Static Analysis Learning of Annotations in Microservices

By

Francisco Miguel Ramírez Méndez

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

Software Engineering Research Group
School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
December 2023

ABSTRACT

In microservices, an architectural style for building large-scale software applications, annotations play a crucial role in adding essential features for structuring and maintaining system settings. However, incorrect configurations and missing annotations impact the performance, quality and system complexity, posing significant concerns to developers. Moreover, the wrong usage of annotations generates potential bugs, and their detection may take days or even weeks due to the analysis of multiple logs and source code files.

To mitigate this, we advocate an approach to make suggestions for adding and keeping annotations according to similarities between microservice operations. The approach learns semantic relations between annotations and operations based on a database of code fragments with annotated operations. The learning process pursues converting operations into numerical vectors to find similar operations. Additionally, we extend the learning approach for creating clusters and identifying their granularity. Then, our approach predicts to which cluster an annotated operation belongs to identify its range of granularity values.

This thesis contributes to (i) a comprehensive systematic review of annotations in microservice construction, complemented by an empirical study that highlights the relevance of annotations in microservice software development; (ii) a semantics-driven learning approach that captures the relation between code fragments and annotations; and (iii) an extension of our learning approach that mines the granularity limits of new annotated operations.

Keywords: Microservices, Annotations, Static Analysis, Semantics Learning

DEDICATION

Dedicate to my daughters Franchesca and Isabella, my beloved wife Jessica, and our soon-to-arrive son Francisco. Their unwavering love and support have given me the energy and strength to navigate this challenging journey. Your presence has been my inspiration.

To my beloved mother, Martha Cecilia Méndez Chávez. I am eternally grateful for your endless dedication to providing me with the education and guidance required to achieve my aspirations. You instilled in me the sacrifices and values that shaped the person I am today.

In memory of my wise father, Francisco de Sales Ramírez Sánchez. Though you've passed away, your wisdom and support remain a guiding light in our life. I am committed to providing my children with the same love and education, ensuring your legacy lives on.

To my brothers, Jaime Enrique and Gabriel Andrés, your unwavering support and assistance during the toughest days have been a source of comfort and encouragement, demonstrating the strength of our familial ties.

This dedication is a small token of my appreciation for the immense role each of you has played in shaping my journey and contributing to the achievement of my goals. Your presence and love have been my greatest blessings.

ACKNOWLEDGMENTS

I want to express my heartfelt gratitude to the individuals who have been instrumental in completing this PhD thesis. First and foremost, I am deeply indebted to my close friend and dedicated colleague, Dr. Carlos Mera, for his resolute encouragement during the most challenging times. His unwavering belief in my capabilities and continuous motivation has been a driving force that propelled me forward, even in the face of adversity.

I also extend a sincere and special appreciation to my dedicated supervisor, Dr. Rami Bahsoon, who has been with me since my master's degree. His mentorship, encouragement, and pragmatic advice on striking a harmonious balance between academic pursuits and family responsibilities have been pivotal in shaping my academic trajectory.

Furthermore, I sincerely thank Dr Yuqun Zhang, supervisor from SUSTech University, for providing the incredible opportunity to participate in the Joint Split program that made this academic journey possible. This journey begins in Ecuador and continues between the United Kingdom and China. I also acknowledge the Government of Ecuador, which gave me an initial scholarship for my master's degree, which unfolds unlimited opportunities.

To all those who have supported, encouraged, and believed in me throughout this journey, your contributions have been truly remarkable and deeply appreciated.

Perseverance is not

a long race; it is

many short races

one after the other

Walter Elliot

Contents

Page

1 Introduction **1**

1.1 Overview 1

1.2 Problem Statement 3

1.2.1 Definition of Annotations, Types and Examples 4

1.2.2 Definition of Granularity 6

1.2.3 Abstract Syntax Tree, Definition and Usability 7

1.2.4 Prediction of Annotations 8

1.2.5 Prediction of Actions 9

1.2.6 Determining Typical Granularity 10

1.2.7 Importance of These Problems 11

1.2.8 Semantics-Driven Learning Algorithm 12

1.3 Research Methodology 14

1.4 Research Questions 15

1.5 Thesis Contributions 16

1.6 Publications Linked to this Thesis 18

1.7 Thesis Roadmap 19

2 Systematic Review of Annotations in Microservice Construction **21**

2.1 Overview 21

2.2 Research Methodology 23

2.2.1 Review Protocol . . . 23
2.2.2 Research Goal . . . 23
2.2.3 Research Questions . . . 24
2.2.4 Literature Search Strategy . . . 25
2.2.5 Publication Quality Assessment . . . 27
2.2.6 Data Extraction and Synthesis . . . 27
2.3 Results . . . 29
2.3.1 Description of Studies . . . 30
2.3.2 Description of Categories . . . 31
2.3.3 Annotations in Microservices Construction . . . 36
2.3.4 Specific Uses in Static Analysis-based Techniques . . . 40
2.4 Discussion . . . 44
2.4.1 Future Outlook for Research . . . 47
2.4.2 Threats to Validity . . . 50
2.5 Gap Analysis . . . 51
2.6 Related Work . . . 53
2.6.1 Annotations in Microservices . . . 53
2.6.2 Static Analysis-based Techniques . . . 54
2.7 Summary . . . 55

3 Classification of Microservice-Based Development Concerns 57

3.1 Overview . . . 57
3.2 The Life Cycle of Microservices at Runtime . . . 59
3.3 Study Design . . . 60
3.3.1 Research Question . . . 61
3.3.2 Search and Selection Process . . . 61
3.3.3 Data Extraction and Synthesis . . . 63

3.4 Results . . . 64
3.4.1 Years vs Life Cycle Activities . . . 64
3.4.2 Findings per Life Cycle Activities . . . 65
3.4.3 Further Categorisation . . . 74
3.4.4 Threats to Validity . . . 76
3.5 Complementing the Systematic Literature Review . . . 77
3.6 Related Work . . . 78
3.7 Summary . . . 79

4 A Semantics-Driven Learning for Annotations in Microservices 81
4.1 Overview . . . 81
4.2 Proposed Approach . . . 83
4.2.1 Pre-processor . . . 85
4.2.2 Learner . . . 87
4.2.3 Predictor . . . 92
4.2.4 Similarity Finder and Adviser . . . 96
4.3 Evaluation . . . 97
4.3.1 Hyper-parameters . . . 99
4.3.2 Experiment Setup . . . 100
4.3.3 Results and Discussion . . . 101
4.3.4 Threats to Validity . . . 105
4.4 Related Work . . . 107
4.5 Summary . . . 108

5 A Mining Approach to Limit Granularity of Annotated Operations 109
5.1 Overview . . . 109
5.2 Proposed Approach . . . 111
5.2.1 Operations with Annotations . . . 111

5.2.2 Abstract Syntax Tree of Operations . . . 112
5.2.3 Approach Components . . . 113
5.3 Evaluation . . . 122
5.3.1 Experiment Design . . . 124
5.3.2 Hyper-parameters . . . 125
5.3.3 Experiment Setup . . . 126
5.3.4 Results and Discussion . . . 127
5.3.5 Threats to Validity . . . 131
5.4 Related Work . . . 132
5.5 Summary . . . 134

6 Reflection and Appraisal 135
6.1 Overview . . . 135
6.2 How the Research Questions are Addressed . . . 135
6.3 Reflection on the Research . . . 140
6.3.1 Selection of Operations and Annotations . . . 140
6.3.2 Evaluation Using Natural Language Processing . . . 141
6.3.3 Integrity in the Operation Database . . . 142
6.3.4 Overhead . . . 143
6.4 Concluding Remark . . . 144

7 Conclusion and Future Work 145
7.1 Overview . . . 145
7.2 Contributions of the Thesis . . . 146
7.3 Future Work . . . 147
7.3.1 Enhanced Annotations Semantics. . . . 147
7.3.2 Annotation Impact on Non-Functional Aspects. . . . 148
7.3.3 Dynamic Granularity Adjustment. . . . 148

7.4 Closing Remarks 148

References **151**

List of Figures

1.1 Code to AST Representation 8

2.1 Collection of Papers Metadata. 27
2.2 Distribution of Studies per Year. 29
2.3 Distribution of Studies per Categories. 32
2.4 Distribution of Purposes for Using Annotations. 39
2.5 Distribution of Purposes Grouped by Context. 40
2.6 Static Analysis-based Techniques and Tools in Purposes. 41
2.7 Specific Uses of Annotations and their Relation in Microservice Construction. 42
2.8 Specific Uses of Annotations and Static Analysis-based Techniques. 43

3.1 Microservice-based System Lifecycle at Runtime 59
3.2 Numbers and Stages of our Search and Selection Process 62
3.3 Life Cycle Activities Over Time 64

4.1 Conceptual Model of Semantics-Driven Learning Approach 83
4.2 Nodes of an AST Representation 86
4.3 Neural Network Architecture 88
4.4 Encoder and Decoder 93
4.5 Numerical Representation from Encoder 94
4.6 Performance and Quality of our Approach 102
4.7 Accuracy of Actions and Annotations 103
4.8 Analysis of Wrong Suggestions 104

5.1 Operation Selection and Granularity Exploration 112
5.2 The Components of our Approach 114
5.3 Isolated and Linked Annotations 127
5.4 Mining Operations 129
5.5 Granularity Limits 130
5.6 Overall Results 131

List of Tables

1.1 Type of annotations and examples. 5

1.2 Annotations and examples utilised in this thesis. 5

2.1 Digital Repositories for SLR 25

2.2 Inclusion and Exclusion Criteria for Study Selection 26

2.3 Quality Questions for Paper Assessment 28

2.4 Data Extraction 28

2.5 Number of Studies per Publication Name (Top 10). 30

2.6 Number of Studies per Publisher. 31

2.7 Catalogue: Purpose of Using Annotations. 34

2.8 Catalogue: Specific Uses of Static Analysis-based Techniques. 35

2.9 Annotations in Microservice Construction 36

2.10 Related Work 53

3.1 Inclusion and Exclusion Criteria for Post Selection 63

3.2 Classification of Posts 66

3.3 Classification of Posts *(Continued from previous page)* 67

3.4 Further Categorisation 75

4.1 Semantics-Driven Learning Settings 100

4.2 Related Work 107

5.1 Limits of Granularity Settings 125

5.2 Related Work 133

Chapter One

Introduction

1.1 Overview

A microservice architectural style is a software development approach that implements a set of refined and highly cohesive services [34, 53]. This approach offers benefits to improve scalability, flexibility, and maintainability. Unlike traditional monolithic architecture, microservices enable developers to concentrate on small and dedicated business domains. Thus, developers do programming and testing faster and more efficiently. Deploying microservices across a network makes the application more resilient to failures [109] because failures in one service do not necessarily lead to a failure of the entire system. In this context, code annotations or annotations provide additional information attached to classes, methods, fields, and parameters for assisting architectures in implementing services [86].

Microservice involves the usage of annotations, which are essential for architecture recovery, vulnerability detection and recognising which annotations are missing. In this

sense, annotations in microservice construction refer to a set of specific uses of annotations that involve static analysis techniques to achieve related purposes in software development and improve the overall quality of the applications, leading to more efficient and reliable microservices. However, despite the importance of annotations, their incorrect usage and inadequate settings negatively impact the quality of services, which may reduce performance for high availability and fault tolerance [84]. Moreover, as the adoption of microservices continues to grow, the challenges of annotations in microservice construction become crucial, considering that developers struggle to identify and predict annotations.

This thesis proposes an approach that utilizes annotations to group operations based on similar behaviour. Our approach performs semantics learning to convert operations into their vector representation. With a sequence-to-sequence model, our approach can predict annotations of operations, predict actions over annotations and identify the typical granularity for unseen operations. Semantics learning extracts features from a text by focusing on its syntax and semantics. Our approach requires a collection of operations with their respective text representation. Since code clone detection algorithms have shown promising results with Abstract Syntax Tree (AST) representation, we rely on an AST of source code. Considering the characteristics of grouping operations of our approach, we can extend it to compare granularity levels and other properties of operations, helping to reduce the inadequate granularity selection by observing levels in similar operations applied in other applications.

Whilst annotations help evaluate test cases and detect bad smells in microservices [94, 93], there exists discussion on using annotations in microservices projects for detecting clones of code fragments. Pigazzini et al. form groups of similar methods where distant groups mean different functionalities [92]. Perez et al. use a syntax tree-based skip-gram algorithm on different programming languages [90]. Unlike those works, our approach centres on using annotations by extracting terms and learning their relation with the code. In this thesis,

we explore the relevance of annotations and evaluate the importance of learning semantics information to predict annotations and find typical granularity.

1.2 Problem Statement

The employment of microservices grows for building large-scale software applications due to their flexibility and scalability. Microservices implement refined and highly cohesive services that depend on each other across multiple service instances to ensure high availability and fault tolerance [34, 53]. These services generally utilise load balancers and circuit breakers to run their instances effectively [118]. Specifically, Spring Boot framework provides program metadata named annotations to facilitate implementations of these features without changing the primary functionality [24]. Thus, annotations play a crucial role in microservices to the extent that a developer community positions missing annotations as a top concern [96].

However, using annotations in microservices presents several challenges, including choosing appropriate annotations, their correct usage, and their proper changes over time. These challenges affect the microservice quality, introduce errors, and increase maintenance costs. In particular, the size of microservices plays a critical role in exacerbating these challenges. Tiny microservices introduce managing issues into the whole architecture, and huge microservices affect the quality attributes, especially performance, which reduces the overall system quality [121]. In the face of these challenges, there is no agreement on the right size of microservices because project teams interpret the size in terms of line of code, number of classes, and entities, among others [53].

We advocate that microservice construction can benefit from semantic information by learning the relation between annotated operations. For example, detecting annotation-related defects may benefit from the semantic information to analyse annotation usage and

identify incorrect and missing annotations. Another example would be the collection of granularity values of annotated operations, grouped by their similar behaviour and choice of the granularity boundaries for a new operation. Additionally, Natural Language Processing (NLP) techniques may enrich the analysis of annotated operations by extracting valuable information to detect patterns of operation similarity and annotation usage.

In this thesis, we propose a semantics-driven learning to demonstrate its feasibility for detecting missing annotations and identifying granularity of annotated operations. In summary, these concerns encompass the purposes of using annotations and their impact in overall quality of microservices. Our approach addresses these concerns effectively by recognising and clustering similar operations. Moreover, the integration of NLP techniques contributes to the understanding of annotations and managing of granularity.

1.2.1 Definition of Annotations, Types and Examples

Annotations are program metadata that add features to the source code without changing behaviour. These annotations provide additional information about classes, methods and fields that support the reuse of features and software evolution. Additionally, developers commonly utilise frameworks which offer stacks of reusable components to improve productivity and maintain consistency. Furthermore, microservice frameworks provide annotations to facilitate the implementation of cloud-based applications.

Type of annotations refers to a classification based on their characteristics and behaviour. Common types include marker annotations, which do not contain any element; single-valued annotations, which have only one element; multi-valued annotations, which have multiple elements, each one with one value; meta-annotations, which annotate other annotations to specify how an annotation should be treated; standard annotations, which

are commonly used across different frameworks; and custom annotations, which are defined to tailor specific requirements. Table 1.1 presents examples of annotations and their types.

Table 1.1: Type of annotations and examples.

Type	**Annotation**	**Description**
Marker	Override	Indicates that a method overrides a superclass meethod.
Single-valued	Depricated	Marks a program element that is no longer recommeded for use.
Multi-valued	RequestMapping	Maps HTTP requests to handler methods.
Meta	Retention	Sets how long an annotation should be retained.
Custom	Auditted	Indicates that a method or class should be audited for a logging framework.
Standard	SupressWarnings	Suppresses compiler warnings due to code constructs.

Table 1.2 presents the annotations utilised during the experimentation in this thesis. These annotations belong to the following frameworks: JUnit to write and run unit tests, Java EE to build enterprise applications, Spring MVC to develop web applications based on model-view-controller pattern, and JAX-RS to create RESTful web services. These specific annotations represent the annotations with enough operations available in the source code repositories explored to build the datasets for the experiments in Chapter 4 and Chapter 5.

Table 1.2: Annotations and examples utilised in this thesis.

Framework	**Annotation**	**Description and example**
JUnit	Before	Executes a method before running each test method. For example: @Before
Java EE	PostConstruct	Executes a method after initialising the bean. For example: @PostConstruct
Spring	Bean	Returns a bean instance to be managed by Spring container. For example: @Bean
Spring MVC	RequestMapping	Maps HTTP a request to a method. For example: @RequestMapping("/home") or @RequestMapping(value = "/hello", method = RequestMethod.GET)
Spring MVC	GetMapping	Maps HTTP GET requests onto specific handler methods. For example: @GetMapping("/welcome") or @GetMapping("/dashboard/{id}")
JAX-RS	GET	Indicates that the annotated method responds to HTTP GET requests. For example: @GET("/users/{id}") or @GET("/dashboard")
JAX-RS	POST	Indicates that the annotated method responds to HTTP POST requests. For example: @POST("/login") or @POST("/upload")

Listing 1.1 shows a Java code fragment which defines a microservice operation using Spring framework. The code fragment contains two annotations each one at two different levels. @RestController creates a web service and indicates that a class will handle incoming requests. @GetMapping maps HTTP requests to a specific operation; in this case, the greeting request calls the greeting method, which performs business logic and returns data to the client.

```
1  package com.example.annotations;
2
3
4  import org.springframework.web.blind.annotation.RestController;
5  import org.springframework.web.blind.annotation.GetMapping;
6
7  @RestController
8  public class GreetingController{
9
10
11     @GetMapping("/greeting")
12     private Greeting greeting() {
13
14         String name = "World";
15
16         return new Greeting(String.format("Hello, %s!", name));
17     }
18     }
19
```

Listing 1.1: Source Code of Annotations

1.2.2 Definition of Granularity

Granularity is the size of the functionality in a microservice. It can encompass the services with the complexity and dependencies of their operations [53]. This granularity is measured by the number of published operations, microservices, code lines, and complexity. In this sense, various types of granularity metrics can be used to evaluate and measure the extent of decomposition. In this thesis, we categorise the granularity according to the scope in:

- Operational granularity, which quantifies the number and complexity of operations exposed

by individual microservices. For instance, a fine-grained service can offer a single operation to perform a specific task, while a coarse-grained service can encompass multiple operations handling a broader set of functionalities.

- Service granularity, which relates to the number of individual microservices composing the application and their interdependencies. For instance, a fine-grained architecture can consist of numerous narrowly-focused microservices, each responsible for a single business capability, whereas a coarse-grained architecture can feature fewer, more comprehensive microservices encompassing multiple functionalities.

Considering that granularity can be assessed in terms of code lines and complexity, where finer granularity implies smaller, more concise units of code with lower complexity, while coarser granularity involves larger, more intricate units of code spanning multiple functionalities. In this thesis, we propose the number of tokens as an operational granularity measure. In terms of Natural Language Processing (NLP), a token is the smallest unit of text that has been segmented from a larger body of text; each token represents a discrete unit of meaning within the text and serves as the basis for various NLP tasks such as parsing, sentiment analysis, or machine translation.

1.2.3 Abstract Syntax Tree, Definition and Usability

An Abstract Syntax Tree (AST) is a tree representation of source code that provides syntactic knowledge on using adopted frameworks [30]. Specifically, an AST contains additional semantic information, and a reading process expresses the tree nodes as a statement. Figure 1.1 presents the AST representation of a code fragment. The most common reading process is the traversal algorithm, which offers three ways to read the tree: inorder, preorder, and postorder. Of these, the preorder is the most employed to read an AST [143, 51].

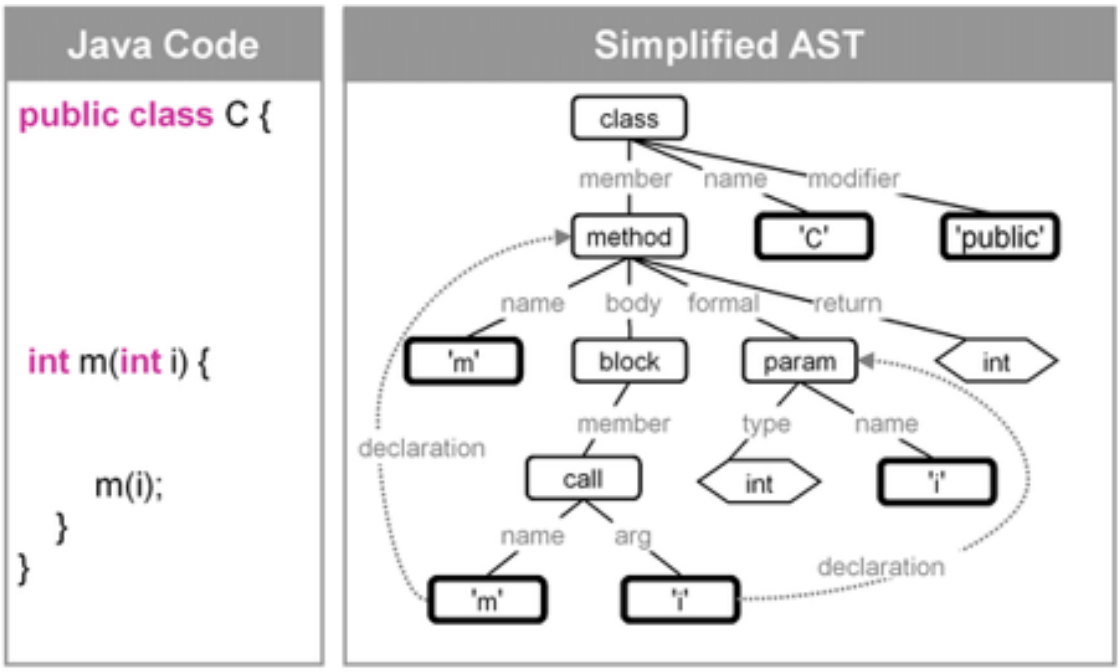


Figure 1.1: Code to AST Representation

After reading the AST representation in preorder, we can derive a text representation of a code fragment named an AST statement. Subsequently, employing tokenization, a fundamental task in Natural Language Processing (NLP), allows us to extract tokens from this statement. These tokens, representing the smallest units of text, provide valuable insights into the structure and semantics of the code, facilitating further analysis and processing. Here, a token is essentially a numerical representation of an AST element, and a sequence of tokens is a numerical vector representation of the whole AST statement.

1.2.4 Prediction of Annotations

Developers frequently need to comprehend a reduced amount of code functionality, often relying on code fragments [73, 51], which is time-consuming due to outdated or missing comments [132]. Additionally, incorrect annotation usage introduces errors, leading to unexpected behaviour. Despite the importance of annotations, only some approaches exist

to detect annotation issues, like cycle dependency and misuse [93, 94]. However, these approaches are limited to specific patterns and rules without offering mechanisms for predicting annotations. Addressing this gap requires learning from annotated code fragments to predict annotations effectively.

