each representation, were missing. The results of these knowledge-seeking research methodologies identify relevant gaps in the literature and practical problems to be addressed in subsequent solution-seeking research. Therefore, **RQ2** adopt experimental research to propose a novel method combining many source code representations in one rich representation that holds both syntactical and semantic information of the source code and then applies GNN as an ML-based approach to predict the scalar value of execution time (i.e., primary research). As for **RQ3** and **RQ4**, empirical research is used to collect performance measures based on workloads and study the performance behaviour according to workloads scenarios in order to eventually predict the performance value based on each workload scenario (i.e., primary research). Empirical research is also used in order to address **RQ5**, where we collect real data from different social media platforms and employ different ML and data science approaches to measure the performance of SBI

software products for brand analysis.

1.5.1 Systematic Literature Review and Mapping Study

A systematic mapping study (SMS) is a form of secondary study that aims at providing an overview of the research area (in our case, source code representation for an ML-based approach) and allows discovering research gaps and trends [39]. These form of studies, used to give an overview of a research area and designed to structure it, involves searching the literature in order to identify the topics covered in the literature [39]. Another form of secondary studies is the systematic literature review (SLR) that, in contrast, focuses on gathering and synthesizing evidence in addition to mapping the topics and synthesis of evidence from original primary studies in a defined field of research [37], [38]. Although these two forms of studies have some commonalities in the process of searching and selecting studies, they are different in terms of goals, and thus the research process is different [39]. A systematic mapping study is primarily concerned with structuring a research area [39], while a systematic review considers only evidence and their strengths [37]. In more detail, the differences are with respect to the type of research questions, analysis conducted on the literature, quality evaluation, and results [39]. The research questions in mapping studies are general and broad since they aim to classify topics covered in the literature and discover research trends [39]. Then again, systematic reviews provide in-depth analysis to answer more specific questions [38], aiming to aggregate evidence [37]. Given the classification conducted in systematic mapping studies, solution proposals are included in the analysis, while this category would not be included in SLRs [37]. This reflects the importance of systematic mapping studies in spotting research trends and topics currently in progress [37].

We use SLR and SMS in order to answer **RQ1** and investigate how source code is represented for different SE tasks. Then we map the representation with different ML-Based models used in research. As in Figure 1.4 we relied on "Google Scholar" as a database to retrieve the research studies for two reasons: (1) Google Scholar has a highly complete index, and it is unlikely that searching in other libraries would lead to additional search results, and (2) since we heavily made use of snowballing, completeness of the initial candidate set was deemed less crucial (as important missing work would appear during the snowballing process). The initial candidate list was generated by executing the following search term on Google Scholar:

"code representation" AND "deep learning" AND "Software Engineering"

The chronological range of the retrieved studies based on our interest is from 2014 until 2021. We chose 2014 as a cut-off point because this was the year the TensorFlow system (one of the mainly used platforms for deep learning) was initially released. We applied different inclusion and exclusion criteria to select the most interesting and related papers (for details, please refer to Paper II, Section 3.3). As for the snowballing techniques, we used explicitly backwards and forward snowballing to extend our initial set of candidate papers: for each selected paper, we further screened the reference list for additional relevant papers and also used Google Scholar's *"cited by"* functionality to discover later papers that have referenced papers in our initial set. We applied the same basic strategy to these additional candidate papers (screening based on title and abstract, followed by an explicit evaluation of inclusion criteria). This process has been repeated iteratively until no new papers could be found.

We defined nine research questions, including discussing the research trends and the literature gaps (Paper II Section 9). Thus, we had a general classification for the SE tasks according to how the ML-based model was used according to the input and output of the model (Paper II, Section 5.1). Thus, the ultimate aim of this study is to mainly understand how the source code is represented for the ML-based model in addition to the representation learning in order to know how the information retrieved from the source code representation contributes to the SE tasks.

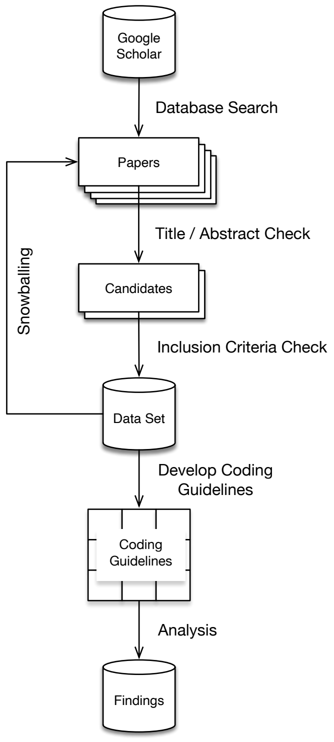

Figure 1.4: Overview of Systematic Literature Review and Mapping Study Process.

1.5.2 Empirical Research

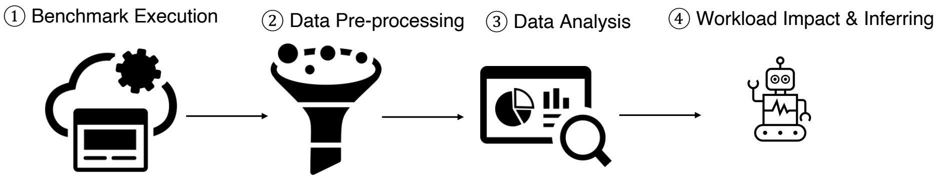

Figure 1.5: Research Process of Benchmarking Field Experiment.

According to the "ABC of Software Engineering" by Stol and Fitzgerald [34], empirical research in software engineering has been characterized by a strong emphasis on quantitative and experimental research. It refers to research approaches to gathering observations and evidence from the real world. We used this methodology in order to answer **RQ3** and **RQ4** in the context of workload dependant performance and **RQ5** in the context of software product performance.

1.5.2.1 Empirical Research for Workload Dependant Performance

According to Figure 1.5, we follow the research methodology through four main steps:

1. **Benchmark Execution:** We select five well-known open source projects as in Table 1.2 for an exploratory study in order to study the workload-dependent performance. We have chosen these projects because (i) they are frequently used in JMH, (ii) they are well maintained, well known, and important in practice (iii) they cover different ranges of different types of projects. We manually select five to six benchmark classes for each of these projects. We choose the benchmark classes with different characteristics (i.e., the structure of code) and multiple benchmarks and workload parameters, each with a different range of values (see the example of JMH 1.1 in Section 1.1.1.1, where the *SetOps* class has 4 benchmarks with @*Benchmark* annotation and four workload parameters on the class level with @*Param* annotation each workload parameter with different type/range of values). Then we build a tool that repeatedly executes the selected benchmark with randomly generated parameter values. Table 1.3 shows an example of such parameters where we have two types of parameters, numerical (i.e., *size*) or categorical (i.e., *fullyRandom*). For both types of metric parameters, we define minimum and maximum values based on the current configuration of the projects on GitHub. Then we randomly generate concrete parameter values through uniform sampling between minimum and maximum. For categorical parameters, we simply select randomly from all predefined parameter values. Thus we collected 1.4 million measures. As a process of data collection, we employ purposive sampling on both project and benchmark levels. In purposive sampling, subjects are selected explicitly

based on the preserved usefulness of the study goal rather than drawn randomly from the population.

Project	**Stars**	**Contr.**	**Description**
eclipse-collections[1]	15k	74	*"Eclipse Collections is a collections framework for Java with optimized data structures and a rich, functional and fluent API."*
RxJava[2]	42k	263	*"RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM."*
JCTools[3]	2.4k	34	*"Java Concurrency Tools for the JVM. This project aims to offer some concurrent data structures currently missing from the JDK."*
Log4J2[4]	1k	99	*"Apache Log4j 2 is an upgrade to Log4j that provides significant improvements over its predecessor, Log4j 1.x, and provides many of the improvements available in Logback while fixing some inherent problems in Logback's architecture."*
jdk-microbenchmarks[5]	na	na	*"The JMH JDK Microbenchmarks is a collection of microbenchmarks for measuring the performance of the JDK API and JVM features using the JMH framework."*

1 – https://github.com/eclipse/eclipse-collections
2 – https://github.com/ReactiveX/RxJava
3 – https://github.com/JCTools/JCTools
4 – https://github.com/apache/logging-log4j2
5 – http://hg.openjdk.java.net/code-tools/jmh-jdk-microbenchmarks/

Table 1.2: Summary of selected study subjects and GitHub metadata (stars and contributors). Data was extracted from GitHub on August 5th, 2020. Descriptions are quoted verbatim from the project's websites. Both `Log4J2` and `jdk-microbenchmarks` are hosted outside of GitHub, explaining the relatively low number of stars for the former and the absence of data for the latter.