The prediction problem investigated in this thesis revolves around the annotation that best matches the purpose or functionality of a given code fragment. This objective entails identifying distinct annotations, such as Before, PostConstruct, or RequestMapping. The prediction task is based on numerical vectors corresponding to the AST representation of code fragments extracted from open-source repositories. Each annotation has a unique number associated with a label for each numerical vector of a code fragment. We use machine learning algorithms to train predictive models capable of classifying code fragments into predefined annotations.

To address the prediction problem, the thesis focuses on harnessing the structural and semantic information encoded within AST representations of code fragments. These ASTs are the primary data source for training a learning model that converts the code fragment into numerical vectors. Then, the numerical vectors serve as the primary input for training and evaluating predictive models. By parsing code fragments into ASTs, the thesis extracts meaningful features and patterns that capture the essence of each piece of code. These features may include node types, hierarchical relationships, token sequences, or syntactic structures inherent to the code.

1.2.5 Prediction of Actions

The problem of predicting actions regarding annotations involves determining whether to add or keep an annotation within a given code fragment based on its corresponding Abstract

Syntax Tree (AST) representation. This task goes beyond merely predicting an annotation associated with a piece of code; instead, it focuses on providing actionable recommendations for developers to enhance code quality and maintainability.

To address the problem of predicting actions, we adopt a heuristic algorithm that leverages the predictive model described in Section 1.2.4. This model learns the rich information encoded within AST representations to predict annotation. The prediction can be evaluated with the original piece of code; if the original code does not have annotations, then the action is to ADD the predicted annotations. On the other hand, if the original code has annotation and is the same as the predicted annotations, then the action is to KEEP the predicted annotation. By integrating these actions into development workflows, developers can make informed decisions regarding managing and utilising annotations.

1.2.6 Determining Typical Granularity

Microservice-based applications have different granularity because there is no agreement on the right granularity, which can produce issues. For instance, applications with tiny microservices introduce managing issues into the whole architecture. On the other hand, large microservices affect the system performance and reduce the overall system quality [121]. Then, inadequate granularity can introduce defects where fixing them is time-consuming. Moreover, the detection effort to solve the above issues and debugging microservices may take days or weeks [131].

We determine the typical granularity of annotated operations by exploring the different numbers of tokens available in similar operations from open-source projects. The whisker and box charts present the distribution of different granularity values and show the outlines and percentiles for each group of similar operations. After examining the distribu-

tion of granularity values for a given group of operations with similar behaviour, we define the granularity limits as the low value for percentile 25 and the high value for percentile 75.

1.2.7 Importance of These Problems

The problems addressed in the thesis, including the prediction of annotations, prediction of actions regarding annotations, and determining typical granularity in microservice operations, are crucial for enhancing software development practices and improving the quality of microservice-based applications.

Firstly, the prediction of annotations is essential as it provides developers with automated assistance in annotating code fragments accurately. By accurately determining the most appropriate annotations for code segments, developers can enhance code clarity and maintainability. This prediction capability streamlines the software development process, especially in large-scale projects where manual annotation can be time-consuming and error-prone. Additionally, accurate annotation prediction aids in standardising code practices and ensuring adherence to coding conventions and best practices, thereby improving overall code quality and reducing the likelihood of errors and bugs.

Secondly, the prediction of actions regarding annotations is crucial for providing actionable recommendations to developers regarding the management and utilisation of annotations within code fragments. By determining whether to add or keep annotations based on similar code fragments, developers can optimise code quality, maintainability, and performance. This predictive capability enables developers to proactively address potential issues related to annotation misuse, redundancy, or inconsistency, thereby minimising technical debt and improving software reliability and scalability. Furthermore, integrating predictive actions into development workflows facilitates a seamless and efficient development process,

where developers can make informed decisions regarding annotation management without disrupting their workflow.

Finally, determining the typical granularity in microservices is crucial for guiding architectural decisions, optimising system design, and ensuring the scalability and reliability of microservice-based applications. The granularity of microservices directly impacts various aspects of application design, performance, and scalability. Fine-grained microservices offer greater flexibility, agility, and autonomy, while coarse-grained microservices reduce management overhead and simplify communication between components. By analysing the distribution of granularity values based on percentiles, developers can identify outliers and anomalies in microservice design, enabling them to refactor or optimise operation boundaries to enhance system performance, scalability, and maintainability.

1.2.8 Semantics-Driven Learning Algorithm

Algorithm 1 describes the overall steps with description of inputs and output for a semantics-driven learning approach. The main inputs of the algorithm are (i) a list of microservice annotations, (ii) a dataset of classes in text files with annotations for training, (iii) a database of code fragments with or without annotations to find the best set of microservice annotations, and (iv) a list of queries to perform the experiments by predicting annotations and suggesting actions.

From the inputs, the list of annotations limits the scope of annotations under analysis. This limitation is according to the collection of code fragments. For instance, Chapter 4 focuses on annotations like Bean, Before, PostConstruct, RequestMapping and GetMapping. In contrast, Chapter 5 focuses on annotations like RequestMapping, GetMapping, GET and POST. The dataset of classes, the database of code fragments, and the list of queries are instructions in a programming language such as Java.

The main components of our algorithm are (i) the pre-processor, which receives the datasets as row data and prepares them for the other components; (ii) the learner, which trains the model and returns an encoder to convert code fragments into numerical vectors; (iii) the finderAdviser where the finder returns for the closets code fragments given a query; (iv) the predictor component, which predicts annotations; and (v) the finderAdviser where the adviser receives predicted annotations to create suggestions of actions.

The main output of our algorithm is a set of suggestions in terms of actions, each followed by the annotation name, e.g., *KEEP PostConstruct*. The *ADD* action advises the incorporation of an annotation, and the *KEEP* action suggests no change in the usage of an annotation. With this information, we evaluate the accuracy of predictions for annotations and actions. Finally, the algorithm requires an adaptation to find the typical granularity values, which is part of Chapter 5.

Algorithm 1 Semantics-Driven Learning Algorithm

Input: *datasets* // contains the datasets for training, validate and testing the learning model
Input: *database* // Java database to search similar code fragments
Input: *queries* // List of queries to predict annotations and suggest actions
Output: *allActions*

```
1:  preProcessor ← new PreProcessor()
2:  learner ← newLearner()
3:  predictor ← newPredictor()
4:  finderAdviser ← newFinderAdviser()
5:  allActions ← newList()
6:  trainingDataset ← preProcessor.prepare(datasets)
7:  encoder ← learner.buildModel(trianingDataset).getEncoder()
8:  finderAdviser.setDatabase(database)
9:  finderAdviser.setEncoder(encoder)
10: for each query ∈ queries do
11:     encodedSubset ← finderAdviser.getClosestCodeFragments(query)
12:     predictor.train(encodedSubset)
13:     predictedAnnotation ← predictor.predictAnnotation(query)
14:     actions ← finderAdviser.suggestAction(predictedAnnotation)
15:     allActions.add(actions)
16: end for
```

1.3 Research Methodology

The challenges and concerns surrounding the microservice construction motivate this thesis.

Problem identification and motivation: The research begins with a comprehensive Systematic Literature Review (SLR), aimed at gaining insights into annotations and their relation to microservice construction. Through this review, we aim to boost our understanding of the field and allows us to identify open challenges. Additionally, we analyse posts from one of the largest sources for developers to measure the practical relevance of annotations in real-world development scenarios.

Objective definition for a solution: The primary focus of this thesis is addressing the challenges associated with annotations in microservice construction. To achieve this, we propose a semantics-driven learning approach designed to support the specific uses of annotations. Our key objectives include the development of predictive models that learn the relation between annotations and operations, facilitating the automated prediction of annotations based on code characteristics. Furthermore, we attempt to explore the semantics length of operations to mine the typical granularity of annotated operations. By accomplishing these objectives, we aim to provide valuable insights that empower researchers and practitioners in optimising the usage of annotations.

Design and development: We conduct a systematic literature review for annotations in microservice construction. The results show that defect prediction is the top purpose of using annotations; and that graph theory and tools help identify annotations mainly for defect prediction and architecture evaluation. Moreover, we classify specific use of annotations and static analysis techniques to identify top relations. The review revealed that mining annotations and constraints played a crucial role in verifying annotations, commonly used to detect annotation misuse. In light of this, we elaborate a semantic-driven approach

to predict annotations by learning their relation to operations.

Demonstration and evaluation: We implement a similarity finder tool that mimics a development environment using real open-source projects. We build semantics learning in PyTorch [58], a library for implementing deep learning solutions. We also add the SciKit Learn library [18], which implements cluster algorithms based on Numpy arrays, another library to manage data. Our experimental operations database collects real exposed operations from GitHub with annotations: RequestMapping, GetMapping, POST, and GET. The quantitative evaluation of the experiments for our approach targets the effectiveness of searching operations within similar behaviour.

1.4 Research Questions

Inspired by the challenges and complexities inherent in microservice construction, this thesis endeavours to address critical concerns surrounding the utilisation of annotations. Specifically, it seeks to delve into the purposes behind employing annotations in microservice development and evaluate their prevalence as a common concern within this domain. Additionally, the thesis aims to identify potential gaps in existing techniques derived from static code analysis. To the best of our knowledge, this thesis is the first to identify the purposes of using annotations, predict annotations for code fragments, and group similar operations by annotation to determine typical granularity. This thesis proposes solutions to address the following Research Questions (RQs):

- **RQ1**: What are the purposes of using annotations in microservice construction? How do static analysis-based techniques support the purpose of using annotations? To what extent is the use of annotations is one of the most common concerns in microservice development?

  Answering RQ1 guide us to gain insights into existing literature of annotations in mi-

croservice construction, identify purposes and gaps in the scope of existing techniques for static code analysis applied in academia and highlight further research areas.

- **RQ2**: How can we leverage semantic connections between code fragments and microservice annotations to predict annotations?

  Answering RQ2 guide us to gain a deeper understanding of the opportunities in a static analysis approach that identifies the lack or misuse of annotations.

- **RQ3**: How can annotations contribute to the understanding of typical granularity degree within existing microservices?

  Answering RQ3 guide us in understanding the best granularity boundaries in practices to improve the overall quality of microservice-based systems.

Next chapters address these questions.

1.5 Thesis Contributions

The research contributes to the broader area of annotations in microservice construction. In particular, the thesis contributes to a novel annotations-driven approach. The approach leverages cloning detection to implement semantics-driven learning for clustering similar annotated operations. Specifically, we plan a thesis based on the following contributions:

A Systematic Literature Review (SLR) on Annotations in Microservice Construction:

We conduct a comprehensive SLR to investigate the role of annotations in microservice construction [99]. Our review aims to analyse the existing state-of-the-art approaches that

employ static analysis-based techniques to enhance the use of annotations. Through this review, we propose a catalogue of purposes for using annotations and explore the specific uses of annotations that contribute to microservice construction.

An Empirical Study on Microservice Software Development:

In addition to the SLR, we conduct a new empirical study [96] that identifies concerns in microservice software development and presents the relevance of annotations. The study results can help researchers consider new mechanisms for recognising and fixing misuse of annotations. Furthermore, our empirical study enlightens the practical challenges developers face when utilising annotations in microservices. Additionally, our insights offer valuable guidance for practitioners seeking practical solutions for microservice concerns.

A Semantics-Driven Learning for Microservice Annotations:

We develop a novel semantic learning approach [98] to suggest actions for correcting the wrong usage of microservice annotations by extending the implementation of natural language processing for code clone detection. Our approach analyses the annotations of code fragments from open-source repositories. The learning approach pursues annotation prediction after finding similar code fragments that guide the adviser in adding or keeping an annotation. Researchers can benefit from our results and analyse other software engineering features to reduce the complexity of microservice applications.

A Mining Approach to Limit Granularity of Annotated Operations:

We elaborate a new semantic learning approach [97] to mine the relation between annotations and the granularity of operations. The learning process pursues building a mechanism

to measure the granularity based on the semantic information length of operations. Then, we cluster similar annotated operations to facilitate the identification of granularity limits. Operations with different lengths and similar behaviour have different granularity, which provides boundaries for unseen operations. Our findings offer a valuable resources for researchers aiming to determine typical granularity values given the behaviour of similar operations.

1.6 Publications Linked to this Thesis

The research compiled in this thesis is based on three papers published [96, 98, 97] and a systematic literature review [99]. This thesis serves as the primary source of ideas and contributions presented in the following works;

Conferences

- Ramírez, F., Mera-Gómez, C., Bahsoon, R., & Zhang, Y. (2021, June). *An empirical study on microservice software development.* In 2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES) (pp. 16-23). IEEE.

- Ramírez, F., Mera-Gómez, C., Chen, S., Bahsoon, R., & Zhang, Y. (2022, November). *Semantics-Driven Learning for Microservice Annotations.* In Service-Oriented Computing: 20th International Conference, ICSOC 2022, Seville, Spain, November 29–December 2, 2022, Proceedings (pp. 255-263). Cham: Springer Nature Switzerland.

- Ramírez, F., Mera-Gómez, C., Bahsoon, R., & Zhang, Y. (2022, November). *Mining the Limits of Granularity for Microservice Annotations.* In Service-Oriented Computing: 20th

International Conference, ICSOC 2022, Seville, Spain, November 29–December 2, 2022, Proceedings (pp. 273-281). Cham: Springer Nature Switzerland.

Journals

- (To be submitted) Ramírez, F., Mera-Gómez, C., Bahsoon, R., & Zhang, Y. (2024). *Systematic Review of Annotations in Microservice Construction.* ACM Computing Surveys (CSUR).

1.7 Thesis Roadmap

This section presents the remainder of the thesis as structured below.

Chapter 2 conducts an SLR of existing studies about annotations and their impact on microservices construction. This chapter also analyses the connection between annotations and software quality. In particular, it investigates static analysis techniques that support specific uses of annotations. As a result, we construct (i) a catalogue of purposes of using annotations; and (ii) a catalogue of specific uses of annotations that benefit from static analysis techniques. Based on our findings, we advocate an annotations approach to microservice construction and outline future research directions. The content in this chapter is derived from our work presented in [99].

Chapter 3 presents an empirical study that analyses posts from one of the largest sources for developers, StackOverflow. Specifically, we collect the bug symptoms and root causes to link them with the activities in the microservice life cycle, quality attributes, software construction activities, and fault classes. The study investigates the common concerns in microservice development and positions missing/misusing annotations as a top concern.

The content in this chapter is derived from our work presented in [96].

Chapter 4 evaluates a semantics-driven learning approach to identify misuse of annotations by learning from a dataset of code fragments with annotations. In general, we extend the benefit of natural language processing over AST on clone detection to predict which annotation is suitable for a code fragment. The approach finds similar code fragments to generate suggestions for adding or keeping annotations. The content in this chapter is derived from our work presented in [98].

Chapter 5 proposes an approach to mine the relation between annotations and the granularity of operations. The approach measures the granularity based on the semantic information length of operations. Then, we evaluate clusters of similar operations that share levels of granularity. This chapter shows how annotations help to detect the granularity limits of operations by training a learning model with a dataset of similar operations. The content in this chapter is derived from our work presented in [97].

Chapter 6 summarises the main contributions and discusses possible future research directions of annotations in microservice construction and the potential usage of semantics-driven learning for other areas.

Chapter Two

Systematic Review of Annotations in Microservice Construction

2.1 Overview

Annotations facilitate the construction of microservices by adding features without modifying their behaviour. Annotations also help to trace a change in microservice structures and their dependencies. Moreover, the usage of annotations for identifying microservice structures supports the detection of potential issues over time [94, 84]. As the microservice architectural style continues to grow, the utilisation of annotations streamlines the effective construction of flexible, scalable and modular systems [100, 11].

Constructing microservices has become increasingly challenging due to their increased operational complexity and insufficient mechanisms to detect problems [10]. This detection effort is not trivial, considering that comprehension of programs takes around 58% of the time spent on software maintenance [51]. Therefore, microservice construction calls for an exploration of the landscape of annotations usages to detect and address development and maintenance issues [138]. This exploration becomes more relevant since the misuse of

annotations is at the top of concerns for microservice development [96].

This Systematic Literature Review (SLR) collects the different purposes for which the authors of primary studies utilised annotations and their relevance in microservices. The study gathers purposes and specific uses of annotations as guidelines to discuss the benefits of static analysis techniques. This chapter contributes to (i) a catalogue outlining the various purposes of annotation used in microservice construction and (ii) an overview of the relation between purposes and specific uses of annotations, including the static analysis techniques observed in studies. Both contributions provide a comprehensive collection of how researchers incorporate annotations when performing microservice development and maintenance.

From the perspective of annotations in microservices, previous works have contributed to detect the scenario of missing annotations [7, 138]. Additionally, some studies have focused on static analysis-based techniques and annotations by investigating graph theory, machine learning, deep learning, and application of rules [14, 33, 65]. Additionally, one previous study have discussed the evolution and usage of annotations without categorising the techniques [138]. However, the previous studies have missed the purpose and specific use of annotations which are part of microservice construction, despite using contextual annotations [110, 149, 81]. Those studies are also missing techniques based on machine learning that use annotations. In contrast, this work performs a catalogue of purposes and their relation to the specific uses of annotations that match static analysis-based techniques.

The remainder of this chapter is organised as follows. Section 2.2 introduces the research methodology with the protocol, goal, research questions, search strategy and data synthesis. Section 2.3 presents the outcomes of this work. Section 2.4 discusses findings, future research and threats to validity, followed by a gap analysis in Section 2.5. Section 2.6 situates our work within the related literature. Finally, Section 2.7 concludes the chapter.

2.2 Research Methodology

This study was guided by the systematic review procedures proposed by Kitchenham et al. [62]. We also examined the studies of Bhuiyan et al. and Pan et al. [17, 88] that contain procedures for publication-quality assessment in a similar research topic. We aimed to increase the probability of producing an unbiased study on the field by selecting representative articles with a perspective towards annotations in microservice construction. The following subsections describe the details of the adopted methodology.

2.2.1 Review Protocol

To systematically conduct our literature review, the protocol consists of the following components: (i) background of research to motivate our SLR; (ii) identification of research to define research questions; (iii) selection of searching platforms to search prior research; (iv) definition of inclusion and exclusion criteria to compile papers; and (v) selection procedure of studies to filter retrieved entries with the most relevant papers.

2.2.2 Research Goal

The high-level goal of this research is to investigate the specific uses and purposes of using annotations in the construction of microservices, particularly within the context of academic literature. We aim to systematically collect and classify the existing empirical evidence from academic papers that discuss annotations and static analysis-based techniques in microservices. Additionally, in response to the division of RQ1 from the thesis, we focus our efforts on the two initial parts of RQ1.

2.2.3 Research Questions

We intend to analyse the mechanisms that utilise the source code of systems during construction of microservices and their application with annotations by summarising the relevant purposes and uses of annotations. To achieve this, we identify all relevant studies to answer the following clear and focused Research Questions (RQ):

RQ1.1: *What are the purposes of using annotations in microservice construction?*

Researchers benefit from annotations by adding characteristics without changing the behaviour of operations. Annotated operations contain information of structures and connections between microservices. The dynamic nature of microservices changes the structures and therefore may impact system performance over time. Thus, annotations assist researchers in understanding the structure of microservices. The objective of RQ1.1 is to investigate the purpose for the utilisation of annotations in the context of microservice construction.

RQ1.2: *How do the static analysis-based techniques support the purpose of using annotations?*

The static analysis-based techniques support the purpose of using annotations through an assisted reorganisation of microservice operations that we categorise as the specific uses of annotations. Academic papers make specific uses of annotations such as identifying annotations to evaluate architectures, adding annotations to refactor microservices, verifying annotations to detect vulnerabilities, among others. RQ1.2 aims to collect the landscape of specific uses of annotations and the adopted static analysis-based techniques.

Additionally, answering both research questions benefits researchers in enhancing their understanding of the connections between annotations, their purpose of usage, and the techniques applied for each specific use of annotations. Moreover, by comprehensively exploring these aspects, researchers can identify potential gaps in current practices and propose novel

approaches to address them.

2.2.4 Literature Search Strategy

Search Criteria

We conducted the SLR with the following search query that retrieves entries of papers with their publication title, year, abstract and keywords: *(Microservice OR service) AND (development OR maintenance OR construction) AND (annotation OR annotations) AND ("static analysis")*. The term development avoided missing studies that refer to annotations. We also performed a manual search to check for any missed paper. This query informed publications by experience and cross-referencing, considering the citations of seminal papers. The insights gained in this study help to address subsequent research questions.

Search Platforms

We identified and selected the relevant digital repositories to ensure that the search terms would yield studies related to annotations in microservice development. Table 2.1 lists the names and links of representative repositories for software engineering research [13].

Table 2.1: Digital Repositories for SLR

Digital repository	**Link**
IEEEXplore	https://ieeexplore.ieee.org/
ACM Digital Library	https://dl.acm.org/
Springer	https://link.springer.com/
Science Direct	https://www.sciencedirect.com/
Scopus	https://www.scopus.com
Web of Science	https://www.webofscience.com/wos/woscc/basic-search

Study Selection Criteria

To ensure that the studies selected for our systematic literature review were relevant and of high quality, we built the following Table 2.2 with (i) *inclusion criteria* for the search process based on the electronic databases; and (ii) *exclusion criteria* to filter the irrelevant studies for the objective of our SLR. The inclusion and exclusion criteria consider when microservices appeared in GitHub and when annotations appeared in Java Specification Request JSR-175.

Table 2.2: Inclusion and Exclusion Criteria for Study Selection

	Inclusion criteria (Is)		**Exclusion criteria (Es)**
I1	Papers published in journal, conference, workshop, report, or book chapter.	**E1**	Papers in another language than English.
I2	Papers explicitly related to service even though they do not refer to code annotations.	**E2**	Papers with similar title or content published in different venues are duplicated entries.
I3	Papers extended software engineering practices for annotations.	**E3**	Papers published without providing sufficient pages like short papers with less than 8 pages.
I4	Paper discussing aspects influencing annotations.	**E4**	Papers in disciplines different from computer science.
		E5	Papers published before September 2004.

Study Selection Procedure

The selection procedure involves executing the criteria, performing this procedure, and determining the relation between the paper and the studies [62]. The publications collected with the selection criteria from the digital databases may contain irrelevant entries. The removal of results that were out of the scope of our study required applying the inclusion and exclusion criteria in three rounds. In the first round, we collected the metadata of papers to choose those papers with titles belonging to our full scope. The second round filtered the papers whose abstracts belong to our scope. The third round required a full-text reading to filter papers related to the scope of annotations. Finally, we identified additional papers with a snowballing strategy which utilises citations (forwards) and reference lists (backwards)

[129]. Figure 2.1 presents the number of papers for each round after following the inclusion and exclusion criteria applied in each step.