2. **Data Pre-processing:** The collected data are stored in a textual format, and the measurement data needs to be preprocessed in a tabular format. Thus, we prepossess the textual format of data and extract the workload parameters to be the independent features for the ML-based model. As each benchmark class has its workload parameters and benchmark methods, we divided the dataset on a class level. Thus we have a dataset for each class of benchmarks.

3. **Data Analysis:** Our goal here is to identify how strongly using different parameter values impacts the resulting benchmark measurements. We study the mentioned impact on the individual and collective levels. As for the individual level, we investigate the correlation between the individual workload parameter and the measurement just to understand the behaviours of each benchmark measurement with respect to each workload. That will help the developer to anticipate how the performance measurement change according to the changes in the values of the work-

Project	Class	BMs	Parameter	Range
eclipse-collections	LongIntMapTest	5	`mapSizeDividedBy16000`	[1; 100000]
			`fullyRandom`	{true,false}
	FunctionalInterfaceTest	13	`megamorphicWarmupLevel`	[0; 10]
	IntIntMapTest	4	`mapSizeDividedBy64`	[1; 100000]
			`fullyRandom`	{true,false}
	TroveMapPutTest	1	`size`	[250000; 10000000]
			`isPresized`	{true,false}
			`loadFactor`	[0.4; 0.6]
	ChainMapPutTest	3	`size`	[250000; 10000000]
			`isPresized`	{true,false}
			`loadFactor`	[0.6; 0.9]

Table 1.3: Example from the eclipse-collections of selected study subjects, benchmark classes, and parameters. BMs are the number of distinct benchmarks in this class. The range is provided as an interval of minimum and maximum values for metric parameters. For categorical parameters, all legal values are listed. All benchmarks in a class use the same parameters.

load parameter. Individual analysis reaches its limits in cases of multiple metric parameters, which strongly impact the benchmark measurements. Hence we study the impact of the workload values on the measurements on the collective level by following the feature selection techniques widely used in ML-Based models to quantify the contribution of each feature (workload parameters) in the dependant output (benchmark measurement value). Preliminary experimentation with our dataset has shown that using an embedded feature selection approach [40] based on importance derived from decision trees and Random Forests ensembles indeed performs best on our data. Hence, we use this feature selection approach in our analysis. More details about the feature selection approaches are in Paper I, Section 3.3.2.

4. **Workload Impact and Inferring:** Here in this section, we investigate the collected data and the analysis conducted in this data in order to build a predictive performance model for predicting the execution time of the untested workloads scenarios with high accuracy, without having actually to run the benchmark for this specific parameter combinations. Thus, we use an ML model based on Random Forest regression. However, since each class has its own set of parameters and benchmarks, we will build one model for each benchmark class. Figure 1.6 shows the pipeline of the ML model for the *SetOps* class explained in Example 1.1. We use the workloads annotated as @Params and the names of the benchmarks annotated as @benchmarks as the features for the model. Then we preprocess the categorical features to factorize them. Then all resulting features are used for training a Random Forest ML model.

1.5.2.2 Empirical Research for Product Performance Prediction

In the following, we present our proposed pipeline. A high-level schematic overview is given in Figure 1.7. As a first step, we crawl three different types of data sources to retrieve different types of textual clips, namely tweets from

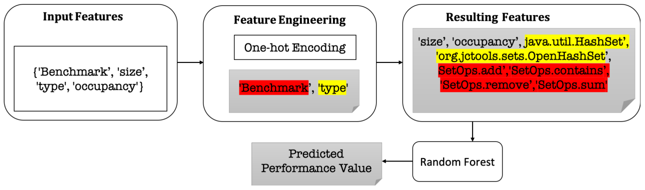
Input Features
{'Benchmark', 'size', 'type', 'occupancy'}
Feature Engineering
One-hot Encoding
'Benchmark', 'type'
Resulting Features
'size', 'occupancy', java.util.HashSet', 'org.jctools.sets.OpenHashSet', SetOps.add','SetOps.contains', 'SetOps.remove','SetOps.sum'
Random Forest
Predicted Performance Value

Figure 1.6: Data preprocessing pipeline to train the ML model. This pipeline presents an example for one specific example class (`SetOps`, from the `JCTools` project 1.1

.

Twitter, news, and reviews from Amazon. Then we collect the stock price data (numerical data) to study the impact of customers' opinions on the market's changes in stock prices. Collected textual clips are then annotated by extracting both implicit and explicit structured information. Explicit features refer to the reference time and brand. Time is usually explicitly associated with textual clips, while brand information can be extracted through a simple keyword search. Implicit features correspond to sentiment information (which is fundamental when mining customer opinion) and geolocation when it is not explicitly provided. As for sentiment information, we used a long-short-term-memory (LSTM) [41] ML-based approach to analyze the text and extract the sentiment score. According to Geolocation information, we implement a model based on Kernel density estimation ML-based approach to associate with each textual clip location. More details regarding the ML approaches used to extract implicit information of sentiment and geo-localization are discussed in Section 4 of Paper IV.

The result of the annotation phase is then transformed and loaded into a data warehouse (see Section 5 in Paper IV) using the Hadoop Data File Systems(HDFS). The ETL (extraction, transformation, and loading) process includes data transformation, cleaning, and aggregation, with the ultimate aim of computing various brand analysis metrics, which can be used for later analysis. Then we implement Spark SQL jobs for an ultimate online dashboard that can present metrics regarding the brand of interest and the competitors and their reputation in the market on a location basis.

1.5.3 Experimental Study Research

According to the ABC of Software Engineering by Stol and Fitzgerald [34], experimental study research is a research strategy conducted in a natural setting where the researcher manipulates some variables to observe some effect (e.g., performance metrics). We use this methodology in order to answer **RQ2**. Thus, we followed the guidelines proposed by Juristo et al. [42] to design and implement the experimental study.

1. **Goal Definition:** The ultimate goal of using this research methodology is to predict the execution time of software independent of any specific workload starting from the source code and based on the structure of the

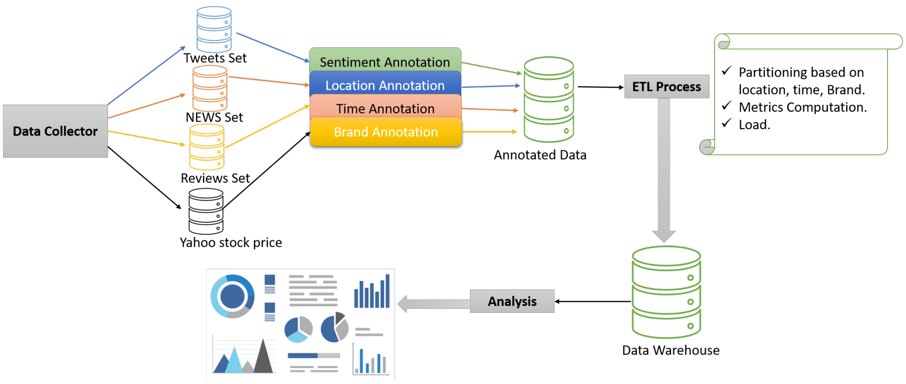

Figure 1.7: SBI Pipeline for Brand Analysis

source code. Hence, we design and implement an experimental study to take source code and represent it based on the representation approaches we investigate through our SLR 1.5.1. Then we employ an ML-Based model for predicting the execution time. For this, we use functional tests as a source code case study.

2. **Experimental Design:** In our SLR, we analyze the methodologies that are mainly used by the retrieved studies that used ML-based models for software engineering tasks. Thus, in Figure 1.8 we create a general pipeline that is representative of the steps of the aforementioned experimental studies.

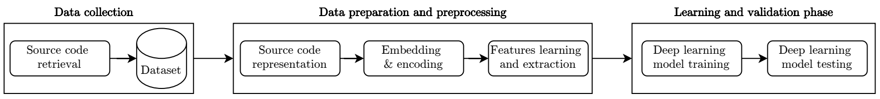
Data collection
Source code retrieval
Dataset
Data preparation and preprocessing
Source code representation
Embedding & encoding
Features learning and extraction
Learning and validation phase
Deep learning model training
Deep learning model testing

Figure 1.8: Abstracted General Code Representation and ML Models in Software Engineering.

Data collection: we collect our dataset from four well-known and diverse application domains (open source projects hosted on GitHub): *H2database*[3], a relational database, *RDF4J*[4], a project for handling RDF data, *systemDS*[5], an Apache project to manage the data science life cycle, and finally the Apache remote procedure call library *Dubbo*[6]. As labelled ground truth data, we collect 922 real test execution traces from these projects' publicly available build systems. All data were extracted from GitHub-hosted runners, which are virtual machines hosted by GitHub

[3]https://github.com/h2database/h2database
[4]https://github.com/eclipse/rdf4j
[5]https://github.com/apache/systemds
[6]https://github.com/apache/dubbo

with the GitHub Actions runner application installed. All shared runners can be assumed to use the same hardware resources, which are available at GitHub's website[7] and each job runs in a fresh instance of the virtual machine. Additionally, all jobs from which the data is extracted use the same operating system, specifically Ubuntu 18.04. This allows us to minimize bias introduced by variations in the execution environment or hardware.

Data representation and preprocessing: The second phase of the general pipeline of ML-based approaches in software engineering (Figure 1.8) is related to source code representation and representation learning. Thus, as in Figure 1.9, we first parse the functional test written in Java into its AST. We choose AST as this model contains a lot of nodes – every token of the code is represented as a node, which leads to many edges between the nodes, and for an ML model, more information is exchanged through the representation 1.1. In graph-based source code representation, every statement is represented as a node, meaning a less rich representation for the source code 1.2. For a complex problem such as execution time prediction, we need to extract all possible information from the source code. As in Section 1.1.2.1, AST can deliver syntactical information, whereas graph-based representations can deliver semantic information. Thus, we require a hybrid representation, as identified in **RQ1**, to build one representation that delivers both types of information together. Thus we augment the AST by adding edges representing control and data flow to constitute a flow-augmented AST (FA-AST). More details about this representation are contained in Paper III, Section 2.3. Then, as a part of representation learning, we initialize the embeddings of FA-AST nodes and edges before jointly feeding a vectorized FA-AST into a GNN.

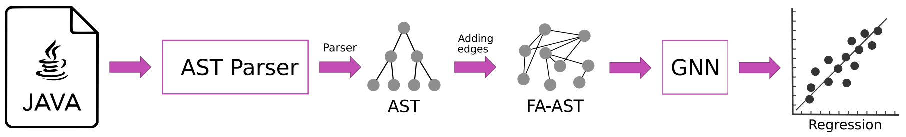

Figure 1.9: Schematic overview of the main phases of the experiment. Java unit tests are parsed into ASTs, which get augmented with control and data flow edges. The resulting graph is then used as input for a GNN.

Learning and validation: The last phase in the pipeline described in Figure 1.8 uses the ML-based approach for predicting the execution time. Thus we use a higher-order graph convolution neural network to predict the execution time. Figure 1.10 shows the architecture of the

[7]https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners#supported-runners-and-hardware-resources

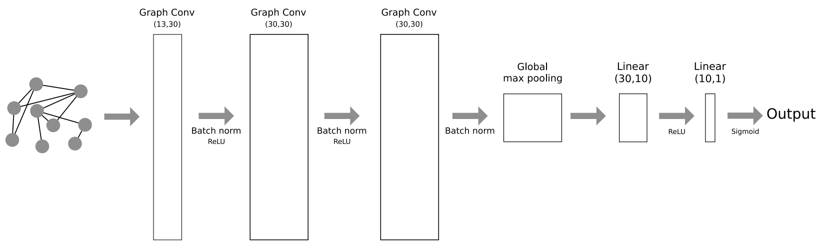

Figure 1.10: Architecture of the GNN Model used in our experiment.

used GNN model. A ReLU activation function follows each layer of this model. Since GNN learns node embedding, we use global max pooling to compute a graph embedding. Finally, the graph embedding goes into two Linear layers with a ReLU and a sigmoid activation function to perform the prediction of the test execution time. To train our model, we use the mean square error loss.

3. **Execution:** The experimental operation is to run the algorithm on the dataset collected in the data collection phase. We combine the collected data for all projects into one dataset entailing 922 code fragments and associated execution times. After that, we apply the approach as discussed in Figure 1.9. For model training, we split the dataset into train and test sets using 80% and 20%, respectively. Each network is trained for 100 epochs. As optimizers, we use Adam [43]. All experiments have been executed in a machine equipped with a GeForce 940MX graphics card and 16GB of RAM.

4. **Hypotheses Testing:** To evaluate the results of our model, we use a Pearson correlation metric, a measure of linear correlation between two sets of data. In addition, as a loss function, we use mean squared error, which is the average squared difference between the estimated and actual values.

Thus after designing and implementing the aforementioned methodologies, we obtained decent results that led to answering the RQs and practical implications.

1.6 Contribution

In this section, we will summarize the main contributions of the research done in this thesis.

- We empirically analyze the impact of different parameter values for a selection of in total 126 benchmarks of five well-known open-source software projects. We find that 40% of metric parameters in our dataset do not correlate with the measurement result (i.e., varying these parameters does not impact the measurement). If there is a correlation, it is often strongly predictable, following a power law, linear, or step function curve.

- We further show that Random Forest ensembles can be used to predict the benchmark output for untested parameter values with an accuracy of 93% or higher for all but one of 26 benchmark classes.
- We systematically study the different source code representations for an ML-based approach
- We invested in Data Centric-AI by designing and implementing a rich source code representation for the functional test code based on the hybrid approach in order to detect information about the structure of the code as well the control and data flow semantic information. That helps in increasing the efficiency of the ML model.
- We then combine this representation with the GNN model to predict the scalar absolute value of performance.

1.7 Results

This section answers the research questions and summarizes solutions to the challenges raised in Section 1.3.

1.7.1 Investigated Source Code Representation Approaches

In this section, we will present the results of the following research question:

RQ1 *What are the proper source code representations for an ML-based approach? Is it feasible to combine more than one representation?*

Answer: All three main groups of code representation introduced in Section 1.1.2.1 are used in literature, with tree-based and token-based code representation being the most prevalent. It is also notable that a substantial number of publications use a hybrid representation approach, combining multiple different representations.

1.7.1.1 Proper Source Code Representations For an ML-based Approach

Through our SLR, we analyzed the source code representation approaches that are utilized for encoding source code into a form that is meaningful and can be fed into ML models. Three primaries (groups of) techniques have emerged from our analysis: token-based representation, tree-based representation, and graph-based representation. Five concrete representation approaches emerged that do not clearly belong to any of these groups and have hence been categorized as Other. These are code gadget (the number of lines of code that are semantically related to each other [44].), binary visualization (the raw representation of any type of file stored in the file system, which exhibits similar behaviours

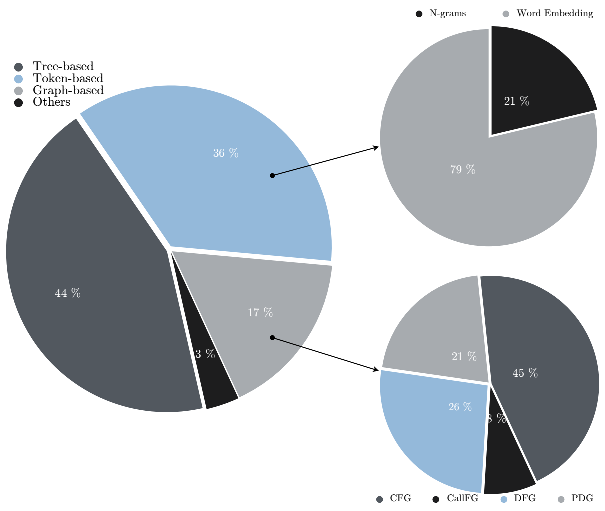

Figure 1.11: Summary of Code Representation Approaches.

of the code while being syntactically different [45]), ASCII which used by Wang et al. [46] to convert each letter of JavaScript code into eight bit binary, latent semantic indexing(LSI, a method of analyzing a set of documents in order to discover statistical co-occurrences of words that appear together which then give insights into the topics of those words and documents [47]), and bytecode (in this representation, a code fragment is expressed as a stream of bytecode mnemonic opcodes forming the compiled code [48]). An overview of the prevalence of the four groups is given in Figure 1.11.