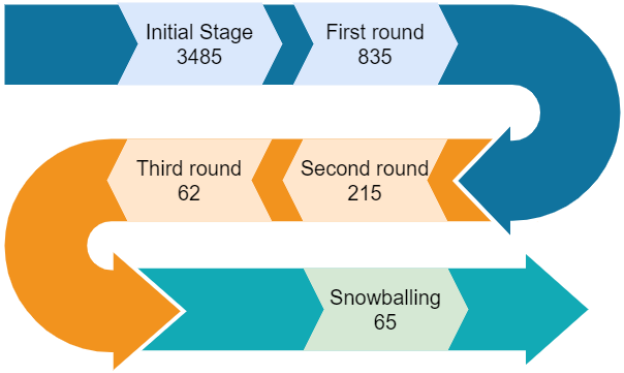


Figure 2.1: Collection of Papers Metadata.

2.2.5 Publication Quality Assessment

After selecting the publications, we evaluate the quality of primary studies by assessing their relevance to the objective of this SLR. Our quality questions follow the criteria proposed by Kitchenham et al. for quality assessment [63]. Table 2.3 presents the quality questions that guide our assessment. Each question may have answered with either *No*, *Partially* or *Yes*. If there is implicit information that could be derived from the text, the question has a *Partially* score. We score each question with values from 0 to 5, where No is 0, 3 is Partially, and 5 is Yes. The quality score of each selected primary study sums all the scores of each question [1].

2.2.6 Data Extraction and Synthesis

Table 2.4 presents the relevant items to consider for answering the research questions. The primary demographic information to extract from each study is the search platform, title,

[1]This assessment is to know the quality of our findings and not to exclude any paper.

Table 2.3: Quality Questions for Paper Assessment

Design	Is the paper based on empirical research with a clearly stated study context?
Aim	Does the paper have a clear description and presentation of the research objectives?
Validity	Does the methodology successfully approach the objectives of the research?
Data collection and analysis	Does the paper fully describe the data collection method and the data analysis?
Clarity	Does the paper have a clear description and presentation of the results?
Limitations	Do the researchers include threats to validity or limitations of the results?

keywords, year and venue. Additionally, the purpose of using annotations and type of static analysis technique help answer RQ1.1, and the specific use of annotations help answer RQ1.2.

Table 2.4: Data Extraction

Item	**Association**
Search platform of the study	Demographic
Title and keywords of the study	Demographic
Year and venue of the study	Demographic
Purpose of using annotations	RQ1.1
Type of static analysis-based techniques	RQ1.1
Specific use of annotations	RQ1.2

We took the software development categories as a guide for classification and included relevant processes related to microservices. We classified by following specific purposes for using annotations: architecture evaluation, bad-smell detection, vulnerability detection, microservice identification, defect prediction, performance assessment, access control, refactoring, and others.

For answering RQ1.1, the purposes of using annotations will be classified by similar characteristics related to microservice construction. The catalogue of purposes will aid in explaining their connection. A catalogue of static analysis-based techniques is required to

collect researchers concerns. For answering RQ1.2, we group similar techniques and match them with the purposes of using annotations while comparing the specific use for identifying annotation against the others to elaborate an approach toward annotations in microservice construction.

For the sake of the replicability of our study, a replication package [2] is available for interested readers. The package includes the research protocol, the selected papers, the extracted and categorised data, and the scripts for generating the information charts.

2.3 Results

This section describes the primary studies and shows the results of this SLR in light of the research questions. By analysing a comprehensive set of primary studies, this section provides valuable insights into the diverse purposes and uses of annotations that facilitate the development and maintenance of microservices.

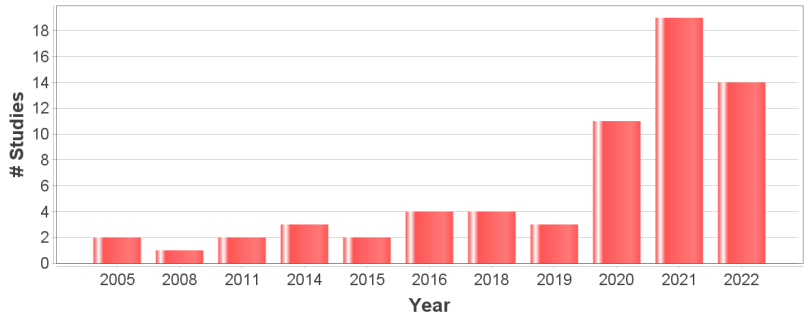


Figure 2.2: Distribution of Studies per Year.

[2]http://www.research.propio.click/paper-slr/replication-package/

2.3.1 Description of Studies

This SLR collects 65 primary studies and describes the publications from the perspective of time and source.

Publication Time

Figure 2.2 presents the number of studies per year from 2005 to 2022. The chart represents the distribution of primary studies across different years and shows an increasing trend with a significant rise in 2021. From this Figure, we can observe that annotations in microservice construction as a hot topic for the last few years.

Publication Source

Table 2.5: Number of Studies per Publication Name (Top 10).

ID	Publication Name	Type	No.
1	IEEE Access	Journal	4
2	International Conference on Software Engineering	Conference	3
3	Journal of Systems and Software	Journal	3
4	Information and Software Technology	Journal	3
5	International Conference on Software Engineering: Software Engineering in Practice Track	Conference	3
6	Transactions on Software Engineering and Methodology	Journal	2
7	Symposium on Applied Computing	Conference	2
8	Journal of Systems Architecture	Journal	2
9	Information Science and Applications	Conference	2
10	International conference on Software Architecture	Conference	2

Table 2.5 shows the top 10 publication names with the publication type and the number of studies. This top 10 list selects the publication names with more than one article, containing 42% of all primary studies. This table shows that journals contains more articles than conferences and represents 54% of the top 10 list. The International Conference on

Software Engineering (ICSE) and Software Engineering in Practice Track (ICSE-SEIP) are two top conferences with six papers. The ASE conference has just two papers; however, there are no papers from ICSOC and ICWS, which are conferences related to services. We observe that annotations stabilise in the software engineering context during 2021, while service conferences focus on other contexts.

Table 2.6 presents the type of studies (conferences, journals, thesis, book chapters) per publisher. IEEE is the publisher with the most amount of studies. It has 13 conference papers plus four journal articles, which is a total of 17 studies. Springer is the second top publisher with 13 studies, where eight are conference papers, four are journal articles and one book chapter. ScienceDirect, ACM and ACM Journal are the other publishers, with 13, nine and four studies. This top five publisher contains 82% of all primary studies.

Table 2.6: Number of Studies per Publisher.

ID	Publisher	Conference	Journal	Thesis	Book Chapter	Total
1	ACM	9				9
2	ACM Journal		4			4
3	Hindaw		1			1
4	IEEE	13	4			17
5	MDPI		1			1
6	Peerj		1			1
7	ScienceDirect		10			10
8	SciTePress	1				1
9	Springer	8	4		1	13
10	University			5		5
11	Usenix	1				1
12	Wiley		2			2

2.3.2 Description of Categories

After providing the distribution of primary studies, the focus shifts to the diverse purposes of using annotations in microservices. Contextual annotations are the text introduced into comments, commits and software documentation, which categorise and provide valuable

information for vulnerability detection, refactoring, and others. This study emphasises code annotations, specifically Java Annotations, that are metadata added to code fragments. Thus, annotations could bind to classes, fields, methods, and parameters.

A catalogue of purposes extracted from the primary studies is available in Table 2.7, which matches the purposes of using annotations in the context of development and maintenance. The provided catalogue offers a valuable resource for practitioners and researchers

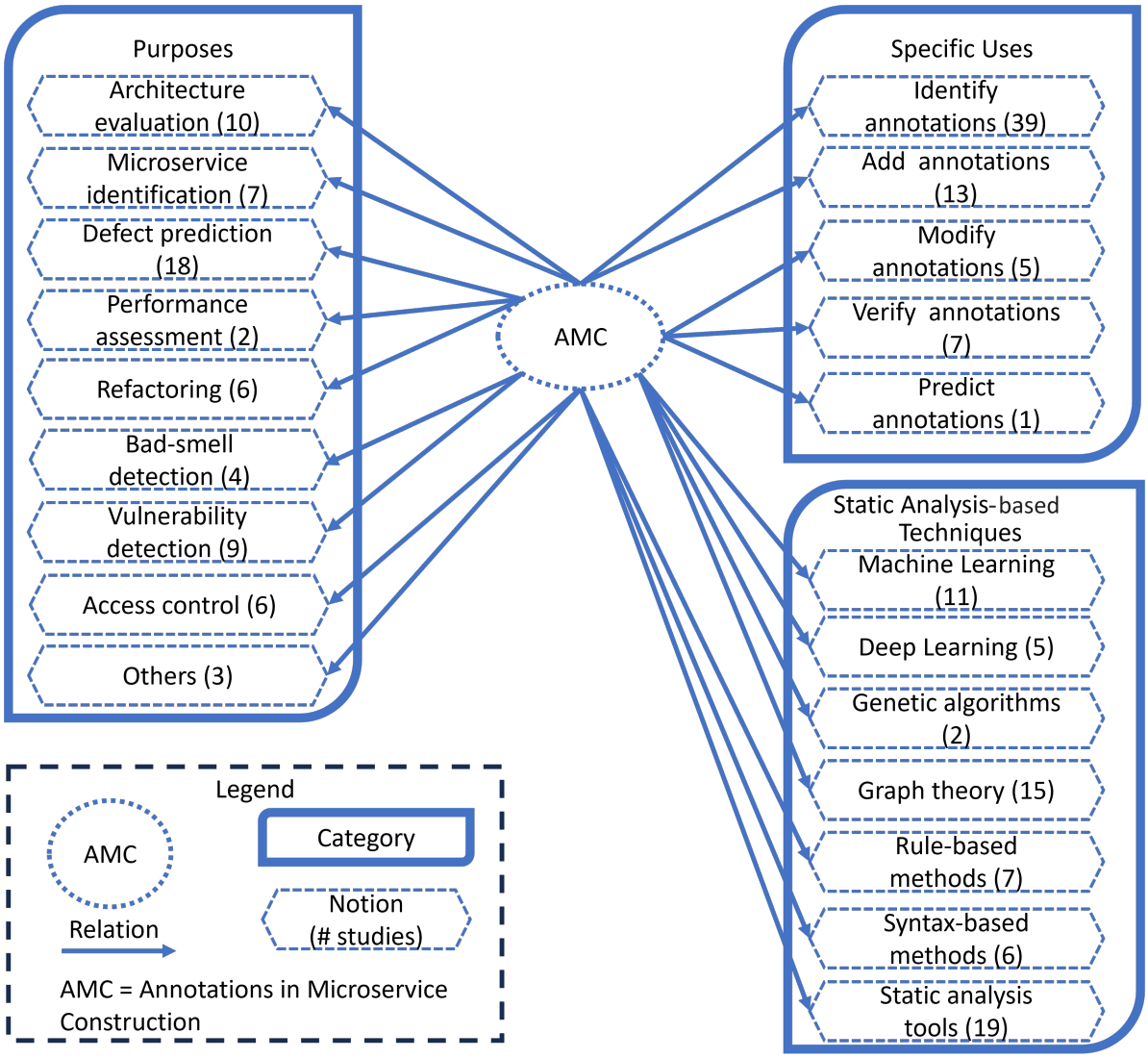


Figure 2.3: Distribution of Studies per Categories.

to guide the decision-making for effective microservice construction. Figure 2.3 presents the studies grouped by their purposes, specific uses and static analysis-based techniques.

Two related purposes that contribute to the improvement of modularisation and scalability during microservice construction are: (i) architecture evaluation aims to assess the overall architecture and identify design decisions that may have been lost during construction; and (ii) microservice identification involves clustering business entities and analysing their coupling, cohesion and microservice metrics.

Additionally, four purposes that ensure the reliability and code quality of microservices are: (iii) defect prediction leads to a more reliable and robust system and aims to detect patterns or indicators that may affect the quality attributes of a system; (iv) performance assessment covers the evaluation of various performance metrics that improve the reliability of the system through the optimisation of critical components; (v) bad-smell detection involves identifying architectural smells, which may indicate potential architectural degradation; and (vi) refactoring involves detecting antipatterns and providing recommendations for improving the code quality.

Regarding security reasons: (vii) vulnerability detection focuses on identifying security weaknesses to mitigate potential breaches through actively scanning methods; and (viii) access control evaluates protection mechanisms of sensitive data through a proper authorisation and authentication process that ensures the appropriate privileges for authorised users. Other categories encompasses documentation, testing, and library compatibility, which involve development and maintenance processes for microservices.

Table 2.8 presents the specific uses of annotations in microservice construction. Annotations offer a versatile set of specific uses that researchers can perform on code fragments to enhance their functionality and analysis. The static analysis-based techniques considers the execution of these specific uses of annotations to accomplish the above purposes.

Table 2.7: Catalogue: Purpose of Using Annotations.

Category	**Purpose**	**Description**	**Studies**
Modularisation and scalability	Architecture evaluation	Assess the architecture to detect lost architectural design decisions.	[10] [74] [140] [25] [32] [126] [23] [114] [9] [16]
	Microservice identification	Cluster business entities to improve coupling, cohesion and microservice metrics.	[44] [31] [21] [139] [100] [40] [36]
Reliability and code quality	Defect prediction	Detect patterns that may affect the quality attributes.	[117] [83] [12] [59] [69] [65] [29] [84] [94] [75] [14] [113] [104] [35] [71] [72] [46] [27]
	Performance assessment	Analyse the performance to identify areas for improvement.	[134] [22]
	Refactoring	Detect antipatterns to provide recommendations for improving the code quality.	[133] [5] [11] [6] [64] [52]
	Bad-smell detection	Identify architectural smells to prevent degradation.	[107] [124] [125] [13]
Security	Vulnerability detection	Detect security weakness to mitigate potential security breaches.	[111] [142] [88] [38] [78] [127] [17] [39] [26]
	Access control	Evaluate mechanisms to ensure proper authorisation and authentication processes.	[7] [33] [106] [66] [67] [80]
Others	Others (documentation, testing and library compatibility)	Refer to a comprehensive understanding of source code through good documentation and effective implementation of annotations by testing and checking library compatibility.	[144] [56] [50]

Specific uses to effectively manage and comprehend code structure are: (i) identify annotations to use search criteria for locating entities, methods or components; (ii) add annotations to some specific part of code; (iii) verify annotations to check constraints in the usage of annotations; (iv) modify annotations to remove or change their parameters or the values of parameters of annotated code; and (v) predict annotations to assigning one annotation to a new code fragment.

Table 2.8 also contains the category for static analysis-based techniques in microservice construction, which is based on the following techniques: (i) machine learning; (ii) deep learning; (iii) graph theory; (iv) genetic algorithms; (v) syntax-based methods; (vi)

Table 2.8: Catalogue: Specific Uses of Static Analysis-based Techniques.

Specific use	**Static analysis-based technique**	**Studies**
Identify annotations	Machine learning	[44] [66] [59] [69] [65] [139] [13] [100] [36]
	Deep learning	[21]
	Genetic algorithm	[144] [52]
	Graph theory	[88] [124] [67] [25] [32] [126] [23] [127] [80] [14] [16] [72]
	Syntax-based method	[107] [12] [50] [40]
	Rule-based method	[11] [114] [39]
	Static analysis tools	[10] [7] [106] [125] [29] [5] [9] [26]
Add annotations	Machine learning	[31] [140]
	Deep learning	[111] [33]
	Graph theory	[133] [78] [71]
	Static analysis tools	[142] [117] [17] [22] [46] [27]
Modify annotations	Syntax-based method	[75]
	Rule-based method	[74] [94]
	Static analysis tools	[6] [64]
Predict annotations	Deep learning	[134]
Verify annotations	Deep learning	[38]
	Syntax-based method	[104]
	Rule-based method	[59] [35]
	Static analysis tools	[56] [84] [113]

rule-based methods; and (vii) static analysis tools.

Machine learning, which uses traditional statistical methods to analyse the source code [31]; deep learning, which employs artificial intelligence based on neural networks for patterns detection [21]; graph theory, which searches paths between the nodes of graph structure for software architecture reconstruction [23]; genetic algorithms, which employ the principle of natural selection to improve code maintainability [52]; syntax-based methods, which use abstract syntax tree to identify patterns [75]; rule-based methods, which check constraints to detect changes in architecture [114]; and static analysis tools, which consolidate various techniques for conducting general code analysis [106].

Table 2.9 provides a comprehensive overview of the relations between purposes and specific uses of annotations. It includes the static analysis-based techniques observed in the primary studies; it also shows a variety of purposes for using annotations in microservice construction. Each purpose is linked to the specific use of annotations with the static analysis-based technique employed to achieve the purpose. Compiling the purposes, specific uses and static analysis-based techniques highlights the different aspects of annotations and their relation in microservice construction.

2.3.3 Annotations in Microservices Construction

Answer to RQ1.1: What are the purposes of using annotations in microservice construction?

The catalogue of purposes reveals that *defect prediction* is the top category with 28% followed by *architecture evaluation* (15%), *vulnerability detection* (14%), *microservice identification* (11%), *refactoring* (9%), *access control* (9%). The last three categories are bad-smell detection, performance assessment and others.

Table 2.9: Annotations in Microservice Construction

Study	**Specific Use**	**Purpose**	**Static Analysis-based Technique**
Cuomo et al. [29]	Identify annotations	Defect prediction	Tools
Bakhtin et al. [14]	Identify annotations	Defect prediction	Graph theory
Ma et al. [72]	Identify annotations	Defect prediction	Graph theory
Kim et al. [59]	Identify annotations	Defect prediction	Machine learning
Lopes et al. [69]	Identify annotations	Defect prediction	Machine learning
Laigner et al. [65]	Identify annotations	Defect prediction	Machine learning
Araujo et al. [12]	Identify annotations	Defect prediction	Syntax-based method

Continued on next page

Table 2.9 Annotations in Microservice Construction *(Continued from previous page)*

Study	Specific Use	Purpose	Static Analysis-based Technique
Apolinario et al. [10]	Identify annotations	Architecture evaluation	Tools
Alshuqayran et al. [9]	Identify annotations	Architecture evaluation	Tools
Cerny et al. [25]	Identify annotations	Architecture evaluation	Graph theory
Das et al. [32]	Identify annotations	Architecture evaluation	Graph theory
Walker et al. [126]	Identify annotations	Architecture evaluation	Graph theory
Bushong et al. [23]	Identify annotations	Architecture evaluation	Graph theory
Bersani et al. [16]	Identify annotations	Architecture evaluation	Graph theory
Streekmann et al. [114]	Identify annotations	Architecture evaluation	Rule-based method
Chowdhury et al. [26]	Identify annotations	Vulnerability detection	Tools
Pan et al. [88]	Identify annotations	Vulnerability detection	Graph theory
Wang et al. [127]	Identify annotations	Vulnerability detection	Graph theory
Ferrara et al. [39]	Identify annotations	Vulnerability detection	Rule-based method
Qiwen Gu [44]	Identify annotations	Microservice identification	Machine learning
Zaragoza et al. [139]	Identify annotations	Microservice identification	Machine learning
Ren et al. [100]	Identify annotations	Microservice identification	Machine learning
Escobar et al. [36]	Identify annotations	Microservice identification	Machine learning
Freitas et al. [40]	Identify annotations	Microservice identification	Syntax-based method
Brito et al. [21]	Identify annotations	Microservice identification	Deep learning
AlOmar et al. [5]	Identify annotations	Refactoring	Tools
Arachchi [11]	Identify annotations	Refactoring	Rule-based method
Ivers et al. [52]	Identify annotations	Refactoring	Genetic algorithm
Alshemaimri et al. [7]	Identify annotations	Access control	Tools
Scherzinger et al. [106]	Identify annotations	Access control	Tools
Meurice et al. [80]	Identify annotations	Access control	Graph theory
Li et al. [67]	Identify annotations	Access control	Graph theory
Le et al. [66]	Identify annotations	Access control	Machine learning
Walker et al. [125]	Identify annotations	Bad-smell detection	Tools
Walker et al. [124]	Identify annotations	Bad-smell detection	Graph theory
Azeem et al. [13]	Identify annotations	Bad-smell detection	Machine learning
Schiewe et al. [107]	Identify annotations	Bad-smell detection	Syntax-based method

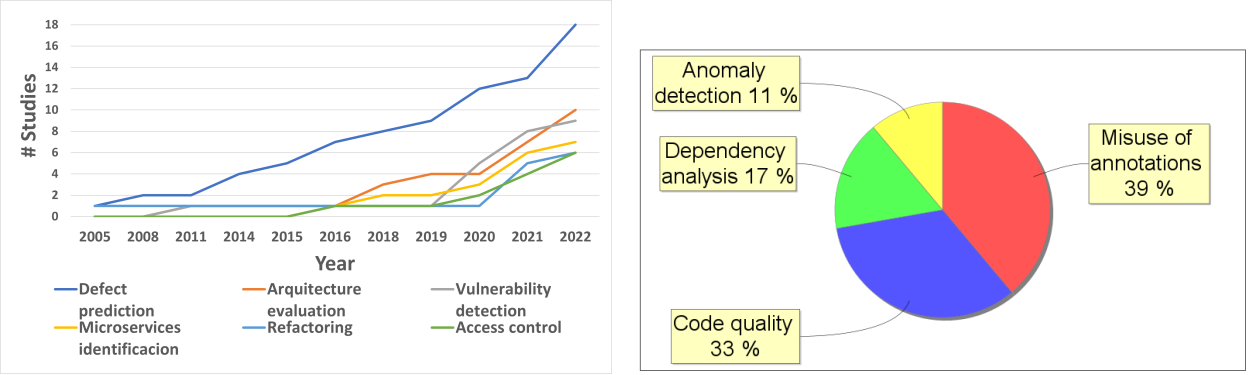
Continued on next page

Table 2.9 Annotations in Microservice Construction *(Continued from previous page)*

Study	**Specific Use**	**Purpose**	**Static Analysis-based Technique**
Hobmaier [50]	Identify annotations	Documentation (Others)	Syntax-based method
Zhang et al. [144]	Identify annotations	Testing (Others)	Genetic algorithm
Tang et al. [117]	Add annotations	Defect prediction	Tools
Christakis et al. [27]	Add annotations	Defect prediction	Tools
Gunawi et al. [46]	Add annotations	Defect prediction	Tools
Ma et al. [71]	Add annotations	Defect prediction	Graph theory
Zdun et al. [140]	Add annotations	Architecture evaluation	Machine learning
Zhang et al. [142]	Add annotations	Vulnerability detection	Tools
Bhuiyan et al. [17]	Add annotations	Vulnerability detection	Tools
Andrea Melis [78]	Add annotations	Vulnerability detection	Graph theory
Shen et al. [111]	Add annotations	Vulnerability detection	Deep learning
Daoud et al. [31]	Add annotations	Microservice identification	Machine learning
Xu et al. [133]	Add annotations	Refactoring	Graph theory
Del Alamo et al. [33]	Add annotations	Access control	Deep learning
Bureš et al. [22]	Add annotations	Performance assessment	Tools
Nuryyev et al. [84]	Verify annotations	Defect prediction	Tools
Spoto, Fausto [113]	Verify annotations	Defect prediction	Tools
Noguera et al. [83]	Verify annotations	Defect prediction	Rule-based method
Eichberg et al. [35]	Verify annotations	Defect prediction	Rule-based method
Sadowski et al. [104]	Verify annotations	Defect prediction	Syntax-based method
Feng et al. [38]	Verify annotations	Vulnerability detection	Deep learning
Jezek et al. [56]	Verify annotations	Library compatibility (Others)	Tools
Pinheiro et al. [94]	Modify annotations	Defect prediction	Rule-based method
Marcilio et al. [75]	Modify annotations	Defect prediction	Syntax-based method
Kollegger [74]	Modify annotations	Architecture evaluation	Rule-based method
AlOmar et al. [6]	Modify annotations	Refactoring	Tools
Ksontini et al. [64]	Modify annotations	Refactoring	Tools
Yang et al. [134]	Predict annotations	Performance assessment	Deep learning

Figure 2.4a focuses on the six significant purposes. It reveals how the number of studies increased over time. Defect prediction begins with a constant increment of nearly 1 study per year from 2005 to 2022. Vulnerabilities and refactoring are stable until 2020 and 2021, respectively, when they increase near the architecture evaluation and microservice identification. Figure 2.4b provides a detailed subcategory of defect prediction, which encompasses four types of defects: misuse of annotations (39%), code quality (33%), dependency analysis (17%), and anomaly detection (11%). Thus, misusing annotations results in inaccurate behaviour affecting the functioning of the system and its overall quality.

Figure 2.5 presents the number of studies for each purpose by grouping the studies into (i) microservices and (ii) services. Then, comparing these two groups helps to observe the importance and evolution of the usage of annotations in microservices.



(a) Top Purposes Over Time.

(b) Defect Prediction.

Figure 2.4: Distribution of Purposes for Using Annotations.

If we consider microservices, then *architecture evaluation* and *microservice identification* are the top purposes, instead of *refactoring* is near the last place. However, *defect prediction* and *vulnerability detection* emerge as the top purposes if we group the studies considering their importance to the service context.

Microservices group shows us a clear trend of purposes for using annotations in the

studies. Architecture evaluation and microservice identification are at the top, with sixteen studies. For instance, some architecture evaluation studies consider reverse engineering techniques to discover the changes in the initial concept. Then, defect prediction and bad-smell detection have ten studies. On the other hand, vulnerability detection and refactoring reduces up to two studies each, taking the last position of all purposes.

Services group provides evidence that defect prediction is the top reason with 11 studies. Vulnerability detection follows second place with seven studies. The research community prefers defect prediction instead of architecture evolution in the context of services. Refactoring with four studies is also preferred in the services group.

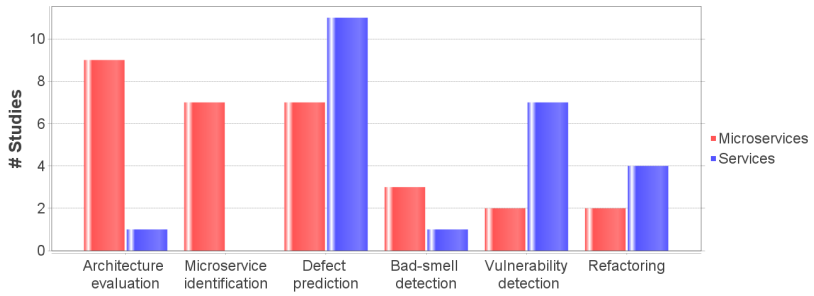


Figure 2.5: Distribution of Purposes Grouped by Context.

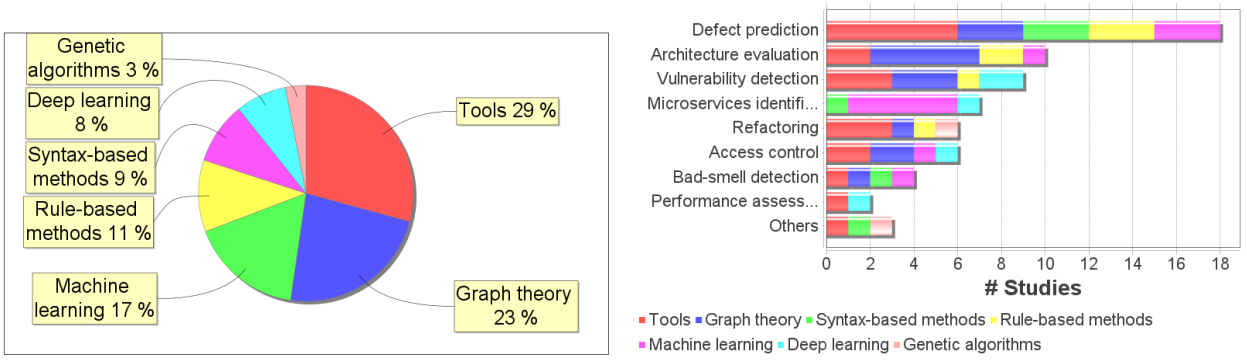
2.3.4 Specific Uses in Static Analysis-based Techniques

Answer to RQ1.2: How the static analysis-based techniques support the purpose of using annotations?

Our work also classifies the primary studies by the following static analysis-based techniques (SAT): tools (29%), graph theory (23%), machine learning (17%), rule-based methods (11%), syntax-based methods (9%), deep learning (8%) and genetic algorithms (3%). Figure 2.6a shows the percentage of studies per category of static analysis-based techniques. Since tools contain different techniques, we analyse the other categories and

their relation with purposes.

The relation between static analysis-based techniques and the purposes of using annotations is visible in Figure 2.6b, which presents the distribution of static analysis-based techniques. In comparison, the tools category has more studies in defect prediction. Machine learning has more presence in microservice identification. Graph theory is present in architecture evaluation, defect prediction and vulnerability detection. Deep learning appears in vulnerability detection, microservice identification, access control and performance assessment.



(a) Static Analysis-based Techniques and Tools.

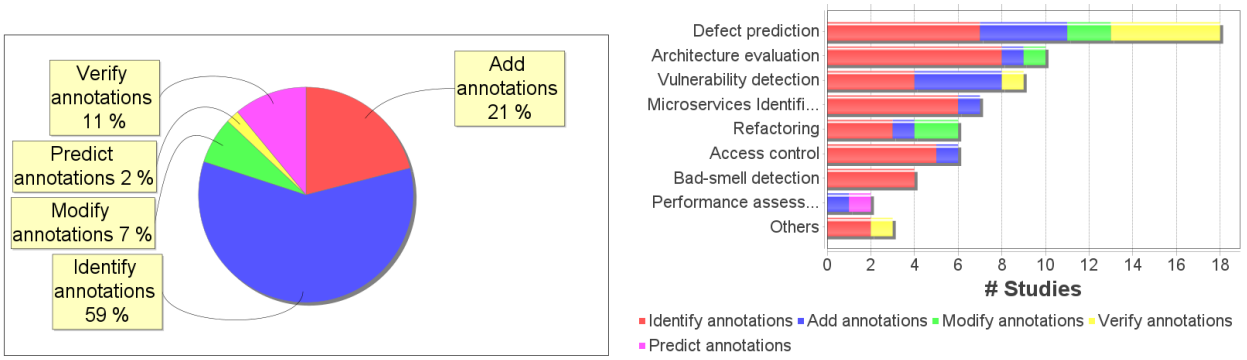
(b) Techniques in Purposes.

Figure 2.6: Static Analysis-based Techniques and Tools in Purposes.

Figure 2.6b also presents interesting results of the relation between static analysis tools and purposes for using annotations. Defect prediction is the top purpose, with 30% of studies. Refactoring and vulnerability detection have 16% of studies each. Architecture evaluation and access control have 11% of studies. Bad-smell detection, performance assessment and others are in the last position with 5% of studies each. Then, solving issues emerges as the main purpose when using static analysis tools.

Near 6.55% of primary studies are focus on (i) migration to microservices by gathering system documentation and inspecting the source code without considering the annotations [44]; (ii) detection of code smells by using contextual annotation instead of code annotation

[13]; and (iii) extraction of keywords from commit messages related to refactoring [6] and from contextual annotations added by tools instead of code annotations [26].



(a) Specific Uses of Annotations

(b) Specific Uses by Purposes.

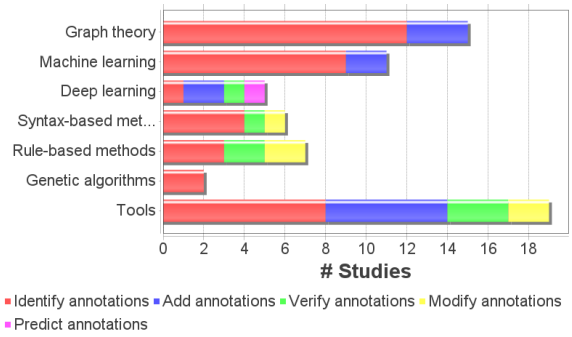
Figure 2.7: Specific Uses of Annotations and their Relation in Microservice Construction.

Figure 2.7a shows that identifying annotations is a top specific use in microservices, with around 59% of the studies. Identifying annotations helps with coupling metrics [10], data entity detection [144, 107, 80], communication detection [12, 72], feature classification [69], and dependency detection [5, 36]. Adding annotations is the second top specific use with 21% of the studies. This specific use helps migration to microservice architecture by exposing new operations, creating new services [31], suppressing warnings for reliability errors [27], monitoring performance [22], and measuring the impact of API changes [56]. Modifying annotations appears in nearly 7% of studies with the following scenarios: (i) changing the values of parameters for basic shell script practices [64, 75]; (ii) removing annotations from a code fragment [74]; and (iii) changing annotations to inject faults for testing [94].

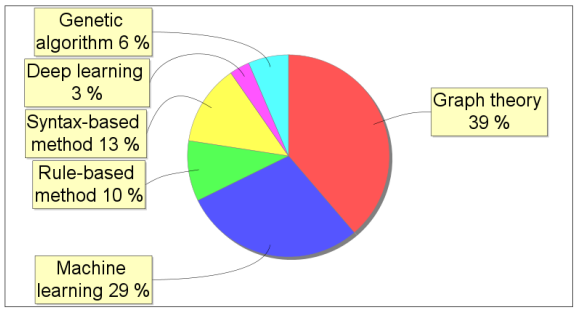
In the case of verifying annotations, we can mention that (i) domain models support the validation of annotation frameworks [83]; (ii) system security benefits from detecting password leaks [38, 35]; and (iii) data race detection is also possible by verifying the annotation constraints [113, 104]. Thus, verifying annotation is the third top specific use in 11% of the primary studies. Predicting annotation has fewer studies by nearly 2%, where Deep

Neural Networks (DNN) help learn semantics from source code to find patterns, predict function types, and recommend the most relevant code for solving issues [134].

Figure 2.7b presents how identifying, adding, modifying, verifying and predicting annotations are considered during microservice construction. The second most specific use, adding annotation, occurs mainly in defect prediction and vulnerability detection to solve issues related to missing annotations. The next top specific use is verifying annotation, mainly in defect prediction. We notice that modifying annotation occurs in defect prediction and refactoring in the same proportion. We infer that refactoring changes source code to create new microservices, while defect prediction solves missing annotations.



(a) Specific Uses in Static Analysis-based Techniques.



(b) Identifying Annotations in Static Analysis-based Techniques.

Figure 2.8: Specific Uses of Annotations and Static Analysis-based Techniques.

Figure 2.8a presents the specific uses related to static analysis-based techniques. *Adding annotation* appears in all the static analysis-based techniques except syntax-based and rule-based algorithms, which are mainly related to *verifying annotation*. Deep learning also have a small presence for *predicting annotation* and *verifying annotation*. Additionally, Figure 2.8b presents *identifying annotations* and its relation to static analysis-based techniques. Graph theory are the top category with 39%, followed by machine learning with 29%. Syntax-based occupied the third place with 13%.

2.4 Discussion

Based on the review, specific uses closely connect to the purposes for using annotations and static analysis-based techniques. This review identified the following major findings:

Defect prediction is the top purpose for microservices and services with a difference of four studies on favor of services. This distinction implies a different set of purposes for comparing their specific uses of annotations and their static analysis-based techniques. Our review showed that tools and rule-based methods are mainly for verifying annotations.

Also, defect prediction is mainly for solving misuse annotations and code quality issues. Misuse annotations appear first, then dependencies detection and code quality in microservices. At the same time, code quality is only one study greater than misuse annotations, and there is only one study for dependencies detection in services.

Finding 1: *Defect prediction for detecting misuse of annotations is the most popular purpose for using annotations. Results suggest that mining annotations and their constraints support the validation of annotations.*

***Implication**: Researchers need to fix misusing annotations, which may contribute to the reliability of microservices by ensuring that annotations correspond to the functionality.*

Although defect prediction is a top purpose, architecture evaluation and vulnerability detection emerge when researchers require to identify annotations. In those cases, researchers prefer to use graph theory, however the usage of static analysis tools are mainly for adding annotations. Graph theory positions vulnerability detection in the third position, where adding annotations is mainly used instead of verifying annotations. Thus, the advice is to research more verifying annotations for vulnerability detection.

Addressing misusing annotations is essential for successfully implementing microservices. Annotated code serves as a reference for understanding the behaviour and depen-

dencies of microservices. Researchers facilitate the smooth transition and integration of microservices by ensuring that annotations are used appropriately. Thus, fixing misused annotations enhances the reliability of microservices and plays a significant role in the successful execution of defect prediction, architecture evaluation and vulnerability detection.

Finding 2: *Architecture evaluation is the second most popular purpose of using annotations when comparing graph theory with other static analysis-based techniques. Vulnerability detection follows closely when researchers use tools and graph theory. However, verifying annotations is underrepresented in these two purposes considering its importance in the top purpose.*

***Implication**: The lack of focus on verifying annotations in architecture evaluation and vulnerability detection indicates a gap in current research. Researchers need to investigate the potential of machine learning for verifying annotations, which could enhance the architecture evaluation and vulnerability detection and lead to more robust and secure microservices.*

Vulnerability detection and refactoring appear as second and third purposes that improve the microservice when comparing the purposes with all the uses of annotations except for identifying annotations. Prioritising the research in deep learning during refactoring helps reduce the gap of missing the specific use of verifying annotations. Thus, the recommendations are (i) including more research on verifying annotation, (ii) explicitly reducing the usage of tools, and (iii) increasing deep learning for refactoring.

It is important to note that predicting annotation is excluded from the recommendation due to the specific goals of refactoring, which aim to improve the existing structure without introducing new features. Therefore, the recommendation focuses on the enhancement of verifying annotations, specifically through the integration of deep learning, which requires a significant amount of data for training, testing and validation.

Finding 3: *Identifying annotations is the top specific use. In this line, researchers prefer tools for defect prediction. However, results suggest that adding annotations requires more deep learning than tools, especially for vulnerability detection.*

***Implication**: Researchers need to investigate deep learning and incorporate it into the tools, especially for improving vulnerability detection. Moreover, access control benefits from deep learning, considering its relation to security enhancement.*

Graph theory is the second static analysis-based technique without considering the preference for static analysis tools. Graph theory is mainly applied to architecture evaluation, defect prediction, and vulnerability detection to improve microservices. We recommend incorporating graph theory when verifying annotation. By integrating graph theory into the tooling landscape, researchers may gain access to advanced analytical techniques that reveal the relations within microservices. This integration would enable more comprehensive verification of annotation, addressing its lack in the architecture evaluation of microservices.

Graph theory may significantly impact two middle purposes: ensuring microservice identification and refactoring. These purposes are critical and strongly consider identifying annotations but miss predicting and verifying annotations. Adopting graph theory with verifying annotations can play a pivotal role in strengthening the identification of microservices. These techniques can aid in recognising potential structures, analysing patterns, and detecting roles within the microservice architecture, thereby enhancing the system quality.

Finding 4: *Identifying and adding annotations are top specific uses of annotations for microservices identification and refactoring. Moreover, these purposes show a preference for graph theory. However, there needs to be more studies on predicting and verifying.*

***Implication**: Researchers need to investigate predicting and verifying annotations based on graph theory to incorporate it into refactoring and microservice identification. This exploration will contribute to enhance refactoring practices.*

Identifying and adding annotations cover almost all the purposes for using them in microservice construction. However, there is a lack of static analysis-based techniques for predicting and verifying annotations. To address this gap, researchers may focus on incorporating more deep learning for predicting annotations, which could improve the system quality through the architecture evaluation providing automated analysis and prediction of architectural elements, dependencies, and potential issues. Thus, researchers may make informed decisions and take proactive steps in improving the architecture of microservices, ultimately enhancing their overall quality and maintainability.

These findings provide new insights into the importance of specific uses of annotations and their relation between their purposes and static analysis in microservices. These findings could inform future research in this area by deepening our understanding of annotations.

2.4.1 Future Outlook for Research

For annotations in microservice construction, several essential topics on software development remain. We outlook a potential research direction, which is discussed next.

Annotations for Energy Consumption in Microservices

With the increased number of cloud-based services and distributed applications, high demand for energy consumption results in environmental and economic concerns. This research direction aims to investigate the potential of annotations for green computing in microservices to address the concerns. In general, green computing techniques provide the following advantages (i) reduction of operational costs, (ii) minimisation of power consumption, (iii) evaluation of energy saving, and (iv) reduction of CO2 consumption and pollution [119].

Cloud computing already benefits from annotations, especially for parallel programming and energy efficiency [77, 19]. In this line, investigating how the usage of annotations

can contribute to evaluate energy in microservices. Moreover, annotations can be leveraged to optimise resource allocation, improve scalability, and enhance energy efficiency. Therefore, by understanding how our purposes and specific uses of annotations are related to energy consumption, the researchers can involve developing energy-aware annotation frameworks, exploring dynamic resource provisioning based on annotations, and studying the impact of annotations on energy consumption in real-world microservices.

Discovering Microservice Structures Changing Over Time

This research idea explores the potential of identifying annotations to discover and understand the underlying structures and dependencies. Although there exist techniques that consider identifying annotations in one instant, this research idea considers the evolution of microservices over time similar of how java annotations changes [138]. By analysing and recognising how microservice structures change, researchers can gain insights into the organisation and management of microservice architectures.

In this research direction, the focus is on investigating the state-of-the-art techniques for identifying structures of microservices, and adapting the techniques to handle changes over time. Researchers can experiment with the specific uses of identifying, adding and modifying annotations to compare evaluation architecture. Thus, researchers can track the changes and their impact on the overall architecture. Additionally, academic papers can focus on algorithms and access tools that leverage annotations as clues to infer relationships, visualize dependencies, and aid in the efficient design and evolution of microservice systems.

Deep Reinforcement Learning for Evolution of Microservices

This research uses Deep Reinforcement Learning (DRL) to predict possible structure changes. Deep learning algorithms have shown remarkable capabilities in learning patterns to make predictions [38]. Reinforcement learning is an algorithm in which an agent interacts with an

environment to achieve a specific goal by taking actions given a reward. In microservice evolution, the environment represents architecture, the actions are possible structure changes, and the reward involves quality attributes of a system. DRL trains the agent to predict the most effective actions based on the changes in the structures of open-source projects.

This research process involves building a dataset of annotated microservices and corresponding actions taken over time, training a DRL model based on historical information and patterns, evaluating prediction accuracy, and integrating annotation prediction models into the lifecycle of microservices. Our specific uses of annotations, especially identifying, adding and modifying annotations, are suitable for DRL actions. Moreover, the reward could consider metrics related to our catalogue or purposes of using annotations. By leveraging DRL techniques, researchers can investigate how predictive models can anticipate and suggest suitable structure changes based on actions for improving microservice performance, adaptability, and robustness.

Verifying Annotations for Security Constraints

In this research direction, the critical aspect of security in microservice architectures benefits from verifying annotations. Adopting of microservices in modern software development posits security to protect microservice-based applications against cyber-attacks and vulnerabilities [38]. Annotations are a valuable source of information for security verification. This research aims to develop techniques to verify the correctness of annotations in the context of security and how this verification mitigates security risks in microservices.

The first step in this future research is conducting a systematic literature review on security for microservice architecture. The second step would involve examining a source code to match security patterns and potential security risks. This step benefits from our data extraction, which provides a list of studies related to vulnerability detection with static

analysis-based techniques. The last step would elaborate a verification technique and validate its effectiveness in identifying security risks and facing cyber threats. The findings and contributions of this study can provide researchers, practitioners, developers and architects with a verification framework to enhance the security of applications.

This research direction emphasises the specific use for verifying annotations, particularly concerning security-related constraints. The focus is ensuring that annotations adhere to security requirements and detect vulnerabilities, mainly to prevent unauthorised access, data breaches, or inserting malicious code/data. Researchers can explore techniques and methodologies for verifying the correctness of annotations and the security of microservices. Additionally, constraint-based verification approaches based on static analysis techniques are suitable for further investigation of verifying data privacy, integrity, and compliance.

2.4.2 Threats to Validity

This section outlines the main threats affecting the validity of the SLR conducted in this study.

- **External validity**. There is a threat to the knowledge generalisation of our study. We categorised primary studies from established digital repositories along with our experience related to the field. To address this potential threat, we captured a wide range of publication years to encompass different software practices that evolve. The publication years allow us to consider how annotations are used in microservice construction.

- **Internal validity**. There is a threat to the selection of studies that ensure that selected studies are relevant to the research questions. To mitigate reviewer bias, we assess the quality of each study, considering its methodology, data collection, and results description. The selection of primary studies with a high-quality value for empirical research and objective description mitigates the potential bias that might lead to inconsistent decisions.

- **Construct validity**. There is a threat of missing some papers despite the attempt to include primary studies of annotations in microservice construction. To mitigate this potential threat related to the collection of relevant studies, we diversified the inclusion criteria explained in Subsection 2.2.4 to capture studies from various contexts, and we applied cross-checking method in the primary studies.

- **Conclusion validity**. There is another threat regarding the reliability and completeness of our catalogue on purposes and specific uses of annotations in microservice construction, considering that additional categories might enrich the classification. To address this threat, we refined our catalogue with the new concepts encountered in the literature. Additionally, our catalogue may evolve according to the new changes.

2.5 Gap Analysis

This section highlights the selected limitations to be addressed in the thesis. Limitations are related to annotations in microservice construction.