All three groups see frequent use in software engineering. Tree-based and token-based representation are the most common and are both utilized in over half of the studies in our dataset (66 or 64% and 54 or 52%, respectively). Graph-based approaches are less common and only used in 25 (24%) of studies, but usage is increasing. The remaining techniques are only used in 5 individual publications.

For tree-based representation, the only specific technique that emerged from our study is AST. However, token-based and graph-based representations can be split into further subcategories. For token-based approaches, these are word embedding and n-grams, with word embedding being the dominant technique (used in 37 or 79% of the studies using a token-based approach, see also Figure 1.11).

There are a larger number of choices of graph-based representations, which are depicted in Figure 1.11. The most common ones are CFG (17 or 45%). Other options include Program Dependencies Graphs (PDG), DFG, and CallFG.

1.7.1.2 Hybrid Source Code Representation

Some studies have utilized a hybrid approach for code representation to capture more information on the source code. This is often promising as tree-based approaches capture syntactical information, while graph-based approaches are better at retaining semantics, and token-based approaches preserve lexical information.

Table 1.4 summarises how often different types of code representation approaches are used alone or in conjunction. The diagonal elements represent the frequency of the frameworks that have used a single representation approach, while the non-diagonal elements represent the frequency of the frameworks that have used hybrid representations. Seven studies [2], [21], [48]–[52] combined representations from all three groups. The most common hybrid approach is a combination of token- and tree-based approaches, used by 25 studies, or almost a fourth of our dataset, in total (note that 18 approaches combine only tree- and token-based representation, plus the seven studies that use all three). Combinations of the tree- and graph-based approaches are also fairly popular, used by 16 studies in total. The problem with all mentioned combined representation approaches is that they are not combined in one representation but separated into multiple representations. These separated representations constitute multiple inputs for either one ML model or multiple models (each with one input representation) to address one or more SE tasks. Thus we do not have one rich representation approach that compresses all different information from the source code. More details about these results can be found in Section 8.2 in Paper II.

All=7	**Token**	**Tree**	**Graph**
Token	25	18	4
Tree		32	9
Graph			5

Table 1.4: Frequency of Combinations of (Types of) Representation Approaches.

1.7.2 Hybrid Representation Approach and GNN for Workload-independent Execution Times Prediction

In this section, we will present the results for the following research question:

RQ2 *How well can a hybrid representation approach that combines tree and graph-based approaches predict workload-independent execution times?*

Answer: By designing and implementing one representation that compresses both syntactical information as well semantic information along with a proper GNN ML-Based model, the test execution time can be predicted with a very high prediction accuracy (Pearson correlation of 0.789).

Results shown in Figure 1.12 illustrate that our GNN model trained on FA-AST is able to predict test execution times with very high accuracy, as can be seen in the Pearson correlation (between predicted and actual execution times in the test data set) of 0.789, and a mean squared error (the average squared difference between the estimated and actual values) of 0.017. These results substantially outperform the accuracy values reported in previous studies that attempted to predict absolute software performance counters [31], [53]. We argue that the key innovation that enables this high accuracy is the combination of FA-AST as a powerful code representation model and GraphConv as a modern GNN.

We conducted more experiments for our approach on the individual projects, as shown in Paper III, Section 3.2.2.

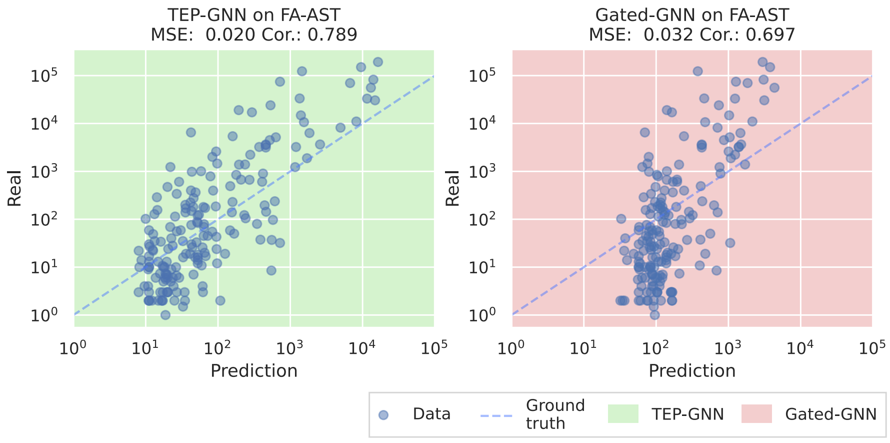

Figure 1.12: Comparison of our approach and a baseline (applying GGNN to the same FA-AST graphs). Dot points show real (y-axes) and predicted (x-axes) execution times produced by our model. The dashed line refers to the perfect prediction.

1.7.3 Impact of The Workload on Execution Time

In this section, we will present the results that answer the following research question:

RQ3 *What is the impact of the workload on execution time, both individually and in the interaction of multiple workload parameters?*

Answer: By statistically investigating the impact of individual workload parameters, we found that almost 40% of the JMH parameters have no observable impact on the measurement results, as the others have different types of correlation. On the collective level, we found out that in most cases, a single parameter dominates the benchmark result

1.7.3.1 Individual Benchmark Workload Parameters and Their Impact on Measurement Results

As we discussed in Section 1.5.2, there are two types of workload parameters, categorical and numerical. In this section, we will present a snapshot of the statistical analysis results of the impact of the individual workload on the execution time for both types of parameters. More details related to that are in Paper I, Sections 4.1 and 4.2.

Categorical parameters often fundamentally change what exactly a benchmark tests (e.g., Listing 1.4, where a categorical parameter is used to configure what specific data structure to benchmark). In the dataset collected for this study, 11 of 27 (41%) benchmark classes contain at least one categorical parameter. One class (`ConcurrentAsyncLoggerToFileBenchmark` in the `Log4J2` project) has only categorical parameters. However, we observe that there are noticeable differences between projects – most categorical parameters are found in the `JCTools` and `Eclipse-Collections` projects (where almost every benchmark has one), while they are rarely or never used in the other three projects.

```
public class QueueOfferPoll {
// ...
    @Param(value = { "SpscArrayQueue", "MpscArrayQueue", "
    SpmcArrayQueue", "MpmcArrayQueue" })
    String qType;
// ...
}
```

Listing 1.4: Example categorical parameter (from the `JCTools` project)

In all cases, choosing different values for categorical parameters leads to a multi-modal distribution in the resulting benchmark measures. An example is depicted in Figure 1.13: depending on which data structure is benchmarked, the resulting distribution changes. Note that in some cases, two or more categorical parameter values may lead to distributions that are not statistically different (e.g., `SpmcArrayQueue` and `MpscArrayQueue` in Figure 1.13).

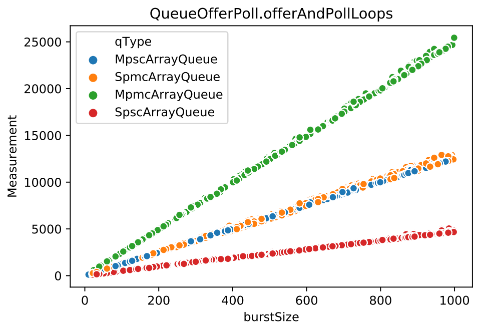

Figure 1.13: A categorical parameter leading to a multi-modal distribution of measurement values (example benchmark from `JCTools`, *burstSize* is a metric parameter, and *qType* is categorical)

In the example in Figure 1.13, all distribution modes are linear. However, we have also observed cases where different categorical parameter values have led to completely different types of distributions (Paper I, Section 4.1)

We identified the following six types of correlation from the 164 combinations of metric parameters and benchmarks distributed across 26 benchmark classes. Examples of these correlations are explained in the following. More details about other correlations can be found in Paper I, Section 4.2.

No Correlation In 66 combinations of benchmarks and metric parameters (40%), we observed that the parameter was uncorrelated with the measured benchmark value. In other words, these parameters do not appear to impact the benchmark result at all, and developers can at least consider removing them entirely.

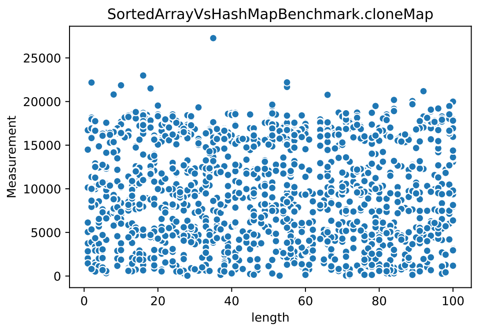

Figure 1.14: A metric parameter which does not impact the benchmark result (example benchmark from `Log4J2`, *length* is a metric parameter)

An example from `Log4J2` is given in Figure 1.14: varying the integer parameter `length` has no observable impact on the resulting measurements.

Linear Correlation Besides power law, the second type of relationship that we have observed with some regularity is linear correlations (22 combinations, or 13%). An example from the `Log4J2` project is given in Figure 1.15.

In many ways, a linear correlation is easy to handle for developers as a linear curve can theoretically be estimated from as little as two measurements. However, in practice, even linear correlations sometimes exhibit irregularities. For an example, observe the distribution of a different benchmark in the same class as the example above (Figure 1.16): while the relationship between the `count` parameter and the measured value for *threadContextMapAlias=Default* is still linear, there is a significant discontinuity around a parameter value of 800.

Logarithmic Correlation Interestingly, logarithmic correlations are considerably rare in our data. We have only observed 2 instances (1%) of this pattern, both in the `SortedArrayVsHashMapBenchmark` benchmark class of the `Log4J2` project. An example is given in Figure 1.17. Note that this plot shows the same parameter in Figure 1.15 but in a different concrete benchmark. This illustrates a technical limitation of JMH – parameters are defined in class-level,

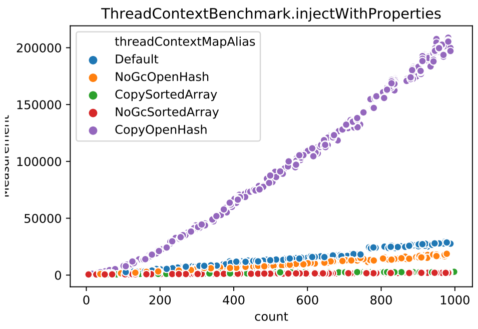

Figure 1.15: A linear correlation between metric parameter and benchmark result (example benchmark from `Log4J2`, *count* is a metric parameter, and *threadContextMapAlias* is categorical)

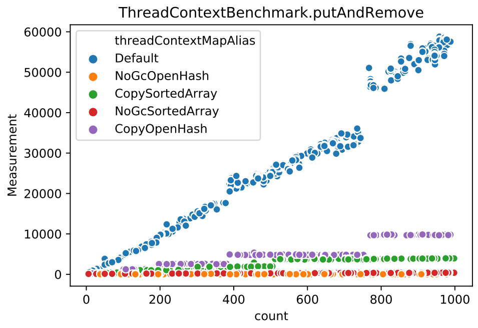

Figure 1.16: A linear correlation between metric parameter and benchmark result, with a discontinuity around *count=800* for one categorical parameter value (example benchmark from `Log4J2`, *count* is a metric parameter, and *threadContextMapAlias* is categorical)

but the same parameter may be more important to some benchmarks than to others.

1.7.3.2 Benchmark Workload Parameters and Their Impact on Measurement Results In a Collective Level

After analyzing the impact of individual parameters in isolation, we now assess the relative importance of parameters in benchmark classes with multiple parameters. In other words, we will study the impact of the workload parameters on the collective level. Table 1.5 presents the relative importance of parameters (features in the trained ML model) for each benchmark class. As parameters are defined at the class level, we report aggregate results for each class. Further note that the table omits classes with only a single parameter (as the results would be trivial). We observe that for most benchmark classes, a single parameter domin-

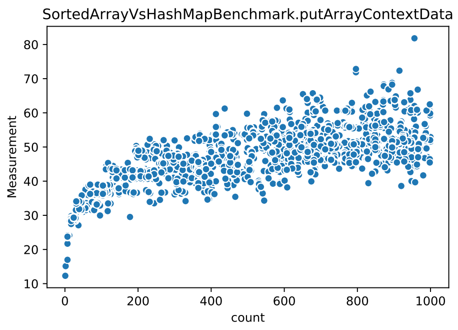

Figure 1.17: A logarithmic correlation between metric parameter and benchmark result (example benchmark from `Log4J2`, *count* is a metric parameter)

ates the measurement result (e.g., *count* for `SortedArrayVsHashMapBenchmark` or *qType* for `QueueThroughputBackoffNone`). More generally, workload-related parameters (*count*, *size*, *mapSizeDividedBy64*, etc.) are often dominant. Parameters that relate more to the configuration of the system (e.g., *threshold*, *workers*, *parallelism*, *qCapacity*, etc.) often only have a minor impact on the benchmark result. The benchmarks `ChannelThroughputBackoffNone`, `IntIntMapTest`, `ParallelPerf`, and `QueueOfferPoll` are interesting because two parameters in these classes have high importance to the benchmark result, i.e., multiple parameters where neither clearly dominates the other.

1.7.4 Inferring the Performance of Untested Parameter Values

In this section we will present the results that answer the following research question:

RQ4*How accurately can the execution time of benchmarks for specific workload parameters be predicted?*

Answer: With sufficient data used in training, the Random Forest ML model can predict the performance of untested parameter combinations with high accuracy without actually having to run the benchmark for these specific parameter combinations.

Connecting to the answer to the previous research question 1.7.3.1, most parameters either do not have an impact on the observed benchmark result at all, or the correlation between parameter value and measurement follows fairly predictable curves. Hence, we speculate that, given sufficient training data, it is possible to infer the performance of untested parameter combinations with high accuracy without having actually run the benchmark for these specific parameter combinations. To evaluate the model, we use the R-squared (R^2) metric to measure the overall model performance.

In Table 1.6, the R^2 scores for all models of all tested benchmark classes

	Class	Parameter	Score
eclipse-collections	LongIntMapTest	`mapSizeDividedBy16000`	**0.8923**
		`fullyRandom`	0.1077
	IntIntMapTest	`mapSizeDividedBy64`	**0.6841**
		`fullyRandom`	0.3159
	TroveMapPutTest	`size`	**0.7892**
		`isPresized`	0.2045
		`loadFactor`	0.0063
	ChainMapPutTest	`size`	**0.8312**
		`isPresized`	0.157
		`loadFactor`	0.0118
RxJava	ParallelPerf	`count`	**0.4711**
		`compute`	**0.4558**
		`parallelism`	0.073
	FlowableFlatMapCompletableSyncPerf	`items`	**0.8283**
		`maxConcurrency`	0.1717
JCTools	QueueThroughputBackoffNone	`qType`	**0.9835**
		`qCapacity`	0.0462
	SetOps	`occupancy`	**0.5759**
		`size`	0.3476
		`type`	0.0766
	QueueOfferPoll	`burstSize`	**0.5257**
		`qType`	**0.4741**
		`qCapacity`	0.0002
	ChannelThroughputBackoffNone	`capacity`	**0.6206**
		`type`	0.3794
Log4J2	ThreadContextBenchmark	`count`	**0.688**
		`threadContextMapAlias`	0.3123
	ConcurrentAsyncLoggerToFileBenchmark	`queueFullPolicy`	**0.8235**
		`asyncLoggerType`	0.1766
	SortedArrayVsHashMapBenchmark	`count`	**0.8517**
		`length`	0.1483
jdk-microbenchmarks	ForkJoinPoolForking	`size`	**0.7916**
		`threshold`	0.1319
		`workers`	0.0765
	ForkJoinPoolThresholdAutoSurplus	`size`	**0.9016**
		`threshold`	0.0814
		`workers`	0.017
	URLEncodeDecode	`count`	**0.4924**
		`maxLength`	**0.487**
		`mySeed`	0.0206

Table 1.5: Relative importance of parameters on benchmark class level for all benchmarks with two or more parameters.

for the *eclipse-collections* project are depicted. (The ML efficiency for the rest projects is in Paper I, Section 5.3).