1. **Absence of a study on technical concerns in microservices:** Some existing studies highlight the significance of defect prediction as a primary purpose of using annotations. Notably, missing annotations emerge as a crucial technical aspect within the domain of defect prediction. However, the current research landscape needs to provide a comprehensive study that reveals what other technical concerns are common development concerns in microservices. Therefore, a broader spectrum of possible defects and symptoms demands an exploration of technical aspects beyond annotations. Within this context, we advocate a critical examination of posts published on one of the largest sources for developers. Chapter 3 addresses this limitation.

2. **Lack of learning techniques to detect missing annotations:** Our findings indicate that more studies on identifying annotations rely on graph theory. However, there is a significant need for learning techniques that facilitate identifying and modifying annotations. The absence of learning techniques is problematic because machine and deep learning have the potential to enhance the automation of detecting missing annotations, which is crucial for maintaining code quality and functionality. Additionally, deep learning techniques have proven helpful for predicting annotations [134]. Although adding and modifying annotations has shown promising results for defect prediction, especially for addressing missing annotations, the development of dedicated learning techniques for detecting missing annotations remains insufficiently studied.

   Chapter 4 addresses this limitation by providing a semantics-driven learning technique, a novel approach inspired by the results of natural language processing for clone detection. The approach utilises a database of code fragments to find similarities.

3. **A semantic learning technique to improve microservice granularity:** While existing literature has primarily focused on using annotations for performance assessment in microservices [22], there remains a gap in learning techniques to optimise microservice granularity. Current approaches, including those utilising deep learning for predicting annotation [134], have overlooked the potential of learning techniques to inform architectural decisions related to the size of microservices. In the context of microservice granularity, the distinction between fine-grained and coarse-grained services significantly influences performance. However, existing methodologies lack adequate consideration of semantics learning techniques to anticipate granularity values. Therefore, there is a clear need to explore learning techniques aimed at identifying typical granularity based on existing open-source projects. Chapter 5 addresses this limitation, where a novel approach is proposed to investigate semantics-driven learning techniques for identifying granularity limits.

2.6 Related Work

This section outlines the related work of annotations in microservice construction, extending beyond the existing studies on the topic. Table 2.10 shows how our work fits the state-of-the-art by comparing key features between our work and the closely related work.

Table 2.10: Related Work

Features	**Related Work**					**Our Work**
	[7]	[33]	[65]	[14]	[138]	
Missing annotations	✓	✗	✗	✗	✓	✓
Refactoring	✗	✗	✓	✗	✓	✓
Bad-smell detection	✓	✗	✗	✗	✓	✓
Testing and documentation	✓	✗	✗	✗	✗	✓
Graph theory	✗	✗	✗	✓	✗	✓
Machine learning	✗	✓	✗	✗	✗	✓
Deep learning	✗	✓	✗	✗	✗	✓
Rule-based methods	✗	✗	✓	✗	✗	✓

2.6.1 Annotations in Microservices

Two previous surveys consider the importance of using code annotations and analyse the missing annotations scenario. Yu et al. [138] discuss the evolution of annotations in the bug history of Java applications. The authors provide findings regarding their possible usage when developers ADD, DEL, CHANGE and UPDATE annotations. Alshemaimri et al. [7] provide insights into the persistence logic and transactional behaviour of database code fragments. The authors focus on Object Relational Mapping (ORM) antipatterns and discuss their impact on performance, maintainability, portability, and data integrity.

Other previous works mention the importance of code annotation usage. Del Alamo et al. [33] discuss annotation development in the context of services offered worldwide as textual documents. The authors investigate the use of constraints based on structural relations

between annotations to analyse the privacy policy texts. Laigner et al. [65] catalogue the antipatterns of dependency injection, like constructors assumed with a particular signature. Bakhtin et al. [14] perform a survey on techniques to detect microservice patterns. The authors focus on four aspects of microservices: foundation, responsibility, evolution, quality and structure. However, they do not employ a perspective for prediction of annotations. Unlike previous efforts, our article introduces the purposes of using annotations.

Due to a limited number of surveys and a systematic mapping study, we explore other previous works which refer to contextual annotations in source code to guide maintenance tasks. Rongrong et al. [110] use the comment lines as a refactoring recommendation to classify the reconstruction of clones. Zhu et al. [149] search SQL sentences to detect security vulnerabilities in sensitive table operations. Omoronyia et al. [85] and Montalvillo et al. [81] add text information to the code and identify which code belongs to shared assets' features. The authors generally perform static analysis to find textual annotations. Although these studies generally employ static analysis techniques to find textual annotations, there exists a need for a systematic review of the purposes of using annotations and applied techniques.

By classifying the purposes of using annotations, we aim to provide a comprehensive framework to (i) understand the approaches used in the literature; (ii) offer insights into the limitations of each technique applied for a specific purpose of using annotations; and (iii) identify opportunities for innovation into specific use of annotations. Therefore, developing a classification represents a novel contribution in this area, offering a structured approach to navigating the landscape of static analysis techniques related to annotations in microservices.

2.6.2 Static Analysis-based Techniques

Regarding the actions that static analysis-based techniques employ with annotations, Alshemaimri et al. [7] discuss the identification of annotations for locating entities and their usage in SQL transactions; while Yu et al. discuss how the annotations evolve after system

changes. Although the authors mentioned the identification of annotations, only Yu et al. explicitly consider the ADD and MODIFY annotations. Based on available information, our study elaborates on classifying static analysis techniques as a new perspective to guide microservice construction.

Despite the previous works utilise static analysis techniques, Laigner et al. [65] and Bakhtin et al. [14] collect studies that employ techniques based on graph theory and set of rules to identify and categorise the patterns. Additionally, different techniques based on deep learning are systematic collected in the study performed by Del Alamo et al. [33], specifically, techniques that leverage the usage of natural language processing with symbolic and statistical approaches. The statistical approaches mix the natural language processing with traditional machine learning like Support Vector Machine, Logistic Regression, Decision Tree, Random Forest and others.

Considering the limit number of previous surveys, we also review studies that apply traditional methods such as Naive Bayes [110], Call Graphs [85], Java Parser [106, 84], among others. Although the authors identify the techniques and tools for static analysis, they do not apply machine learning techniques for annotations. To the best of our knowledge, our study is the first to provide a classification of purposes of using annotations and their relation with the specific usage of annotations performed by static analysis techniques.

2.7 Summary

The systematic literature review conducted in this study aimed to explore the usage of annotations in microservice construction. The review focused on the specific uses of annotations, the purposes of using annotations and the static analysis-based techniques applied to achieve the corresponding purpose. By analysing a wide range of relevant research articles,

the review shows the essential findings and trends in the field of microservice construction.

Based on the findings from the systematic literature review, a compelling argument emerged regarding the importance of defect prediction. Thus, verifying annotations is commonly used for the detection of annotation misuse, which is critical for defect prediction. The review revealed that mining annotations and constraints played a crucial role in verifying annotations. Consequently, the argument is that researchers must address and fix missing and misused annotations, as this contributes to the construction of microservices.

From this study, we gained insights into the existing use of annotations to suggest future research directions as follows: (i) discover the relation between annotations and energy consumption; (ii) capture the structures changing over time to understand the management of microservices; (iii) predict practical uses of annotations for the evolution of microservices; and (iv) consider the verification of the correctness and security of annotations.

The findings from this systematic literature review base the foundation for the subsequent three chapters of this thesis: (i) Chapter 3 aims to identify common concerns related to annotations in microservices faced by developers; (ii) Chapter 4 aims to reduce the issues related to missing annotations by learning the relation between annotations and code fragments for predicting annotations; and (iii) Chapter 5 aims to determine granularity limits by learning the typical granularity values from operations with similar behaviour that have same annotations.

In summary, the subsequent chapters build on the insights gained from the systematic literature review, addressing specific gaps and proposing innovative solutions to enhance the use of annotations in microservice construction.

Chapter Three

Classification of Microservice-Based Development Concerns

3.1 Overview

Microservice is an approach for designing, developing and delivering applications based on agile methods focusing on service choreography to promote isolation and autonomy [34]. This approach enables fast application development and potentially localises bugs to limit their propagation within the system [76, 136]. However, debugging is complex due to the efforts required to trace the failed requests among several log files spread on different containers [103, 128, 49]. The processes of locating and fixing bugs are eventually one of the core purposes of adopting microservices (i.e. designing for failure recovery); the effectiveness and efficiency of the process are essential for ensuring the availability and compliance with its Services Level Agreements and maintaining its revenue streams [42].

Although the automation of bug correction helps reduce the time during the testing stage, in practical terms, detecting bugs on microservices may take days or even weeks [146]. Furthermore, a high number of microservices implies a large vulnerable surface where bugs

can propagate across dependencies, which complicates the identification of the root of the issue [37]. In this context, developers usually search online communities to learn about development issues and their solutions [82]. Consequently, developers unfold related posts with the suggested answers, comments and examples to find solutions to their problems [102]. Notably, these answers often provide insights into addressing concerns related to annotations.

The novel contribution of this chapter is an empirical study that analyses 406 posts from Stack Overflow [54], one of the largest sources for developers [102, 79] , focusing on bugs, their symptoms, and root causes. We categorise each post by development lifecycle phases, quality attributes, software construction activities and fault classes [112, 20, 57], to understand the concerns developers face with microservices. This classification highlights specific concerns including those related to annotations, offering insights into their impact on development processes. This work inspects the corresponding response and comment thread for each post and contributes to strategies for enhancing microservice construction.

Despite the significance of Stack Overflow, only one previous work has reported technical issues on microservices by examining answers from Stack Overflow posts [15]. However, it excludes all the comments that provide relevant post information, like different results after applying one answer [55]. Different from that work, this research categorises common development concerns by including posts and their comments. Moreover, previous empirical studies on microservices have either (i) introduced algorithms using fewer issues to evaluate their approaches [89, 145, 147], or (ii) interviewed fewer developers to identify undesired practices [116, 53]. In contrast, this chapter examines concerns from a community perspective.

The remainder of this chapter is organized as follows. Section 3.2 describes the life cycle of microservices at runtime, whereas the design of the study is presented in Section 3.3. Section 3.4 presents the results, and Section 3.5 shows how these results complement Chapter 2. Section 3.6 discusses the related work. Finally, Section 3.7 summarizes our work.

3.2 The Life Cycle of Microservices at Runtime

The life cycle of a microservice-based system at runtime is triggered by request protocols like Hypertext Transfer Protocol (HTTP) and Advanced Message Queuing Protocol (AMQP). This life cycle is composed of four system activities: service routing, service discovery, service authentication & authorization, and service invocation. Figure 3.1 shows the process that starts with service routing and ends with service invocation, which calls the microservices and returns the required response.

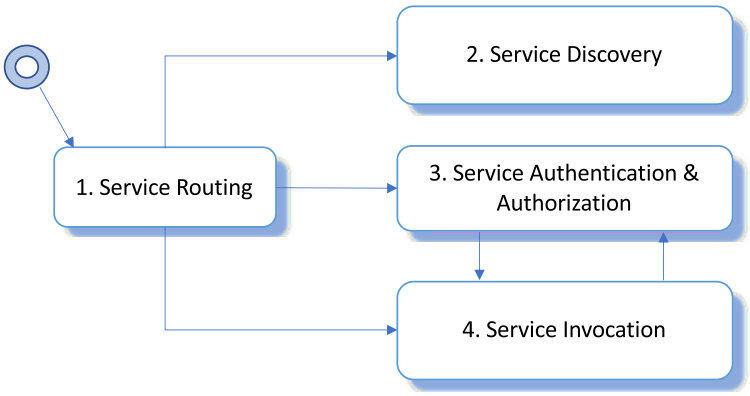


Figure 3.1: Microservice-based System Lifecycle at Runtime

Service routing has an entry point that routes all requests to other microservices [120]. Specifically, the API gateway is the component that receives an initial request, examines the availability of a microservice, selects its correct version and collects the responses of the required microservice [8]. In the context of asynchronous responses, API gateways implement *polling* for capturing responses. Additionally, common frameworks that help to construct gateways are *Kong*, *JHipster*, and *Swagger* [2]. Furthermore, the service routing communicates with all the other life cycle activities.

Service discovery collects microservice health status (i.e. location, IP address, and ports of available instances) and keeps this information in a component called register, which returns these details when requested. Additionally, load balancers, databases, repositories,

and other components are initialised in this activity. Depending on changing environments or workloads, these components may use a configuration manager to initialise and adjust the system performance.

Service authentication & authorization validates user credentials and permissions. If a request requires authentication, the user credentials are validated. If a request requires authorization, the access permissions are also checked. Depending on the implementation, this activity communicates to the service invocation activity to propagate access permissions to other microservices. This activity needs the use of security protocols. *JSON Web Token (JWT)* and *OAuth* are the most common security protocols. The former is an open-source JSON-based standard that enables tokens as a medium for secure message transmission [2]. The latter focuses on simple client authorization for Web applications [101].

Service invocation is the activity in which one or more microservices are invoked. These invocations are performed in either a synchronous or an asynchronous manner. The former use invocation patterns such as *aggregation, chain, proxy* [47], whereas the latter use tools based on messaging broker, streaming, and cache to send data to the invoked microservices. Regarding the origin of the service requests, most of these are received from the service routing. However, some requests come from the service authentication & authorization due to permissions propagation. For the requests that come with a token to establish a secure invocation, the validity of the token is checked before processing the request.

3.3 Study Design

This section presents the key aspects of this study design, which follows well-established guidelines on systematic studies [60, 91].

3.3.1 Research Question

This study aims to gain insights into the main concerns in microservices development. This work is an effort to (i) prevent junior developers from introducing accidental errors during the implementation of distributed patterns and (ii) suggest future research on static code analysis to recognise and fix the misuse of annotations. This investigation centres around addressing the following Research Question (**RQ**):

RQ1.3: *To what extent the use of annotations is one of the most common concerns in microservice development?*

Developers benefit from microservices to enhance the scalability and modularisation of applications. To achieve both enhancements of microservice development, they also motivate the isolation of functionalities into highly cohesive services. However, many services may increase communication and data consistency complexity. The objective of RQ is to provide empirical evidence of existing microservice development concerns classified according to different software engineering activities in the life cycle of microservices at runtime.

3.3.2 Search and Selection Process

Figure 3.2 shows the stages of our search and selection process and the number of Stack Overflow posts at the end of each stage. For a better control of the post characteristics, this study considered the following stages in the design of our study: initial search, accepted answer criteria, conceptual and technical criteria, merging and duplication removal, and application of selection criteria.

1. Initial search. In this stage, Stack Overflow was the first option as a search engine since it is one of the largest and most popular online Question & Answer (Q&A)

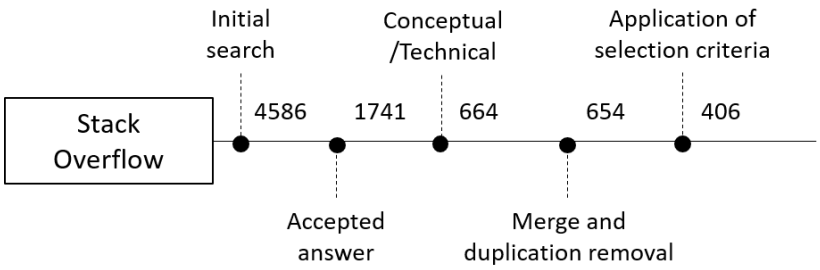


Figure 3.2: Numbers and Stages of our Search and Selection Process

forums, in which developers post programming issues [102, 82]. The tag *microservices* was the main criterion considering the main context. This study limited the search to posts dated until December 2019.

2. Accepted answer criteria. A post could have multiple answers; however, the owner of the post can select only one answer as accepted. The selection of posts considers those with an accepted answer using Query Stack Overflow [1]. This website executes simple Structured Query Language (SQL) statements against public data from Stack Overflow.

3. Conceptual and technical criteria. The posts were manually classified into two main categories, namely conceptual and technical. On the one hand, a post falls into the conceptual category when the question is too general, contains non-technical details, and excludes answers or comments. On the other hand, a post falls into the technical category when the question is more specific, and their answers or comments contain technical text. The posts in the technical category are the ones which pass to the next stage.

4. Merging and duplication removal. Moderators on Stack Overflow mark posts as closed when duplicated, off-topic, or unclear. Based on this, this study excludes closed and duplicated posts. Additionally, collected posts consider those posts suggested as main threads.

5. Application of selection criteria. The selection process manually filters all the

collected posts according to the selection criteria in Table 3.1.

Table 3.1: Inclusion and Exclusion Criteria for Post Selection

	Inclusion criteria (Is)		**Exclusion criteria (Es)**
I1	*Microservices* posts created until December 2019.	**E1**	Posts with duplicated content.
I2	Posts with an accepted answer.	**E2**	Posts with a status of closed.
I3	Technical posts for microservice implementation.		

3.3.3 Data Extraction and Synthesis

In this stage, (i) the classification process puts the posts according to the activities in the microservices life cycle, quality attributes, software construction activities, and fault classes; and (ii) the collected data for this study includes the publication year, bug symptom, root cause, and system activity.

The data synthesis involved a collection and a summary of the data extracted from the posts [61]. It was aimed at understanding and analysing posts on microservices development. Specifically, the categorisation of posts facilitated content analysis, and narrative synthesis explained the findings coming from the content analysis.

For the narrative synthesis, the creation of groups considered posts with similar contexts by using the extracted bug symptom(s) and root cause from each post. A discussion of created groups was also performed until the authors reached a consensus.

For the sake of the replicability of our study, a replication package [1] is available for interested readers. The replication package includes the research protocol, the SQL scripts, the raw data with the list of the retrieved posts, the extracted and categorised data, and the scripts for generating the information charts.

[1] http://www.research.propio.click/paper-empirical/replication-package/

3.4 Results

This work presents the observation of posts over the years in Section 3.4.1 and the main findings per life cycle activity in Section 3.4.2. Additionally, Section 3.4.3 shows the categorisation of posts based on quality attributes, software construction activities and fault classes [112, 20, 57]. Since referencing hundreds of posts occupies a significant space, the identification of posts use the letter *P* followed by a unique hexadecimal number (e.g. P1B).

3.4.1 Years vs Life Cycle Activities

Figure 3.3 shows the distribution of posts on microservice development over the years. Although only a few posts were created during 2014, large organisations became interested in microservices [87]. Moreover, the trend indicates that the activity of the community in microservice-related topics has been steadily growing over the years, except in 2019, which has a reduction of 27% of the posts compared to 2018.

The results show that the life cycle activity with the highest number of posts is *service discovery* (161/406), followed by *service invocation* (136/406), *service routing* (69/406), and *service authentication & authorization* (40/406).

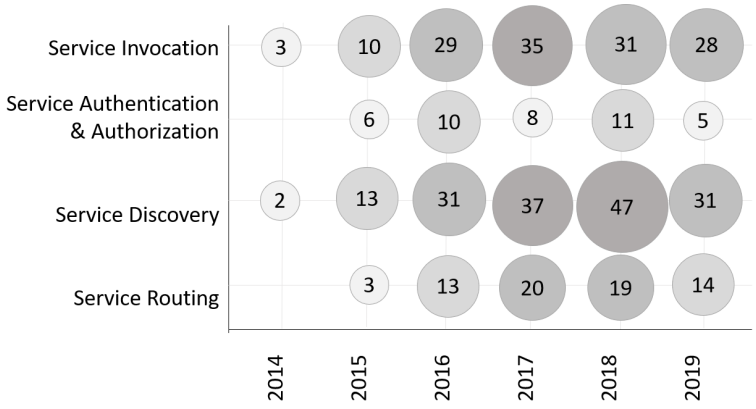


Figure 3.3: Life Cycle Activities Over Time

3.4.2 Findings per Life Cycle Activities

This work categorises the posts based on their pertinence to *service routing*, *service discovery*, *service authentication & authorization*, and *service invocation*.

Tables 3.2 and 3.3 present the distribution of posts per life cycle activities. The system activity column indicates the name of the activity and the percentage of posts related to annotations. These tables indicate that 40% of posts are related to annotations. In this sense, *Service Routing* has 17% of posts related to annotations, *Service Discovery* has 8%, *Service Authentication Authorisation* has 4%, and *Service Invocation* has 11%. Finally, 27% of posts are mainly related to missing annotations.

Service Routing

Table 3.2 shows the service routing distribution. The main tasks in service routing are *client discover* 35/59, followed by *send request* 18/59 and *polling response* 6/59. *Client discover* involves issues in implementing clients intended to discover microservices; for instance, uploading files fails to discover a microservice with an overridden UTF-8 configuration (P7D). *Send request* issues appear when routing requests with the wrong configuration of cross-origin in Zuul (P12B) [2]. *Polling response* occurs when the reception of an asynchronous response fails. For example, the navigation flow of an application is interrupted due to periodically missing a resource status (PE1).

Observations revealed that 40% of the posts in client discover are related to missing parameters or operations. The former, missing parameters, refers to the absence of parameter names and values during the invocation of scripts or configuration files. In contrast, missing operations refers to the absence of sentences for adding specific behaviour. For instance, a client uses the reported status to interact with the dependencies according to their health.

Table 3.2: Classification of Posts

System Activity	Activity Task	Task Description	Issue Category	Ratio
Service routing (17% are based on Annotations)	Client discover	Issues in the implementation of clients that discover microservices	Missing parameters Missing operations	9.9%
	Send request	Wrong settings for routing of request	Wrong configuration of environment Missing publish event Missing serialise mechanism Missing annotations (3%)	5.4%
	Polling response	Incomplete asynchronous response	Missing response (message, record id, promise or link) Missing event listener	1.7%
Service discovery (8% are based on Annotations)	Configuration	Missing values, parameters or annotations for settings of components	Wrong configuration of frameworks Wrong usage of parameters Wrong version of dependencies Missing annotations (24%)	30.3%
	Start	Failures when starting components	Wrong version of dependent libraries Wrong login options Missing operations for services	5.7%
	Registry	Issues during interaction with registry service	Load balancing error caused by different version of microservices	3.7%

Continued on next page

Table 3.3: Classification of Posts *(Continued from previous page)*

System Activity	Activity Task	Task Description	Issue Category	Ratio
Service authentication & authorization (4% are based on Annotations)	Authentication	Incomplete validation of a user account	Missing operations Missing Web tokens	5.2%
	Authorization	Deny the access to resources	Missing operations Missing Web tokens Missing event messages	3.4%
	Grants	Incomplete adoption of user roles to keep authentication & authorization	Wrong data usage High workload High coupled microservices	1.2%
Service invocation (11% are based on Annotations)	Asynchronous	Errors when calling microservices simultaneously	Common cache errors Missing patterns for distributed systems	21.9%
	Synchronous	Interruption of calling microservices sequentially	Cross-origin resource sharing (CORS) Broken contracts Outdated databases	8.1%
	Token validation	Errors when validating secure tokens	Missing implementation Missing Token Unsecured zone	3.4%

Regarding missing parameters, common issues include environment variables, security on network access, framework settings, health metrics, and values of annotations are the most common concerns. For instance, the variables *HOST* and *PORT* require valid values to connect a remote machine (P59, P85, PD7), a configuration management as Zookeeper is required to hold the parameters for frameworks such as Zuul, Consul, gateways (P3E, PC6), and annotations such as *ConsulConfiguration* and *EnableSwagger* requires values to fetch data, generate documentation and others (PF6).