We observe that the trained models perform very well for all of the classes of the *eclipse-collections* project with R^2 scores ranging between 93% and 97%.

1.7.5 Inferring the Performance of Software Product

In this section, we will present the results that answer the following research question:

RQ5 *What is the efficiency of the SBI software product in the context of brand analysis?*

Answer: With sufficient collected data, proper ML-based models, and big data analysis approaches, the proposed SBI architecture is able to compute useful KPIs for brand analysis.

Project	Class	R^2
eclipse-collections	LongIntMapTest	96%
	FunctionalInterfaceTest	94%
	IntIntMapTest	93%
	TroveMapPutTest	97%
	ChainMapPutTest	96%

Table 1.6: Accuracy of the trained models for all benchmark classes in the experiment.

We collected data for technology brands such as Apple, Huawei, and Samsung. The obtained dashboard presents different numerical, geographical and chart visual results. In the following, We will present an example of every type of result. Table 1.7 is an example of numerical results obtained by the SBI software products. Sentiment score is one example of the used KPIs (KPIs metrics and related groups are all discussed in Paper IV). The mentioned numerical results present the customer opinion for the tackled brand in different communities (data sources). Figure 1.18, on the other hand, is an example of the visual results through the dashboard, which is the main output of the SBI software product. The map in this figure presents the brand's popularity in different locations. At the same time, the line chart presents the visual correlation between the customer polarity and stock price changes for each brand through a specific timeline.

Brand	Community	Average Brand Sentiment		
		Neutral	Negative	Positive
Apple	News	1.45	2.93	2.72
Apple	Amazon	0.069	0.39	0.71
Apple	Twitter	17.11	11.36	12.06
Huawei	News	1.034	1.73	1.75
Huawei	Amazon	0.076	0.20	1
Huawei	Twitter	13.56	5.3	10.58
Samsung	News	1.03	1.83	3.37
Samsung	Amazon	0.093	0.35	1.1
Samsung	Twitter	10.82	4.52	11.62

Table 1.7: Average Sentiment, by Brand and Community

1.8 Discussion

This section will investigate how each RQ's answer contributes to solving the challenges. In other words, what are the implications of these results according

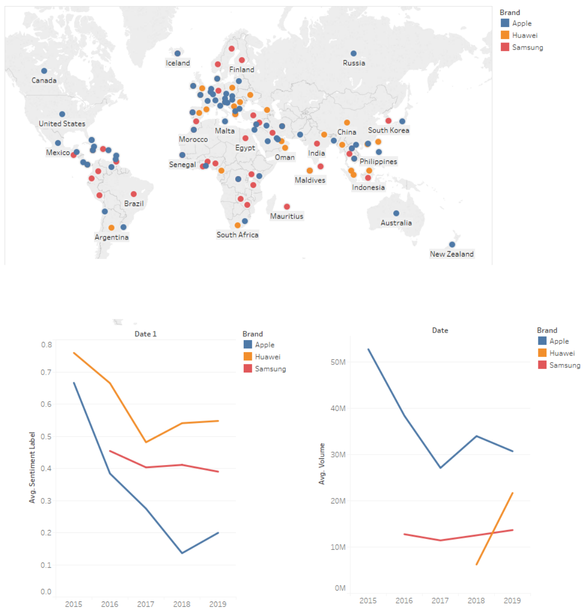

Figure 1.18: Example Brand Dashboard Containing Geographical Brand Strength and a Long-Term Comparison of Brand Sentiment and Stock Prices

to the challenges introduced above? In addition, we will present the threats to validity through the following methods.

1.8.1 Implications

RQ1 (addressing C2) discusses the different source code representations for ML models. That will eventually help the developers to understand the information conveyed by each representation. Then the developer can decide which representation is used for their SE tasks depending on the complexity of the task. In other words, we map between each representation, ML model, and the corresponding SE task. However, for complex SE tasks such as execution time prediction based on source code, the developer now has a perspective of hybrid representation approaches where different information is combined with a rich representation for ML models for more efficient prediction.

RQ2 (addressing (C1)), demonstrates that the prediction of execution times of functional tests is another context where performance prediction is possible. Test execution times are crucial in agile software development and continuous integration. While individual test cases might have short execution times, software products often have thousands of test cases, which makes the total execution time in the build process high. Our solution approach aims to speed up the testing process by optimizing the code or prioritizing test cases. Thus, the developers are provided with predictions of the execution times of their test cases and consequently giving them an early indication of the time required to run the cases in the build process. We believe that this would support decisions regarding code optimization and test case selection in the early stages of the software life cycle.

In **RQ3** (addressing (C3)), we study the impact of workload parameters on the performance both on the individual and collective level. The impact of studying this behaviour is that the developers know to expect changes in the measurements when they change the workload value since they know the correlation type. That reduces the effort and time of running the benchmarks many times to explore these changes. Moreover, for the workloads that have no impact at all (Figure 1.14), for developers, identifying parameters of this type is valuable, as these parameters can – in principle – be removed without loss of coverage in the benchmark suite. The results, on the collective level, imply that the developer should invest in one workload parameter since, for each class, we have one dominant parameter and contribute more to the measurement value. These results are beneficial since it is avoided to run the benchmark and consumes computational resources for less important tests.

RQ4 addresses (C4), supports what-if analysis, and allows developers to reason about the performance of their system in specific situations (e.g., when defining low-level throughput or response time guarantees). Thus, the developers can investigate the measurement value for untested parameters without executing the respective benchmarks.

1.8.2 Threats to Validity

Despite following a well-defined methodology, our study is always subject to limitations and threats to validity. This section will discuss the main points in three types of threats to validity, construct, and internal and external threats. We use the classification proposed by Ampatzoglou et al. [54] to contextualize these threats.

1.8.2.1 Construct Threats

In our SLR, we chose to construct our dataset based on an initial search on Google Scholar followed by extensive snowballing rather than a more conventional search strategy using major digital libraries, such as Scopus, IEEE Xplore, ScienceDirect, or the ACM Digital Library. We argue that relying on snowballing leads to a more complete and comprehensive dataset than traditional search, which suffers from limitations due to inconsistent naming and terminology. However, one challenge is that it is hard to replicate our study similarly since Google Scholar personalizes search results. To mitigate this threat, we provide a replication package that includes all studied manuscripts as well as our resulting coding sheet.

1.8.2.2 Internal Threats

A key design choice in our experimental study was the usage of existing, real-world data from GitHub's build system rather than collecting performance data ourselves (e.g., on a dedicated experiment machine). This has obvious advantages with regard to the realism of our approach but raises the threat that our training and test data may be subject to confounding factors outside of our knowledge. In particular, prior research has shown that even identically configured cloud virtual machines can vary significantly in performance. However, the high accuracy achieved by our prediction models indicates that this is not a major concern with the data we used. That said, we expect that approach would perform even better on the performance data that has been measured more rigorously.