In relation to missing operations, the most relevant operations are the implementation of variables, getters, setters, and rules for service discovery. For instance, Swagger requires ApiImplicitParam on getters and setters to generate documentation (PA7), service discovery clients ignore the different DNS ports to get a list of microservice instances from SRV records (P11, P55, P89), and developers miss rules for checking the version/health/context of microservices in Zuul, Nginx or vector clocks (P15, P76, PCB).

Observations revealed that 56% of the posts in *send request* are related to wrong configuration, 22% are related to missing publish events and missing format for case class, and 6% refer to missing annotations. The wrong configuration issues refer to the settings of the environment, relative paths, and security properties. Specifically, the settings of Eureka for queue messaging and replication need to check the waiting time for the next retry or increase the maximum number of retries (P36, P8F, PA8, PD4, P12F, P143). In the case of missing publish events, ActiveMQ, RabbitMQ, Kafka, Spring Framework, and Spray JSON must catch exceptions when the CrossOrigin annotations are missed.

Finding 1: *Missing parameters emerges as one of the most popular issues in service routing. Results suggest API gateway frameworks require those parameters for configuration.*

***Implication**: Developers need to fix missing parameters and environment variables for Somata, Cassandra, and Kong [2], which may contribute to reduce bugs at the compilation stage.*

Finding 2: *Missing operations emerge as a popular topic in service routing. Among these kinds of operations: ApiImplicitParam to generate documentation, SRV record to list microservices and health status to monitor microservices.*

***Implication**: Developers need a deeper understanding of frameworks such as Consul, Mesos-DNS, Nginx, Swagger, and Zuul [2]. A guide of missing operations in these frameworks could help mitigate some issues.*

Service Discovery

Table 3.2 also presents the distribution of posts related to service discovery activity. The main tasks in service discovery are *configuration* 110/142, followed by *start* 17/142 and *registry* 15/142. *Configuration* refers to the lack of values, parameters, or annotations; for instance, if a parameter discovery is disabled then localhost and port between Java Spring and Consul are unresolved (P23) [2]. *Start* refers to issues that occur when starting components, such as using the wrong package version of JHipster when launching a microservice (P112). *Registry* refers to issues that occur when components interact against the registry service. For instance, Eureka clients are still working despite an Eureka server stop (P121).

Regarding the configuration task, 24% of the posts are related to the usage of the following frameworks: Spring, Netflix, Cassandra, Zookeeper, GemFire and others [2]. In this context, the common issues of configuration tasks are (i) lack of values for configuration of fallback, and circuit breaker and lack of parameters for dependency injection or binding exception; (ii) wrong version of libraries; (iii) annotations of Spring Boot, Lombok and GemFire are missed/misused [2]; (iv) data serialization provides communication outside private networks; and (v) missing fault-tolerant clusters to avoid single point of failure.

In the context, the issues arise when the following scenarios appear: query performance is deficient when essential filter is not an enumeration (P3A, P57, P67, P6C, PFA),

BindingException occurs when server port is absent of the Java Virtual Machine (JVM) parameters (P22, P2E, P80, PBC, PD1, PE4, P11D, P134), *unsupported class version* occurs when using a higher version of servo-core dependency (P33, P51, P6B, P97, PA9, P92), *UnknownHostException* may occurs when *RestTemplate* miss the *Autowired* annotation (P47, P3C, PC9, PD0), front-end microservices should return a *ResponseEntity* serialized as JSON to include the HTTP status code (PB2, PD8, P17, P93), and a system lose events if a central messaging hub has no cluster and becomes unavailable (P2, P98, PE5, PB1).

Regarding the start task, 47% of the posts are related to (i) the usage of libraries for dependency, especially when *bootRepackage* is no longer supported by JHipster version 5+ (P7, PB3, P112); (ii) container initialization due to wrong login parameters, when docker containers require the option –with-registry-auth to forward credentials to other nodes (PBB, PE0); and (iii) operations for a high number of services such as backup/restoration, logs, and scaling, when there is no configuration for timeouts and retries then the system could have scaling troubles (P18, P35, PF2).

In relation to the registry task, 40% of the posts are concerned about a load balancing between different versions of microservices (P16, P108, P10D), for instance, load balancer considers a version of microservices unless the older version is marked as OUT OF SERVICE; and detecting the connectivity of services and components (P13, P1E, P69), for instance, if health checking is not enabled then a system cannot detect when microservices are down.

Finding 3: *The wrong version of libraries is one of the top issues in service discovery. Consequently, compilers do not find the requested classes in the library.*

***Implication**: Developers may benefit from the usage of project management tools like Apache Maven or build automation tools like Gradle [2]. Both kinds of tools offer support for library and dependency management.*

Finding 4: *Annotations, protocols and clusters appear among the top issues in service discovery. According to our survey, microservice-based applications benefit from (i) the usage of NonNull annotation to warn about the presence of null values; (ii) the adoption of JSON protocol for external communications; and (iii) the activation of clusters with dynamic IP.*
***Implication**: Developers could reduce these issues by using static code analysis to recognise the misuse of annotations, protocols and clusters.*

Service Authentication & Authorization

Table 3.2 illustrates the task distribution of service authentication & authorization. The main tasks in service authentication & authorization are *authentication* 18/37, *authorization* 14/37, and *grants* 5/37. *Authentication* refers to issues that occur when trying to validate user accounts (P48). *Authorization* issues occur when trying to provide access permissions to specific functionality (PE8). *Grants* issues arise when adopting the user roles to avoid repeating authentication & authorization processes (PB).

Observations revealed that 44% of the posts in authentication & authorization are related to missed operations and usage of web tokens. On the one hand, missed operations refer to the absence of sentences for the usage of user credentials, configuration, and others. On the other hand, web tokens refer to the time life of tokens and excessive validation.

Missed operations raised concerns about user credentials and configuration usage to avoid failures of unique constraints and missing annotations. Common scenarios included are: requests return 401 unauthorized responses when credentials are unavailable into Eureka or Spring Cloud [2] (P46, P52, PBE, PFE); transactions include the username field as part of unique constraints (P3F, P48, PAC); and clients missed *RestTemplate* with the *LoadBalanced* annotation to support multiple instances of microservices (P62).

Regarding exposing web tokens, there are two concerns: (1) a slow response of the overall system due to multiple validations of a token where developers could pass valid JWT tokens to other microservices (P9, P1D, P2A, PF7) and (2) tokens that live longer than expected, they could pass between microservices, stay in the same domain and create a security vulnerability (P5F, P8B).

Finding 5: *The absence of authentication & authorization operations as JNDI lookup with the connection factory causes issues when getting user credentials in a centralized solution.*
***Implication**: Developers could choose a fully decentralized solution that includes Zookeeper, ZeroMQ, or Consul [2] to reduce the response time of the overall system.*

Finding 6: *Exposing the web tokens for a long time makes a microservice-based application vulnerable to attacks and hacks.*
***Implication**: Short timelife tokens minimize security breaches. Hence, developers could improve renewal mechanisms to ensure that compromised tokens remains valid for a brief period.*

Service Invocation

Table 3.2 depicts the task distribution of service invocation. The main tasks in service invocation are *asynchronous* 78/118, followed by *synchronous* 26/118 and *token validation* 14/118. *Asynchronous* issues occur when simultaneously calling microservices using eventual consistency patterns to handle inconsistent states over multiple responses (P19). *Synchronous* issues arise when sequentially calling microservices use HttpHeaders to send JSON data for HttpEntity instances (P1C). *Token validation* issues refer to tokens received by microservices and need internal validation when passed by the header between microservices (P4E).

Regarding the asynchronous task, 29% of posts are related to common cache errors and missing patterns for distributed systems. The most common cache errors include missing

Terracotta servers for backing up the cache [2] (P95), missing shared cache layer on top of a database to respond with latest data even when dependant microservices are inactive (P50, P11A, P5D), wrong cache parameters for the avoidance of data duplication, reduction of unnecessary communication, and validation of outdated data (P3, P1B, P7A, PED, PF3).

Concerning the missing patterns for distributed systems, observations revealed that sagas, event-store, eventual consistency, aggregator, reporting, shared database, and produce/consumer are the most mentioned patterns. Generally, the sagas pattern is used to get fresh data for completing complex processes by keeping a trace of events (PBF, PDA, PC4, PE2). The event-store replicates events when the system provides outdated user information (PBF, PDA, PC4, PE2). The other design patterns are combined for easy and fast access to connected data (P19, P4D, P8E, P96, P135).

With the synchronous task, 31% of the posts are concerned about (i) missed HTTP headers for microservices in different domains, (ii) change of bounded context that results in broken contracts and (iii) data synchronization after data structure modification. To illustrate, the Cross-origin resource sharing (CORS) filter becomes imperative in systems with Zuul and Azure Service Fabric with corresponding requirements for request headers on the client side (P34, P74, P75). Bounded contexts of microservices change according to the data structure dependency (P6F, PA3, PF9). Moreover, microservices without versioning produce errors after data structure modification (P107, P111).

Finding 7: *The absence of configuration parameters for cache is one of the most common issues on service invocation. This study indicates that communication processes among microservices require those parameters for optimal performance.*

***Implication**: Developers could use configuration parameters of cache (e.g. Ehcache enabled with Terracotta server) in external files such as YML configuration files.*

Finding 8: *Patterns for long-running transactions are another common issue on service invocation. Results suggest that transaction patterns reduce code duplication on services.*

***Implication**: Developers must focus on transaction patterns such as SAGAS, try/cancel/confirm, and event-store to maintain data consistency across microservices.*

3.4.3 Further Categorisation

Quality Attributes

Quality attributes are system characteristics a quality management team assesses to judge software quality [112]. Table 3.4a shows the distribution of posts in terms of quality attributes. The main identified attributes are *reliability* 105/406 and *security* 90/406, followed by *performance* 65/406 and *availability* 64/406. Posts related to different quality attributes are presented in a category called *Others* 83/406.

Security concerns are related to the quality attribute of microservices inside a private network with IP addresses that expose them to public networks (P32, P98, PAF). *Reliability* issues are related to missing handlers, retries, publish events or incomplete rollback that affect the operation time of a system (P3C, P8, PF, P10). *Performance* refers to issues when response time is affected by missing cache implementation (P4, P105, P7A, P3, P1B). *Availability* issues arise when the execution of a system stops; for instance, the Zuul filter launches errors when missing the http header X-Forwarded-Host (P87, PA3, PC5, PE3).

Software Construction

Construction of software refers to the elaborated creation of software by combining coding, verification, debugging, unit testing, integration testing, and debugging [20]. Table 3.4b

Table 3.4: Further Categorisation

(a) Quality Attributes

Quality attributes	**#Posts (%)**
Availability	64 (16%)
Performance	65 (16%)
Security	90 (22%)
Reliability	105 (26%)
Others	83 (20%)

(b) Software Construction

Software construction	**#Posts (%)**
Coding and debugging	128 (32%)
Detailed design	57 (14%)
Integration	152 (37%)
Unit testing	6 (01%)
Others	63 (16%)

(c) Fault Class

Fault class	**#Posts (%)**
Input/output faults	65 (16%)
Logic faults	171 (42%)
Interface faults	92 (23%)
Data faults	48 (12%)
Others	30 (07%)

presents a distribution of posts in terms of software construction activities. The main tasks in software construction are *integration* 152/406 and *code & debugging* 128/406, followed by *detailed design* 57/406, *unit testing* 6/406 and *others* 63/406.

Integration issues occur during the interaction of multiple components by missing operations to keep a clean state of data (P2C), single-sign-on with discovery (P1A, P23) or aggregator with integration platform (in P4C, P4D). *Code & debugging* concerns are related to missing code for cache (P3), handling events (in P19, P3C), overriding methods (in P6C, P7B, P86), and annotations to validate and enable transactions (in PC9, PD0). *Detailed design* issues occur in approaches to build reports based on information from multiple microservices (P43), cache for event store to keep updating the user information (P79, P7A), and usage of data-streaming with eventual consistency to decouple the database (P7C).

Fault classes

Fault classes classify bugs that can be detected by static analysis to detect anomalies introduced in a piece of code [57]. Table 3.4c shows a distribution of posts in terms of fault classes. The identified fault classes are *logic faults* 171/406 and *interface faults* 92/406, followed by *input/output faults* 65/406, *data faults* 48/406, and *others* 30/406.

Logic faults issues appear in correct code statements that produce a wrong behaviour, especially when reading/writing a model, changing versions, making notifications, implementing events, tolerant clusters, JSON protocols and others (P6B, P73, P93, PA6, PB2). *Interface faults* concerns occur while adding identifiers, implementing failure recovery, aggregated patterns, and others (P6, P4C, P66, P8E, P91). *Input/output faults* arise when missing or using wrong a version of libraries, location of parameters, and others (P31, PA2, P123, P2E, PD1). *Data faults* occur when optimising the cache, managing persistent messages, and using decorators for accessing/storing/formatting data (P50, P7F, PD5, P109).

3.4.4 Threats to Validity

This Subsection outlines the primary threats that impact the validity of the empirical study performed in this chapter.

- **External validity**. There is a threat to the generalizability of knowledge in our empirical study. We categorised posts from one of the largest sources for developers, along with our experience related to the field. To mitigate this potential threat, we included a diverse range of publication years to cover evolving software practices. The publication years allow us to address a spectrum of concerns in microservice development.
- **Internal validity**. A threat arises in the selection of posts that directly relate to our research question. To address this reviewer bias, we evaluate each posts, considering its

data collection. By selecting posts with a high-quality value for empirical research and objective description mitigates the potential bias that might affect our decisions.

- **Construct validity**. There is a threat of overlooking specific posts, even though we try to cover the main concerns in microservice development. To mitigate this potential threat in collecting relevant posts, we kept simple the inclusion and exclusion criteria from Table 3.1 to capture posts from various contexts and remove unnecessary content.

- **Conclusion validity**. There is another threat regarding the completeness of our findings because we conducted the study in 2021 and set 2019 as the maximum publication year, even though additional years could enhance the classification. To address this threat, we refined our categories with new concepts to get a novel pattern up to the selected date, recognising that developers can continue to follow historical patterns.

3.5 Complementing the Systematic Literature Review

The results presented in this chapter provide a practical perspective that complements the theoretical findings of the systematic literature review (SLR) in Chapter 2. The SLR identified key issues, like the importance of defect prediction through annotations verification, which serve as initial understanding. This chapter delves into real-world instances of these problems by examining posts related to microservice development activities over several years. By categorising and analysing these posts, this chapter highlights specific challenges developers face in areas like service routing, service discovery, authentication and authorisation, and service invocation. These empirical observations reinforce the argument that addressing missing and misused annotations is crucial for microservice construction.

Moreover, the findings in this chapter underscore the practical implications of development concerns in the microservice lifecycle. The detailed analysis of posts reveals that missing

parameters, wrong configurations, and misused annotations frequently disrupt microservice operations. This real-world evidence supports the SLR for future research directions, such as predicting practical uses of annotations and verifying their correctness. By identifying specific problems and their contexts, this chapter not only validates the theoretical insights from the SLR but also provides concrete context that can inform the development of tools and techniques for improving microservice reliability and performance. This synthesis of theoretical and empirical insights lays a robust foundation for the subsequent chapters, which aim to address these identified gaps with innovative solutions.

3.6 Related Work

This section discusses closely related research.

Bandeira et al. [15] implemented an unsupervised machine learning that detects word or phrase patterns in Stack Overflow posts to build a general taxonomy of subjects on microservices. However, it excludes relevant information from comments and possible key answers that clarify posts [55]. Our research differs from their work in two dimensions. First, the classification framework is based on the following aspects of software engineering: software quality attributes, software construction activities, and fault classes. Second, this work follows systematic mapping guidelines with the corresponding replication package to increase the reliability and replicability of the study results.

Regarding the origin of the data in previous empirical studies on microservices, they either: (i) introduced algorithms using a reduced number of faults or issues to evaluate their debugging approaches [49, 89, 145, 147]; or (ii) interviewed a limited number of developers to identify undesired practices [116] or code smells [53] in microservice-based applications. In this context, to the best of our knowledge, our work constitutes the first attempt to reveal active development concerns from Stack Overflow as a big data source, relevant for

the development community, which may serve as a baseline for future empirical works.

3.7 Summary

This chapter surveyed Stack Overflow to provide a classification of posts based on microservice life cycle activities at runtime, namely service routing, service discovery, service authentication & authorization, and service invocation.

The results indicate that (i) missing parameters/operations are the most common concerns in service routing, (ii) wrong versions of libraries, annotations, protocols, and clusters appear as main concerns on service discovery, (iii) the absence of authorisation operation and web tokens exposed for a long time are critical concerns in service authentication & authorisation, and (iv) the absence of cache parameters and inadequate patterns for long-running transactions emerge as trending concerns in service invocation.

The observations reveal that the top concerns focus on security and reliability, followed by integration testing and logic faults. The implications for developers summarise in (i) the usage of short timelife tokens to reduce vulnerabilities, (ii) the usage of static code analysis to recognise misuses of annotations, (iii) the storing the configuration in external files to improve the cache performance, and (iv) the transaction patterns to maintain data consistency.

Chapter Four

A Semantics-Driven Learning for Annotations in Microservices

4.1 Overview

Annotations are a form of program metadata that generate code, configuration files, and warnings, among others. Microservice frameworks provide annotations to facilitate the implementation of cloud-based applications in terms of the reuse of features and support for software evolution. However, the misuse of annotations generates potential bugs whose detection requires the analysis of multiple logs and source code files. This detection effort is not trivial for developers since debugging microservices may take days or even weeks [146].

Developers go through a reduced amount of source code [73] and infer the functionality of code fragments [51]. Additionally, comprehension of programs takes around 58% of the time spent on software maintenance due to outdated or missing comments [132]. Moreover, the wrong usage of annotations introduces errors with unexpected behaviour. Despite their significance for microservice development, only a few approaches have worked with annotations to detect cycle dependency and misuse of annotations [93, 94]. However, these

approaches apply specific patterns and rules and do not provide mechanisms to match the source code with annotations for predicting them. Learning from annotated code fragments to predict annotations is essential to address this gap.

This chapter contributes to a novel static analysis approach using semantics-driven learning of code fragments collected from open-source repositories. Specifically, it combines a Recurrent Neural Network (RNN) and a K-Nearest-Neighbour (KNN) classifier to capture the semantic relation of code fragments and predict suitable annotations. Predicting annotations helps to identify missing annotations, detect misuses of annotations, and explain possible causes of these bugs through various experiments that simulate code fragments without annotations. The evaluation uses open-source projects from GitHub, extracting Abstract Syntax Trees (ASTs) from Java classes to provide syntactic knowledge [30]. This approach converts AST representations to vectors and finds vectors with similar features, drawing on techniques to detect similar code fragments [92, 90].

The approach in this chapter is the first to explore the relation between code fragments and their annotations and exploit them to predict suitable annotations for a given code fragment. The primary goal is to facilitate the maintenance activities of microservices, including code inspection, analysis, and debugging. By accurately predicting annotations, the approach aims to improve the quality of maintenance activities. Predicting annotations supports ensuring that annotations match according to the code behaviour. A simulation tool facilitates evaluating the approach by extending the PyTorch and Sci-Kit Learn Library. Results indicate that semantic learning achieves an average of 87% of the correct predictions of annotations.

The remainder of this chapter is organized as follows. Section 4.2 presents details of the approach. Section 4.3 provides the evaluation, followed by a discussion of related work in Section 4.4. Finally, Section 4.5 concludes the chapter and outlines future research.

4.2 Proposed Approach

This approach explores connections between a code fragment and its annotations. In this sense, analysing open-source code helps to understand how developers use microservice annotations, with a particular focus on the AST representation containing syntax and semantic information of code fragments. This information helps detect semantically similar code fragments. Thus, the approach learns the semantic information of existing code fragments to give suggestions about the declaration of microservice annotations in a code fragment.

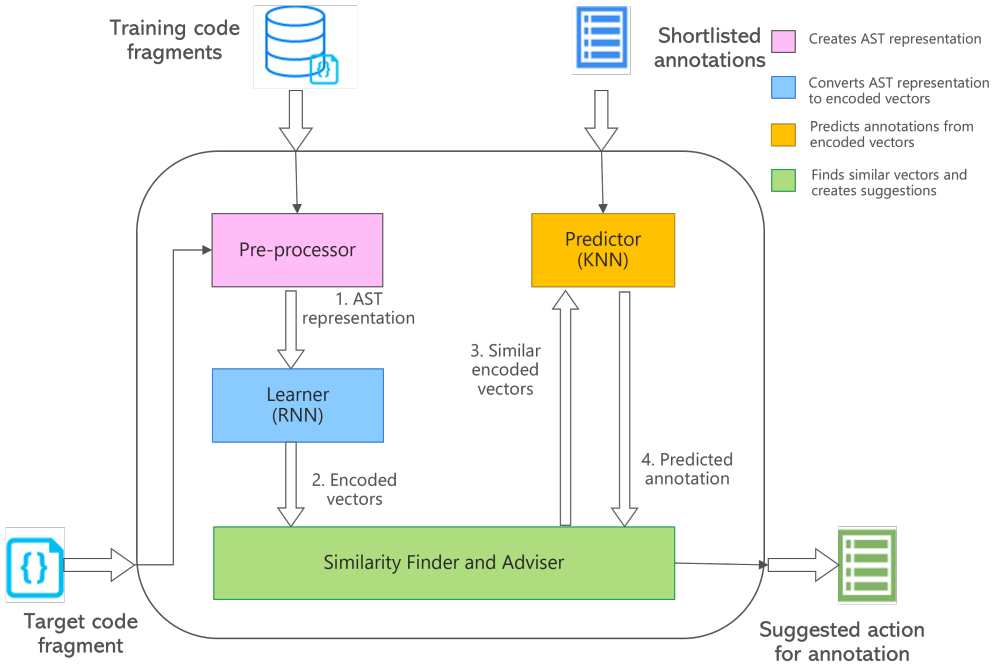


Figure 4.1: Conceptual Model of Semantics-Driven Learning Approach

Figure 4.1 shows the interaction between the components. The main inputs of the approach are: (i) a shortlisted microservice annotations ; (ii) a dataset of classes in text files with annotations for training; and (iii) a database of code fragments with or without annotations to find the best set of microservice annotations. The shortlisted annotations limits the scope of annotations under analysis. The input format of code fragments are instructions in a programming language such as Java. Listing 4.3 shows a sample of a code

fragment that includes one annotation. The output of our approach is a set of suggestions in terms of actions followed by the annotation name, e.g., *KEEP PostConstruct*. The *ADD* action suggests the incorporation of an annotation, and the *KEEP* action suggests no change in the usage of an annotation.