1.8.2.3 External Threats

Our empirical study in workload-dependent performance investigated only a specific version of each benchmark project. While we argue that the chosen benchmark versions represent the problem we want to tackle, this evidence cannot be generalized to other versions, particularly since each version's performance value may change even with the same set of parameter values. Similarly, we are not able to generalize to other open-source projects. We have employed purposive sampling when identifying relevant study subjects and benchmarks. This is common in exploratory research but inherently does not allow us to draw conclusions about the population in general. Similarly, our study does not generalize to other benchmark frameworks or programming languages. To summarize, our results should not be interpreted as a comprehensive survey of benchmark parameters in JMH or micro benchmarking overall.

1.9 Conclusion

This thesis addressed the performance prediction problem in two main settings: workload-dependent and workload-independent. To reach this goal, many research methodologies have been used. We fix the workload setting and try to predict the execution time based on the source code structure to meet the workload-independent scenario. To that aim, we first did an SLR to investigate different source code representation ML models for different SE tasks. Semantic, syntactical, and lexical information are the different levels of information depending on the source code representation approach. Then we built on that study to design a useful source code representation that combines semantic and syntactical information in one representation, followed by using a GNN ML mode to predict the scalar value of the performance. Then, in the workload-dependent setting, we focused on studying the impact of workload parameters on performance both individually and collectively. That will eventually give the developer a clear strategy for testing to avoid investing more time, effort, and computational resources in unnecessary tests, especially if the tests are taken via the cloud instances where each implementation is charged.

1.9.1 Future Work

Online Learning vs offline learning This thesis followed the offline learning mode to train the ML model. In this setting, we assumed that the data was already available. In other words, enough performance measurement data was collected for training. On that basis, the predictive model will predict the complexity of the written code (execution time) once the code is written.

In the future, we will work on an online learning mode where the data is unavailable from the beginning. In this setting, the predictive mode interacts with the IDE and the developer and predicts the complexity of each chunk of the source code once it is written. So the developer in this setting does not wait until finishing the code to infer the complexity but even before.

Active Learning and Reinforcement Learning Based on our conducted SLR, one of the main gaps in the literature is the lack of data (Paper II, Section 9). Millions of source code files are available on public hosts like GitHub, but very few are labelled according to a specific SE task, which limits ML's usefulness for SE research. However, collecting the data is costly in our case since we have to execute many sample programs to collect measurements as training data. Therefore, Active Learning [55] will be used in the future to actively look for the most valuable and informative labels for our data to infer a reliable model.

Additionally, and towards investing in online learning, Reinforcement Learning will be used by designing a predictive model as an agent to analyse the code while writing in the IDE environment. Moreover, these ideas will also be extended to multi-agent reinforcement learning agents where each agent interacts with the IDE environment to collect specific information, conduct analysis, and infer to obtain accurate predictions in an online setting.

Bibliography

[1] L. Allen, A. O'Connell and V. Kiermer, "How can we ensure visibility and diversity in research contributions? how the contributor role taxonomy (credit) is helping the shift from authorship to contributorship," *Learned Publishing*, vol. 32, no. 1, pp. 71–74, 2019. DOI: https://doi.org/10.1002/leap.1210. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/leap.1210. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/leap.1210 (cit. on p. iv).

[2] J. Hua and H. Wang, "On the effectiveness of deep vulnerability detectors to simple stupid bug detection," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 530–534. DOI: 10.1109/MSR52588.2021.00068 (cit. on pp. 1, 26).

[3] Y. Li, S. Wang and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 661–673. DOI: 10.1109/ICSE43902.2021.00067 (cit. on p. 1).

[4] K. Shi, Y. Lu, G. Liu, Z. Wei and J. Chang, "Mpt-embedding: An unsupervised representation learning of code for software defect prediction," *Journal of Software: Evolution and Process*, vol. 33, no. 4, e2330, 2021, e2330 smr.2330. DOI: https://doi.org/10.1002/smr.2330. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2330. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2330 (cit. on p. 1).

[5] S. Chakraborty, Y. Ding, M. Allamanis and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, 1–1, 2020, ISSN: 2326-3881. DOI: 10.1109/tse.2020.3020502. [Online]. Available: http://dx.doi.org/10.1109/TSE.2020.3020502 (cit. on p. 1).

[6] H. P. Samoaa, A. Longa, M. Mohamad, M. H. Chehreghani and P. Leitner, *Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks*, 2022. DOI: 10.48550/ARXIV.2208.11947. [Online]. Available: https://arxiv.org/abs/2208.11947 (cit. on p. 1).

[7] J. Dorn, S. Apel and N. Siegmund, "Mastering uncertainty in performance estimations of configurable software systems," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 684–696 (cit. on p. 1).

[8] I. P. Kumara, M. Ariz, M. Baruwal Chhetri, M. Mohammadi, W.-J. V. D. Heuvel and D. A. A. Tamburri, "Focloud: Feature model guided performance prediction and explanation for deployment configurable cloud applications," *IEEE Transactions on Services Computing*, pp. 1–1, 2022. DOI: `10.1109/TSC.2022.3142853` (cit. on p. 1).

[9] D. Arcelli, "Envisioning model-based performance engineering frameworks," *Procedia Computer Science*, vol. 184, pp. 541–548, 2021, The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2021.04.008`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S187705092100781X` (cit. on p. 1).

[10] M. Woodside, G. Franks and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 171–187. DOI: `10.1109/FOSE.2007.32` (cit. on p. 1).

[11] H. Samoaa and B. Catania, "A pipeline for measuring brand loyalty through social media mining," in *SOFSEM 2021: Theory and Practice of Computer Science*, Cham: Springer International Publishing, 2021, pp. 489–504, ISBN: 978-3-030-67731-2. DOI: `10.1007/978-3-030-67731-2_36` (cit. on p. 2).

[12] Y. Einav. "Amazon found every 100ms of latency cost them 1% in sales." (2019), [Online]. Available: `https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales` (visited on 20/01/2019) (cit. on p. 3).

[13] M. D. Syer, W. Shang, Z. M. Jiang and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, vol. 24, no. 1, pp. 189–231, 2017 (cit. on p. 3).

[14] J. Cheng, C. Gao and Z. Zheng, "Hinnperf: Hierarchical interaction neural network for performance prediction of configurable systems," *ACM Trans. Softw. Eng. Methodol.*, 2022, Just Accepted, ISSN: 1049-331X. DOI: `10.1145/3528100`. [Online]. Available: `https://doi.org/10.1145/3528100` (cit. on p. 3).

[15] J. Dorn, S. Apel and N. Siegmund, "Mastering uncertainty in performance estimations of configurable software systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20, Virtual Event, Australia: Association for Computing Machinery, 2021, 684–696, ISBN: 9781450367684. DOI: `10.1145/3324884.3416620`. [Online]. Available: `https://doi.org/10.1145/3324884.3416620` (cit. on p. 3).

[16] C. Laaber, S. Würsten, H. C. Gall and P. Leitner, “Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, 989–1001, ISBN: 9781450370431. DOI: `10.1145/3368089.3409683`. [Online]. Available: `https://doi.org/10.1145/3368089.3409683` (cit. on p. 4).

[17] N. Siegmund, A. Grebhahn, S. Apel and C. Kästner, “Performance-influence models for highly configurable systems,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, 284–294, ISBN: 9781450336758. DOI: `10.1145/2786805.2786845`. [Online]. Available: `https://doi.org/10.1145/2786805.2786845` (cit. on p. 5).

[18] S. Castano, A. Ferrara, E. Gallinucci *et al.*, “Sabine: A multi-purpose dataset of semantically-annotated social content,” in *The Semantic Web – ISWC 2018*, D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa *et al.*, Eds., Cham: Springer International Publishing, 2018, pp. 70–85, ISBN: 978-3-030-00668-6 (cit. on p. 6).

[19] V. Teller, “Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition,” *Computational Linguistics*, vol. 26, no. 4, pp. 638–641, Dec. 2000, ISSN: 0891-2017. DOI: `10.1162/089120100750105975`. eprint: `https://direct.mit.edu/coli/article-pdf/26/4/638/1797597/089120100750105975.pdf`. [Online]. Available: `https://doi.org/10.1162/089120100750105975` (cit. on p. 7).

[20] T. Niesler and P. Woodland, “A variable-length category-based n-gram language model,” in *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, vol. 1, 1996, 164–167 vol. 1. DOI: `10.1109/ICASSP.1996.540316` (cit. on p. 7).

[21] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794. DOI: `10.1109/ICSE.2019.00086` (cit. on pp. 7, 26).

[22] M. White, M. Tufano, C. Vendome and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98 (cit. on p. 7).

[23] L. Mou, G. Li, L. Zhang, T. Wang and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI’16, Phoenix, Arizona: AAAI Press, 2016, 1287–1293 (cit. on p. 7).

[24] H.-H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI’17, Melbourne, Australia: AAAI Press, 2017, 3034–3040, ISBN: 9780999241103 (cit. on p. 7).

[25] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler and H. Leather, *Programl: Graph-based deep learning for program optimization and analysis*, 2020. arXiv: `2003.10536 [cs.LG]` (cit. on p. 8).

[26] B. Weisfeiler and A. Leman, “The reduction of a graph to canonical form and the algebra which appears therein,” *NTI, Series*, vol. 2, no. 9, pp. 12–16, 1968 (cit. on p. 9).

[27] N. D. Q. Bui, Y. Yu and L. Jiang, “Infercode: Self-supervised learning of code representations by predicting subtrees,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1186–1197. DOI: `10.1109/ICSE43902.2021.00109` (cit. on p. 10).

[28] D. Cao, J. Huang, X. Zhang and X. Liu, “Ftclnet: Convolutional lstm with fourier transform for vulnerability detection,” in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020, pp. 539–546. DOI: `10.1109/TrustCom50675.2020.00078` (cit. on p. 10).

[29] V. Markovtsev, W. Long, H. Mougard, K. Slavnov and E. Bulychev, “Style-analyzer: Fixing code style inconsistencies with interpretable unsupervised algorithms,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 468–478. DOI: `10.1109/MSR.2019.00073` (cit. on p. 10).

[30] H. P. Samoaa, A. Longa, M. Mohamad, M. H. Chehreghani and P. Leitner, “Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks,” in *Product-Focused Software Process Improvement*, D. Taibi, M. Kuhrmann, T. Mikkonen, J. Klünder and P. Abrahamsson, Eds., Cham: Springer International Publishing, 2022, pp. 464–479, ISBN: 978-3-031-21388-5 (cit. on p. 10).

[31] S. H. K. Narayanan, B. Norris and P. D. Hovland, “Generating performance bounds from source code,” in *2010 39th International Conference on Parallel Processing Workshops*, 2010, pp. 197–206. DOI: `10.1109/ICPPW.2010.37` (cit. on pp. 11, 27).

[32] K. Meng and B. Norris, “Mira: A framework for static performance analysis,” in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 103–113. DOI: `10.1109/CLUSTER.2017.43` (cit. on p. 11).

[33] V. Stich, R. Emonts-Holley and R. Senderek, “Social media analytics in customer service: A literature overview - an overview of literature and metrics regarding social media analysis in customer service,” SciTePress, 2015 (cit. on p. 13).

[34] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," vol. 27, no. 3, 2018, ISSN: 1049-331X. DOI: 10.1145/3241743. [Online]. Available: https://doi.org/10.1145/3241743 (cit. on pp. 13, 17, 20).

[35] P. Ralph, S. Baltes, D. Bianculli, Y. Dittrich, M. Felderer, R. Feldt, A. Filieri, C. A. Furia, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, R. Robbes, D. Mendez, J. Molleri, D. Spinellis, M. Staron, K. Stol, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, and S. Vegas, *"acm sigsoft empirical standards,"*, 2020. [Online]. Available: https://github.com/acmsigsoft/EmpiricalStandards (cit. on p. 13).

[36] R. Wieringa, "Design science as nested problem solving," ser. DESRIST '09, Philadelphia, Pennsylvania: Association for Computing Machinery, 2009, ISBN: 9781605584089. DOI: 10.1145/1555619.1555630. [Online]. Available: https://doi.org/10.1145/1555619.1555630 (cit. on p. 13).

[37] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, ver. 2.3 ebse technical report. ebse, Tech. Rep., 2007 (cit. on pp. 13, 15).

[38] V. Garousi, M. Felderer and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.09.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301939 (cit. on pp. 13, 15).

[39] M. Salama, R. Bahsoon and N. Bencomo, "Chapter 11 - managing trade-offs in self-adaptive software architectures: A systematic mapping study," in *Managing Trade-Offs in Adaptable Software Architectures*, I. Mistrik, N. Ali, R. Kazman, J. Grundy and B. Schmerl, Eds., Boston: Morgan Kaufmann, 2017, pp. 249–297, ISBN: 978-0-12-802855-1. DOI: https://doi.org/10.1016/B978-0-12-802855-1.00011-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128028551000113 (cit. on p. 15).

[40] H. Liu, H. Motoda, R. Setiono and Z. Zhao, "Feature selection: An ever evolving frontier in data mining," in *Proceedings of the Fourth International Workshop on Feature Selection in Data Mining*, H. Liu, H. Motoda, R. Setiono and Z. Zhao, Eds., ser. Proceedings of Machine Learning Research, vol. 10, Hyderabad, India: PMLR, 2010, pp. 4–13. [Online]. Available: https://proceedings.mlr.press/v10/liu10b.html (cit. on p. 19).

[41] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735 (cit. on p. 20).

[42] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st. Springer Publishing Company, Incorporated, 2010, ISBN: 1441950117 (cit. on p. 20).

[43] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: `10.48550/ARXIV.1412.6980`. [Online]. Available: `https://arxiv.org/abs/1412.6980` (cit. on p. 23).

[44] Z. Li, D. Zou, S. Xu *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. DOI: `10.14722/ndss.2018.23158`. [Online]. Available: `http://dx.doi.org/10.14722/ndss.2018.23158` (cit. on p. 24).

[45] N. Marastoni, R. Giacobazzi and M. Dalla Preda, "A deep learning approach to program similarity," ser. MASES 2018, Montpellier, France: Association for Computing Machinery, 2018, 26–35, ISBN: 9781450359726. DOI: `10.1145/3243127.3243131`. [Online]. Available: `https://doi.org/10.1145/3243127.3243131` (cit. on p. 25).

[46] Y. Wang, W.-d. Cai and P.-c. Wei, "A deep learning approach for detecting malicious javascript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, 2016. DOI: `https://doi.org/10.1002/sec.1441`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1441`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1441` (cit. on p. 25).

[47] V. Csuvik, A. Kicsi and L. Vidács, "Source code level word embeddings in aiding semantic test-to-code traceability," in *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*, 2019, pp. 29–36. DOI: `10.1109/SST.2019.00016` (cit. on p. 25).

[48] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 542–553 (cit. on pp. 25, 26).

[49] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021. DOI: `10.1109/TDSC.2021.3051525` (cit. on p. 26).

[50] C. Fang, Z. Liu, Y. Shi, J. Huang and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, 516–527, ISBN: 9781450380089. DOI: `10.1145/3395363.3397362`. [Online]. Available: `https://doi.org/10.1145/3395363.3397362` (cit. on p. 26).

[51] Y. Li, S. Wang, T. N. Nguyen and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: `10.1145/3360588`. [Online]. Available: `https://doi.org/10.1145/3360588` (cit. on p. 26).

[52] T. Sonnekalb, "Machine-learning supported vulnerability detection in source code," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, 1180–1183, ISBN: 9781450355728. DOI: `10.1145/3338906.3341466`. [Online]. Available: `https://doi.org/10.1145/3338906.3341466` (cit. on p. 26).

[53] K. Meng and B. Norris, "Mira: A framework for static performance analysis," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 103–113. DOI: `10.1109/CLUSTER.2017.43` (cit. on p. 27).

[54] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek and A. Chatzigeorgiou, "Identifying, categorizing and mitigating threats to validity in software engineering secondary studies," *Information and Software Technology*, vol. 106, pp. 201–230, 2019, ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2018.10.006`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950584918302106` (cit. on p. 35).

[55] B. Settles, "Active learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 6, no. 1, pp. 1–114, 2012 (cit. on p. 37).