The inputs must be provided by researchers who have expertise in microservice development. Researchers build the inputs by compiling Java classes from many open-source repositories and projects that employ microservices to ensure a diverse and extensive source of examples. The shortlisted annotations are based on annotations commonly used in the compiled open-source projects. We determine the shortlisted by examining the source code and analysing which annotations have more code fragments. The training code fragments are a dataset of classes with annotations built from compiled Java classes with enough code fragments per annotation. The approach uses the training code fragments to train, validate and test the learner. The target code fragments are a database built from unseen code fragments, automatically generated by introducing differences with or without annotations.

Listing 4.1 shows examples of shortlisted annotations and inputs used for training the learner. It shows that shortlisted annotations are just the names of annotations to be evaluated during the experiments. In the listing, we show one annotation as an example. Additionally, Listing 4.2 shows examples of targets for doing the experiments and the possible output returned by our semantics-driven learning approach. This listing has two examples: (i) the first has a code fragment without annotation, and the expected output is to ADD the predicted annotation; (ii) the second example has a code fragment with annotation, and the expected output is to KEEP the annotation because it is correct.

```
// input shortlisted annotation
@RequestMapping

// example 1
// input training code fragment
@RequestMapping("/network")
private void findNetwork() {
}

// example 2
// input training code fragment
@RequestMapping("/network")
private void findNetwork() {
}

// example 3
// input training code fragment
@RequestMapping("/network")
private void findNetwork() {
}
```

Listing 4.1: Shortlist and Inputs

```
// example 1
// input target code fragment
@RequestMapping
private void findNetwork() {
}

// example 1
// output suggested action for annotation
ADD RequestMapping

// example 2
// input target code fragment
@RequestMapping
private void findNetwork() {
}

// example 2
// output suggested action for annotation
KEEP RequestMapping
```

Listing 4.2: Targets and Outputs

This approach addresses problems related to microservice construction. As identified in Chapter 1, developers often struggle with code comprehension due to outdated or missing comments. Additionally, the incorrect usage of annotations leads to errors that existing methods for detecting annotation issues do not offer predictive mechanisms for. This approach directly tackles these issues by leveraging semantics-driven learning from annotated code fragments to predict annotations.

4.2.1 Pre-processor

The pre-processor transforms raw data into training, validation, and testing sets. The raw data for training contains Java source code and the list of targeted microservice annotations. The first step is converting each Java file into its AST representation and splitting it into methods with attached annotations. The second step is to build two databases (Java and Queries) for the experiments. A Java database is required to find subsets and references for the predictor, similarity finder and adviser.

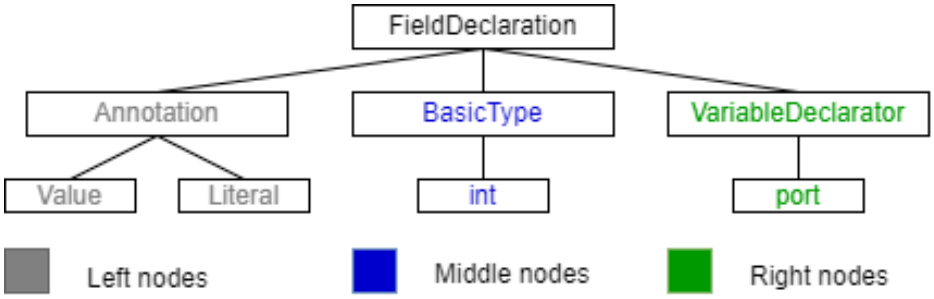


Figure 4.2: Nodes of an AST Representation

Natural Language Processing (NLP) is a series of techniques for understanding the meaning of text in a context. NLP provides tasks to automate translation, summarisation, classification and others [150, 45]. Tokenisation is a way of separating the text into words or tokens. Although tokenisation is a common NLP task, it is prone to miss part of the content when meaningful punctuation symbols are removed [30]. Additionally, it may produce the same sequence of tokens for two different class methods due to source code ambiguity [51]. However, tokenising the AST representation of the source code instead can overcome these issues, which usually is a tree that represents the syntactic structure of a source code.

Specifically, an AST contains additional semantic information inside a tree structure and a reading process expresses its nodes as a statement. Figure 4.2 presents the tree structure of an AST representation. *FieldDeclaration Annotation Value Literal BasicType int VariableDeclarator port* is the statement after reading the nodes starting with its root and following from the left to right. The most common reading process is the traversal algorithm which offers three ways to read the tree: inorder, preorder and postorder. From these, the preorder is the most employed to read an AST [143, 51] because it accepts (i) functions as arguments, and (ii) a variable number of arguments.

Pre-processor builds the AST representation by using a Python implementation of Java Lang Parser, which has been successful for the clone detection. The Java Parser follows the language syntax by tokenising the source code to identify the types, constructors,

members and expressions. Then, it creates a compilation unit as the root of the tree and inserts nodes according to the different declarations of the Java language such as imports, enums, classes, interfaces, fields, methods, annotations, arguments, parameters, and others. After building the AST, the Java Parser reads the tree and returns a string representation.

An AST representation has keywords to identify the definition of methods and attributes. A text parser detects the keyword *MethodDeclaration* for methods, and the annotations appear after *Annotation*. Listing 4.3 shows a Java code fragment with its corresponding AST representation shown in Listing 4.4 about how the pre-processor increases the amount of text. Splitting classes into methods increases the amount of samples and reduces the maximum size of each sample. The pre-processor reduces the execution time by loading databases simultaneously with multi-processing.

```
protected RestTemplate restTemplates;

@PostConstruct
private void postConstruct() {

    restTemplate.setErrorHandler(new
     DefaultResponseErrorHandler());

    restTemplate.setMessageConverters(
     httpMessageConverters.getConverters());
}
```

Listing 4.3: Source Code

```
FieldDeclaration ReferenceType RestTemplate
 VariableDeclarator restTemplate

MethodDeclaration Annotation PostConstruct
 postConstruct StatementExpression
 MethodInvocation restTemplate ClassCreator
 ReferenceType DefaultResponseErrorHandler
 setErrorHandler StatementExpression
 MethodInvocation restTemplate
 MethodInvocation httpMessageConverters
 getConverters setMessageConverters
```

Listing 4.4: AST Representation

4.2.2 Learner

The goal of this component is to learn the semantic information of a code fragment. The implemented learning process contains a series of actions that examine extracting features from data [70]. For this purpose, the component is similar to a learning task that generates a code description using models based on sequence, deep learning or graphs [73, 70, 115]. Specifically, a sequence-to-sequence-based learner is selected, considering its good performance in capturing the relation between code and text [51].

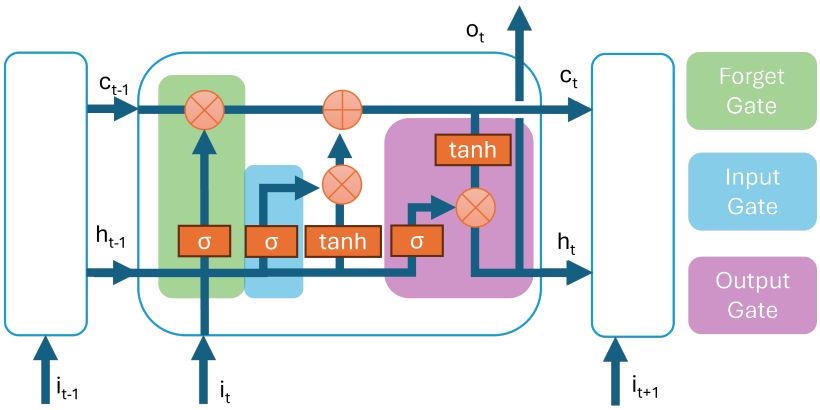


Figure 4.3: Neural Network Architecture

A sequence-to-sequence model reads tokens one by one. It learns the semantic meanings of a text file and uses this knowledge to generate multiple output tokens in a new sentence. This approach allows the generation of coherent and contextually appropriate sentences. The model generalises standard classification to multiple output variables, making it versatile for various applications [4]. Sequence-to-sequence models are particularly useful in tasks where the output is a sequence of tokens. Examples include language translation, text summarisation, and code generation.

The learner in this context comprises two sub-components: an encoder and a decoder. The encoder converts the code into vector space, capturing the semantic meaning of the input. The decoder performs the inverse operation, transforming the vector representation back into a sequence of tokens. Both components are intricately connected, allowing for the simultaneous training of both sub-components [51, 45]. This connection is crucial as it ensures that the learned representations are effectively utilised during decoding.

Encoder-Decoder Architecture

The architecture of the neural network is depicted in Figure 4.3. An encoder transforms the AST from a source code to vectors, which has a good performance for finding clones [4]. The

input of our encoder is the AST of code fragments that represents methods with or without annotations. Encoders could include attention, which is a mechanism that selects essential parts from the input and aligns code tokens with natural language words [51]. Additionally, the encoders with attention based on Recurrent Neural Network (RNN) extracts features from the text with better accuracy than Support Vector Machines (SVM) based on traditional methods [143].

There exist two ways of reading a sequence of tokens. The unidirectional schema processes the context in the same direction. In contrast, the bidirectional schema processes the tokens in both directions: from left to right and from right to left [45]. Our encoder takes advantage of the bidirectional schema when learning from the context forward and reverse. Additionally, the bidirectional schema captures more information due to three hidden states: one for each direction and one for connecting both directions [143].

Decoders aim to generate a sequence of tokens, i.e., in our case a sequence of annotations. However, selecting a predictor considers that decoders tend to have low accuracy [45]. In contrast, classifiers show promising results for clone detection. Additionally, the approach requires the name of annotations and high accuracy instead of a long sequence of tokens that could have annotations with similar names but different definitions.

Let $X = (x_1, x_2, \ldots, x_T)$ be the sequence of tokens representing the AST. The bidirectional RNN generates forward hidden states $\overrightarrow{h}_t$ and backward hidden states $\overleftarrow{h}_t$. This bidirectional RNN provides crucial contextual information that LSTM cells utilise to manage memory effectively and understand the context comprehensively. The following equations express the link between the forward and backward hidden states:

$$\overrightarrow{\mathbf{h}}_t = \mathrm{RNN}_{\mathrm{fwd}}(x_t, \overrightarrow{\mathbf{h}}_{t-1}) \qquad \overleftarrow{\mathbf{h}}_t = \mathrm{RNN}_{\mathrm{bwd}}(x_t, \overleftarrow{\mathbf{h}}_{t+1}) \tag{4.1}$$

Implementing an RNN for a sequence-to-sequence model that utilises LSTM cells

with two or more layers requires the calculation of the following gates: input, forget and output. These gates allow the model to selectively update, discard, and expose information for effective memory management. Additionally, the model requires the cell and hidden states to retain information and to understand the context across the sequence. Finally, the model requires a cell candidate to introduce and update new information into the cell state based on the current input and the previous hidden state.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{4.2}$$

The input gate i_t determines how much of the new information from the current input x_t and the previous hidden state h_{t-1} should be incorporated into the cell state. This gate uses a sigmoid activation function σ, which outputs values between 0 and 1, effectively deciding which parts of the new input to let through.

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{4.3}$$

The forget gate f_t controls which information from the previous cell state c_{t-1} should be discarded. Like the input gate, it uses a sigmoid activation function to scale the contributions of the previous cell state, thereby allowing the model to forget irrelevant information.

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{4.4}$$

The cell candidate g_t is created using a tanh activation function, which produces new candidate values to be added to the cell state. This step combines the current input x_t and the previous hidden state h_{t-1} to generate potential new information for the cell state.

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{4.5}$$

The output gate o_t decides how much of the cell state c_t should be exposed to the output. This gate also uses a sigmoid activation function to determine which parts of the cell state should influence the next hidden state h_t.

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \qquad h_t = o_t \odot \tanh(c_t) \tag{4.6}$$

The cell state c_t combines the previous cell state c_{t-1} and the new candidate values g_t, modulated by the forget gate f_t and the input gate i_t. The element-wise multiplication $\odot$ ensures that gates control the information flow appropriately. Additionally, the hidden state h_t is updated by applying the output gate o_t to the tanh-activated cell state c_t. This hidden state serves as the LSTM cell's output and the input for the next time step.

Finally, the attention mechanisms generate a context vector utilising the cell and hidden states. In this sense, the encoder is the right part of the model, the context vector is in the middle, and the left part is the decoder, which generates output tokens.

Vector Representation

Finding two pieces of code with similar behaviour is a time-consuming task, which usually transforms source code files into a vector space. Specifically, a vector representation is a mechanism that converts a code fragment to a single vector with key features [90]. The usage of vectors allows the application of math operations such as finding distance, grouping, and prediction over code or text. For instance, our bidirectional encoder returns vectors with dimensions NxM, where N represents the number of tokens and M the feature dimensions of each token.

Domain transformation is a mechanism that changes a text to a different domain, similar to a language translation. It has been used in software engineering to generate code comments from source code. In this context, the identification of annotations for a given source code domain can also be considered as a transformation problem that uses vector representations.

Pseudo Code for Training Sequence-to-Sequence

Listing 4.5 presents a pseudo code which outlines the training process of the sequence-to-sequence model with bidirectional RNN with attention:

```
Input: AST sequences {X}, Annotations {Y}
Output: Trained sequence-to-sequence model

Initialize: Encoder and Decoder parameters

for each epoch do:
    for each (X, Y) in training_data do:
        # Forward pass through the encoder
        for t in range(1, T+1):
            h_fwd[t] = RNN_fwd(X[t], h_fwd[t-1])
            h_bwd[T-t+1] = RNN_bwd(X[T-t+1], h_bwd[T-t+2])

        h = concatenate(h_fwd, h_bwd)

        # Attention mechanism
        for t in range(1, T+1):
            alpha[t] = softmax(h[t] . h)
            c[t] = sum(alpha[t] * h)

        # Forward pass through the decoder
        for t in range(1, T+1):
            y_pred[t] = Decoder(c[t], s[t-1], y[t-1])
            loss += compute_loss(y_pred[t], Y[t])

        # Backward pass and optimization
        loss.backward()
        optimizer.step()
    print('Epoch', epoch, 'Loss:', loss)
return trained_model
```

Listing 4.5: Pseudo Code

4.2.3 Predictor

This model predicts the best microservice annotations for a given source code. A classifier predicts whether an annotation is correct for a given code fragment. For the training process, each code fragment pairs with the elements of the annotation list. Suppose the original code has one annotation from the list. Here, the target in the predictor is a numerical representation of the target code fragment. The learner provides this numerical representation.

The decoder of a learner provides a sequence of tokens with words of similar meaning. However, the approach requires specific words for each annotation. Among the requirements for the predictor, the approach considers that (i) every query requires a subset of limited code fragments with similar features; (ii) fast classifiers that support non-linear boundaries with sensitivity to overfitting; and (iii) classifiers based on probabilistic are unacceptable due to their demand for more training data.

Numerical Representation

The numerical representation of the target code fragment is the vector that connects the encoder and decoder in the sequence-to-sequence learner. To understand it, Figure 4.4 simplifies the connection between each cell of the neural network architecture used by the learner. We observe that the cells on the left are the encoder; in the middle, the hidden states capture the contextual information into a numerical vector, also known as contextual vector; in the right size, there is the decoder to generate corresponding annotations.

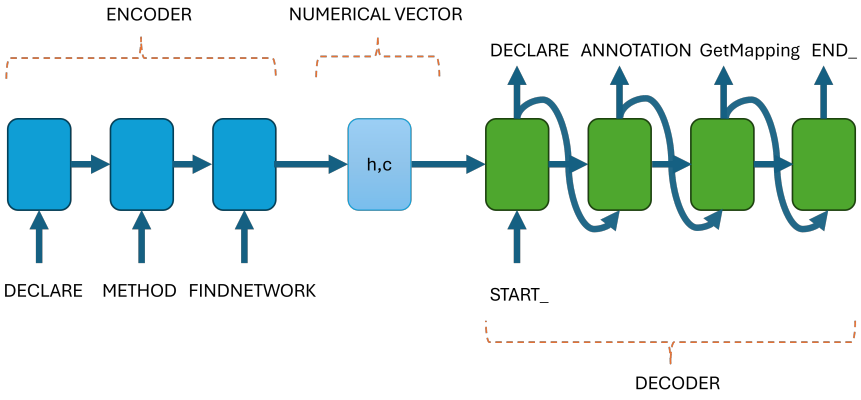


Figure 4.4: Encoder and Decoder

Figure 4.5 presents how the numerical vector is organised, it seems that each column is associated to each word. In this sense, each column of the numerical vector corresponds to a different feature or dimension that captures the semantics and structural aspects of the word inside the code fragment.

KNN Classifier

The key component of the predictor is a K-Nearest-Neighbour (KNN) classifier, which fulfils the above requirements. KNN predicts a class by (i) calculating the distance between the target and all the training points (subset), (ii) locating the K inputs which are closest to the target, (iii) calculating the probabilities of the target belonging to the classes of K inputs, and (iv) selecting the class of the neighbour with higher probability. KNN selection is due to its good results for predicting warnings [68].

Once the model is trained, KNN is fast for predictions due to its ability to memorise the training dataset and predict based on the closest data points. However, when the data size grows, KNN tends to be slow. Despite that, KNN is selected because it is one of the algorithms with high precision and recall [68]. Our approach deals with the scenario of big training data by introducing a constraint to use a dataset with a fixed size.

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ be the set of training code fragments, where each $\mathbf{x}_i$ is a numerical vector representation of a code fragment. The corresponding annotations are $\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$, where y_i is an integer representing the annotation for $\mathbf{x}_i$. Each annotation has a unique integer label associated with it.

Given a query vector $\mathbf{q}$, the KNN classifier finds the K nearest neighbours among the training vectors. The predicted annotation $\hat{y}$ for the query is determined by majority voting

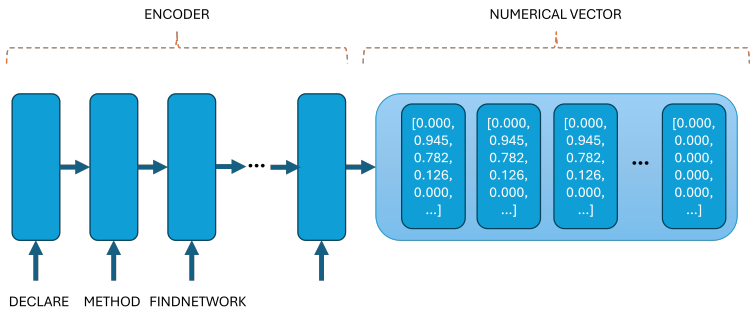


Figure 4.5: Numerical Representation from Encoder

among the annotations of the nearest neighbours. Mathematically, the $\text{KNN}(\mathbf{q}, K)$ denotes the set of K nearest neighbours of $\mathbf{q}$ as in the expression: $\hat{y} = \text{mode}(\{y_i \mid \mathbf{x}_i \in \text{KNN}(\mathbf{q}, K)\})$.

To ensure the efficiency of predictor, a subset of the Java database is used during training to mitigate the issue of large data sizes. This subset is selected based on code fragments that share similar features with the query. The selection of this subset depends on the Similarity Finder in Section 4.2.4.

Pseudo Code for Predictor Operation

The predictor performs training for every query; therefore, it is crucial to reduce the execution time. To accomplish this, a subset of the Java database is a key input for training. The predictor requires a K value to create a group of closest elements and a list of microservice annotations to encode their names. After the training, the predictor can receive the query vector and returns the predicted annotation as detailed in our replication package.

Listing 4.6 presents a pseudo code which outlines the operation of the predictor component:

```
Input: Query vector q, Training set X, Training labels Y, Number of neighbours K
Output: Predicted annotation y_hat

Function KNN_Predictor(q, X, Y, K):
    distances = []
    for i = 1 to length(X):
        distance = ComputeDistance(q, X[i])
        distances.append((distance, Y[i]))
    sorted_distances = SortByDistance(distances)
    nearest_neighbours = sorted_distances[1:K]
    y_hat = Mode(nearest_neighbours)
    return y_hat

Function ComputeDistance(a, b):
    return sqrt(sum((a - b)^2))
```

Listing 4.6: Pseudo Code

4.2.4 Similarity Finder and Adviser

Code search is a software engineer activity where developers employ natural language queries to find code fragments [4]. A similarity finder automates this activity and requires an unknown code fragment as a query. The goal of our similarity finder is the suggestion of annotations. Initially, the engine encodes the query to get its vector representation. Next, it selects a subset of vectors from the Java database to reduce the scope and execution time of the prediction. Finally, it communicates with the adviser to suggest actions for the query.

The selection of code fragments with similar features of the query vector is a crucial problem for the similarity finder and adviser. The comparison of the query vector against each code fragment from the Java database allows the approach to address this problem. The cosine similarity calculates the angle between two vectors. The ideal subset is a set of vectors with a similarity close to zero. However, there are several vectors with similar behaviours. This proposes the selection of the top closest vectors.

The query is a text that represents a single method, and it may include microservice annotations. The similarity finder utilises the Java database to get code fragments as one subset. Then, it calls the predictor with the query and a subset and returns an annotation. The adviser examine the query annotations and builds the possible actions of ADD or KEEP. Examples of responses would be *ADD RequestMapping* or *KEEP GetMapping*.

The selected actions (ADD or KEEP) for each annotation are the basis for future actions. The predictor returns the predicted annotations, and the adviser mainly predicts actions by comparing the original code fragment with the predicted annotations. If adviser returns no KEEP and ADD action, developers could modify the annotation. Thus, the approach is not limited to a small number of actions, and it focuses on the key ones. Listing 4.7 presents the pseudo code for this component, it has three functions one for each part (finder, adviser) and one function that reuse the previous two.

```
Function SelectSubset(Java_database, query_vector, M):
   distances = []
   for code_fragment in Java_database:
       fragment_vector = Encode(code_fragment)
       distance = ComputeDistance(query_vector, fragment_vector)
       distances.append((distance, code_fragment))

   sorted_distances = SortByDistance(distances)
   # Select the top M closest code fragments
   subset = sorted_distances[:M]
   return subset


Function Adviser(predicted_annotation, original_code_fragment):
    original_annotation = get_annotation(original_code_fragment)
    if original_annotation == predicted_annotation:
        return "KEEP"
    else:
        return "ADD"

Function SimilarityFinderAndAdviser(query, Java_database, predictor):
   // M is hyperparameter
   query_vector = Encode(query)
   subset = SelectSubset(Java_database, query_vector, M)
   predicted_annotation = predictor(query, subset)
   actions = Adviser(predicted_annotation)
   return actions + " " + predicted_annotation

```

Listing 4.7: Pseudo Code of Finder and Adviser

4.3 Evaluation

The study investigates how the semantics information of different code fragments helps keep or add annotations for unseen code fragments. Although the approach can learn from different programming languages, the selected language supports annotations from more than 109,000 open-source repositories related to microservices on GitHub. Most of those repositories (76%) adopt ten different languages. The inputs for the approach evaluation are based on the Java language, which is the top language used in 29% of the repositories. Specifically, this study aims to answer the RQ2 from Chapter 1: *How can we leverage semantic connections between code fragments and microservice annotations to predict annotations?*

Several experimental works on NLP fundamentals prove that inducing an RNN-based solution can lead to a better prediction when compared to traditional techniques (SVM,

TFIDF, SMT). Additionally, the selection of an RNN-based solution considers that more than 62% of those works implemented RNN-based techniques, as shown in Table 4.2.

Instantiated code fragments with and without annotations facilitate measuring the correct and wrong predictions returned by our approach. Calculations for the evaluation are (i) the **F1-Score** and **Accuracy** calculate the percentage of correct prediction of annotations to evaluate the predictor and measure the percentage of correct prediction of actions to evaluate the adviser, (ii) **Bilingual Evaluation Understudy (BLEU) Score** measures the quality of similar code fragments in the training subset and helps the evaluation of the similarity finder, (iii) **Confusion matrix** presents the false positive and false negative to evaluate the adviser by exploring the possible causes of wrong suggestions.

Accuracy is the percentage of correct values versus their total instances in the dataset. Precision is the percentage of correctly predicted positive values versus the total instances predicted as positive. Recall is the percentage of correctly predicted positive values versus the total actual positive instances. F1-score is derived based on the precision and recall of correct and incorrect values. BLEU Score is a measurement that quantifies the difference between an automatic translation and human-created reference translations of a source sentence [51], where a higher score means a higher similarity with the reference translation. The approach uses fragments of source code instead of text translation. A confusion matrix is a technique for summarising correct and incorrect predictions grouped by each class.

The approach considers a database of Java code fragments; thus, the availability of computing resources limits the maximum number of code fragments and annotations. The approach searches the top annotations in different open-source projects for evaluation purposes. The selected five annotations are related to microservices and have many records of small and medium-size to facilitate the database creation. These annotations are (i) *Bean* for returning objects of business logic; (ii) *Before* for preparing a test; (iii) *PostConstruct*

for executing after creating a new instance; (iv) *GetMapping* for exposing a method by a single URL; and (v) *RequestMapping* for directing a request toward a method.

4.3.1 Hyper-parameters

Hyper-parameters are variables that adjust the performance of our process for suggesting annotations. The sequence-to-sequence model requires: (i) the layers settings (number, arrangement, type) that receive and transform input; (ii) the dimensions to represent previous inputs (hidden states), continuous vectors (embedding) and training subset for one iteration (mini-batch); (iii) the learning rate that controls the updating speed; and (iv) the decay and dropout to reduce the loss and prevent over-fitting.

In particular, the predictor, similarity finder and adviser require: (i) the length of the vector, which affects the execution time during training, which means that a long number makes it slow, but a small number reduces its accuracy; (ii) the maximum number of closest vectors that reduces the scope of searching vectors with similar behaviours; and (iii) the number of neighbours, which shortens the classification.

In general, tuning techniques optimise the hyper-parameters and increase the performance. Standard tuning techniques include random search, genetic algorithm, and grid search. The genetic algorithm requires high computer performance to achieve reasonable accuracy. The random search is efficient in high-dimensional space. In contrast, grid search allows parallel execution with promising results for multi-label classification, prediction of the method name, and clone detection. Table 4.1 shows specific values for hyper-parameters.

The hyper-parameters were tuned based on a grid search, which works through multiple combinations of hyper-parameter values. The hyper-parameter values investigated by the grid search included various settings for dropout, vector dimensions, subset size for a maximum number of closest vectors, and the number of neighbours for the KNN. This thor-

Table 4.1: Semantics-Driven Learning Settings

Component	**Hyper-parameter**	**Range**	**Value**
Learner	Layers	N/A	LSTM
Learner	Hidden states and Mini-batch	N/A	100
Learner	Learning rate and Decay	N/A	0.9
Learner	Dropout	[0.1 - 0.5]	0.2
All	Vector dimensions	85x[32-256]	85x128
Similarity Finder and Adviser	Subset size	[10-50]	25
Predictor	K Neighbours	[5, 7, 9, 11]	7

ough search ensured that the chosen hyper-parameters delivered an optimal performance.

4.3.2 Experiment Setup

PyTorch is a Python-based machine learning library for NLP and deep neural networks. We extend PyTorch to train the learner and build vectors. The KNeighborsClassifier of the Sci-Kit Learn library allows the predictor to classify new code fragments. The similarity finder and adviser have a Java database of code fragments to identify corresponding pieces of code and provide suggestions for annotations.

The dataset comprises Java source code from GitHub repositories related to microservices with annotations. Specifically, it is possible to clone a few methods using keywords and distribute them into 20 experiments. The method adaptation occurs before replacing common words with unique keywords and specifying the number of new clones. Then, the cloning process replaces the keywords according to a list of words. For instance, if the code fragment says *public List<NAME> findAllNAME()*, its keyword is NAME and when we replace the keyword with the following words *[Student, Employee, Customer]*, we have the operations named as *findAllStudent, findAllEmployee, findAllCustomer*, each one returns *List<Student>, List<Employee>, List<Customer>*, respectively.

Additionally, A query database is built with correct annotations as the target for prediction based on the scenario of five annotations with a minimum of five references. A script kept or removed the annotations randomly. Then, the suggestion for a query is successful if the predicted annotation is correct.

The experiments performed well on two environments: (i) laptop and (ii) nodes. The training part of the learner runs in the nodes with GPU and CPUs, whereas the rest runs in the laptop environment. The laptop has a Core i7-4700MQ CPU at 2.4GHz with 16GB RAM. Nodes provide several CPUs/GPUs, and scripts allocate 5/35 CPUs and 1 GPU.

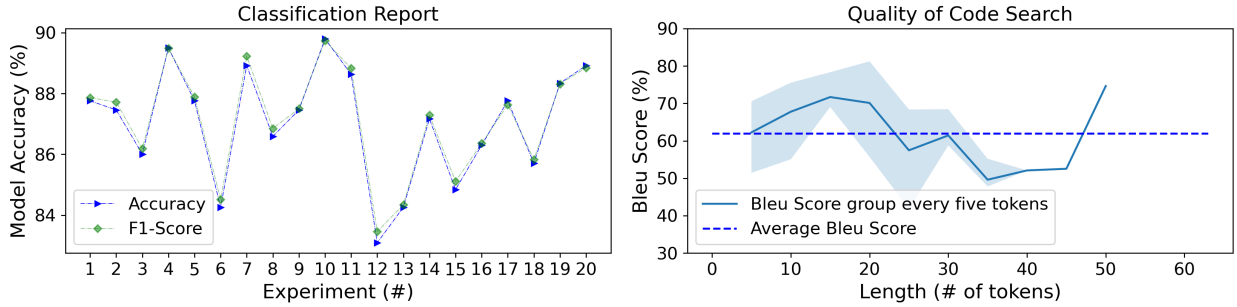
This section offers a replication package [1] which is publicly available for interested readers. The package includes the dataset for training, the databases and scripts to replicate and run the experiments. There are four main scripts: (i) training the learning model; (ii) building the databases for experiments; (iii) running the experiments; and (iv) plotting the charts. The source code is also available as a zip file with the following secret key: asus-RUBYcall2022. The replication package also includes more detail and examples of critical steps for the pre-processor, learner, predictor, similarity finder and adviser.

4.3.3 Results and Discussion

Scatter and box-and-whisker plots show the accuracy of queries, actions and annotations. Besides, a plot of the average bleu score per length shows the quality of searching code fragments. Additionally, a confusion matrix plots actions plus annotations to identify the miss-classification. The last chart helps compare the distance of queries and their subset in case of correct and wrong predictions.

Our approach achieves an accuracy between 83.09% and 89.8%, with an average

[1] https://bitbucket.org/semantics-driven-learning/replication-package/



(a) Performance Evaluation of the Model.

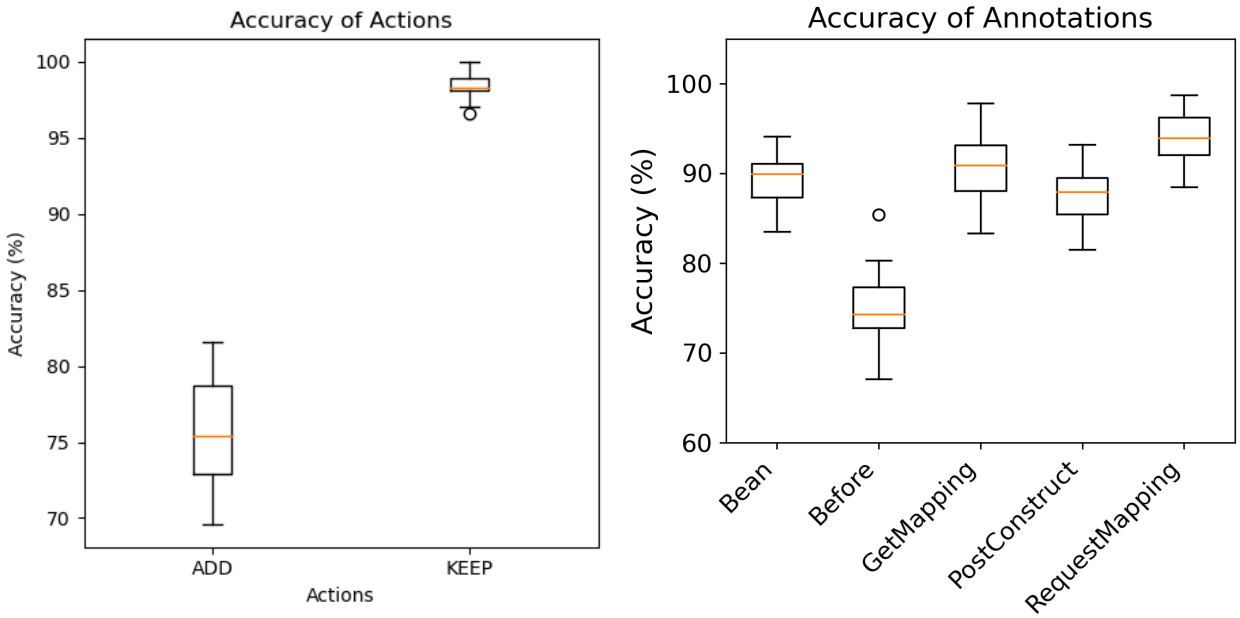
(b) BLEU Score by Vector Length

Figure 4.6: Performance and Quality of our Approach

of 87.03%. Figure 4.6a shows that 60.00% of the experiments are above the average. The difference between F1-score and accuracy is slight, i.e., 0.13% and 0.37%. The small difference means the precision and recall are good because of low false positives and negatives.

The approach assesses the quality of finding code fragments using the BLEU score, which evaluates one sentence against a reduced group of five references or sentences with similar features. Figure 4.6b shows the quality of good queries with different lengths grouped every five tokens. The score analysis considers the values in terms of small, medium and large queries, i.e., less than ten tokens, between 10 and 40 tokens, and more than 40 tokens, respectively. Large queries (nearly 23.33%) have an average bleu score of 61.87%, the medium queries (66.67%) have 63.69%, and the small queries (10.00%) have 62.26%. The average of all good queries is 63.12%, which is a good value considering that comment generation has 38% and translation language has an average of 41% [51].

The filled area in Figure 4.6b indicates that experiments have more queries with similar lengths (less than 40 tokens) but different values of bleu scores. The observations revealed bleu scores between 41.84% and 81.24%, consistent with the changes made in four of ten queries with lengths of less than 40 tokens. The changes including adding *System.out.println* sentences at the beginning or end of methods. Additionally, avoiding



(a) Accuracy of Actions.

(b) Accuracy of Annotations.

Figure 4.7: Accuracy of Actions and Annotations

System.out.println sentences in the Java database helped us to evaluate their quality, given those differences. Queries with lengths greater than 40 tokens have no fill area in the chart because the Queries database requires more similar code fragments with some differences such as *System.out.println*.

The accuracy shows the quality of the approach by counting its positive results. The elaborated box-and-whisker plots the percentage of correct predictions and focuses on two actions with five annotations, which open-source projects widely attach to their methods. Accuracy of actions from Figure 4.7a shows that *KEEP* action has an average of 98.41% and performs 15.06% better than *ADD* action. On the other hand, *ADD* has 74.56% on average and 81.56% as maximum. This difference is due to all the records in the Java database having annotations, increasing the distance between queries without annotations.

The experiment results indicate that our semantics-driven learning achieves an average accuracy between 74.95% and 93.92% for annotations and a general average of 87.26%. Our

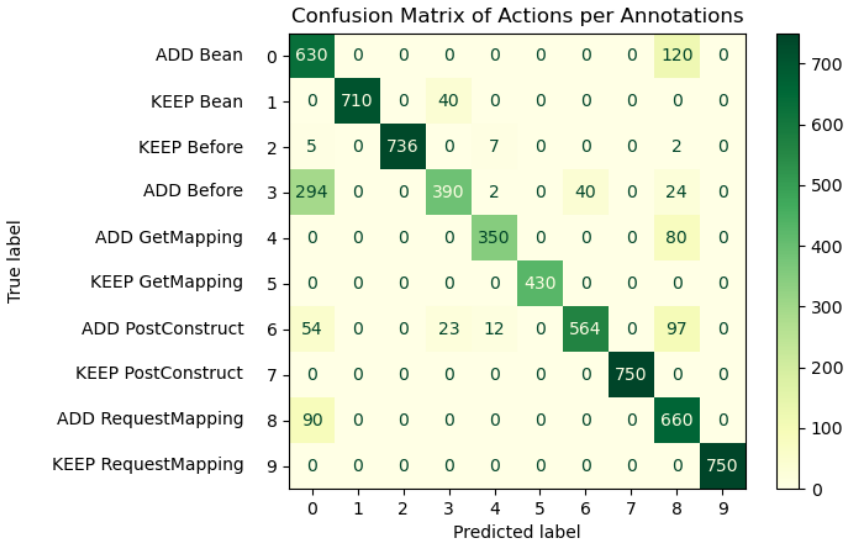


Figure 4.8: Analysis of Wrong Suggestions

results also show that the minimum accuracy of 67.11% is for *Before* annotation, while the unexpected annotations have 90.37% on average. Figure 4.7b also shows that *RequestMapping* and *GetMapping* have higher accuracy with an average above 90%. *Before* and *RequestMapping* have a minimum difference of 31.64% and a maximum of 3.11%.

The confusion matrix of actions plus annotations in Figure 4.8 shows that *ADD Before* has the lowest percentage of correct predictions. It is consistent with Figure 4.6b, and it mismatches against Bean (39.20%), PostConstruct (5.33%) and RequestMapping (3.20%). The first and second-highest percentage of wrong predictions are *ADD Bean* and *ADD RequestMapping* with 41.8% and 32.86%, respectively. To analyse possible causes of wrong suggestions, the evaluation also considers the intersection between two similar code fragments.

For the sake of discussing the data, the evaluation explores possible causes of wrong predictions with *Before* annotation. The observations show that wrong predictions occur for distances above 0.40. The green area indicates the expected annotations of the good

predictions are below 0.70 with distance ranges: (i) bellow 0.10 has 60%. (ii) between 0.10 and 0.45 has an average of 60%; and (iii) above 0.45 increase from 10% up to 39%. The red line refers to the unexpected annotations of the wrong predictions and shows that 80% occurs for a distance between 0.41 and 0.75. The blue line refers to the unexpected annotations of the good predictions between 40% and 60%.

According to the confusion matrix, *Bean* represents 35% of mismatches for *Before*, and *RequestMapping* is the second most mismatched annotation. The results revealed 58% of *ADD Bean* suggestions with three unique annotations for (i) returning new instances created with or without parameters; (ii) returning a local attribute or a default type such as integer or string; (iii) assigning a new instance to a local attribute; (iv) invoking at least one local method with parameter(s); (v) invoking static methods; and (vi) validating a condition/catch before throwing a new exception. Despite including different features, the AST representation excluded literal strings, which affected the balance of the subset.

Beyond the results of our experiments, our approach can reduce the misuse of annotations. We provided a quantitative way of indicating if a set of code fragments are different from a particular Java database and how different they are. Developers can thus identify which code fragments need adjustment to the usage of annotations. Additionally, they can increase the Java database to add new rules and extend them for other features such as parameters and their types.

4.3.4 Threats to Validity

We collected code fragments from open-source repositories and built a database of Java and queries to evaluate the suggestion of annotations. Thus, we present the threats to validity

of our evaluation according to the guidelines in [130].

External validity: A limited number of repositories may reduce the successful application of the approach. We mitigate it by splitting the source code into methods that increase the number of records for training. We cannot mitigate the generalizability of obtained results due to the size of the database. However, we leave a mechanism to increase data size for future work.

Internal validity: A subset of code fragments that share certain similarities with different annotations for a query introduce noise, reducing the number of correct suggestions. We mitigate this by reducing the number of code fragments as a subset. A noisy Java database of code fragments may increase wrong suggestions, which we reduce by cloning real-world code fragments. Another significant threat is using a single split for training, validation, and test sets. This single split can lead to overfitting and may not provide a robust evaluation of the model's performance. We mitigate this by randomly creating the validation test and building a pool of examples accessed in batch. Additionally, we report performance metrics averaged over multiple runs to account for any variability.

Construct validity: We mitigated the bias in the selection of weak projects by (i) using the star numbers and (ii) focusing on methods with standard annotations. We consider the bias in the manual construction of Java and query databases to assess the suggestions. We mitigate it by selecting simple code fragments, introducing small changes and checking similarity against others.

4.4 Related Work

This study focuses on the prediction of annotations in the area of microservices. Table 4.2 shows how this work fits in the state-of-the-art techniques by comparing key features such as AST, NLP, clone detection and others.

Table 4.2: Related Work

Features	**Related Work**										**Our Work**
	[93]	[94]	[92]	[90]	[135]	[68]	[148]	[30]	[137]	[150]	
Microservices	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓
Annotations	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓
AST	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓
NLP	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
RNN	✗	✗	✓	✓	✓	✗	✗	✓	✓	✗	✓
Extracting features	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗	✓
Feature detection	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓
Defect detection	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
Clone detection	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓
Bug detection	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓

This research differs from their work in two dimensions. First, the extraction process of methods recognises microservices and annotations by using the AST of the source code. Only two previous works focus on detecting microservice invocations and cycle dependency [93] and defining mutation operators to modify the declaration of annotations and apply mutation testing [94]. Second, the training process of a model that learns the usage of annotations to compare code fragments and suggests the best annotations.

Previous works used NLP in microservices projects for detecting different feature and clones of code fragments. Pigazzini et al. form groups of similar methods where distant groups mean different functionalities [92]. Perez et al. use a syntax tree-based skip-gram algorithm on different programming languages [90]. Unlike those works, our approach focuses on using microservice annotations by extracting terms from annotations and learning their

relation with the code.

Other previous approaches for bug detection applied in different contexts (no microservices) inspire us to apply NLP techniques such as (i) generation of sentences from source code for summarisation of its behaviour [135]; (ii) extracting pieces of text/comments/reports and features from source code for detecting bugs and warning analysis [68]; (iii) detecting defects by using AST [148, 30]; and (iv) detecting clones or similarities on code fragments [137, 150].

4.5 Summary

This chapter proposes a semantics-driven learning approach to suggest annotations according to the similarities between code fragments. The approach implements a Recurrent Neural Network (RNN) and a K-Nearest-Neighbour (KNN) classifier to learn the semantic relation of code fragments against their annotations and predict a suitable annotation, respectively.

This section concludes that using a database of rules based on code fragments with annotations is good enough to predict annotations and predict actions. The accuracy in the actions and the confusion matrix of suggestions, specifically for the ADD Bean and ADD Before, indicate that the approach needs for improvement when operations with similar behaviour are written for different purposes. Therefore, the purposes of using annotations is a dimension that researchers need to consider for prediction of annotations.

In our ongoing research, we are including the REMOVE action and introducing the analysis of multiple interconnected annotations. Additionally, we are incorporating other vector representations such as Flow2vec for advanced features. A research on the overlapping and imbalance of the training subsets for predictor could help to investigate if increasing the distance between code fragments. Improving the overlapping and imbalance could reduce the number of wrong suggestions.

Chapter Five

A Mining Approach to Limit Granularity of Annotated Operations

5.1 Overview

The microservice architecture style is a software development approach that advocates independent units of development based on lower coupling and higher cohesion [53]. Coupling is the degree of interdependence or interconnection between microservices, while cohesion is the degree of relationship between the responsibilities assigned to a microservice. Granularity is the detail level of services presented in applications. In general, it can be defined by the number of services with their operations complexity and dependencies [53]. Among others, microservice granularity can be measured by the number of published operations, microservices, code lines, and complexity (e.g. the number of tokens).

However, microservice-based applications have different granularity because there is no agreement on the right granularity, which can produce issues. For instance, applications with tiny microservices introduce managing issues into the whole architecture. On the other hand, large microservices affect the system performance and reduce the overall system quality

[121]. The selection of adequate granularity is not trivial since there is a trade-off between microservice maintainability and computational efficiency. Then, inadequate granularity can introduce defects where fixing them is time-consuming. Moreover, the detection effort to solve the above issues and debugging microservices may take days or weeks [131].

The novel contribution of this chapter is a semantics-driven learning approach for searching operations with semantics similarity and mining the granularity operations. The learning process pursues building a vector space for operations with annotations that facilitate the identification of granularity by segregating similar operations. The approach contributes to the fundamentals of microservice granularity, where this work is the first to cluster similar operations to reduce the amount of microservice issues related to excessive invocation due to fine-grained services or time response due to coarse-grained services. This chapter aims to obtain an empirical answer to RQ3 stated in Chapter 1: *How can annotations contribute to the understanding of typical granularity degree within existing microservices?*

While creating statistics tables for each annotation might provide some insights, this approach is limited in capturing complex patterns and relations within the data. On the other hand, a semantics-driven learning approach offers a better understanding of the intricate relation among annotated operations. By employing this technique, we identify the typical granularity of operations and suggest granularity limits based on analysing various operations with similar behaviour. This learning method enables a more dynamic and accurate approach to optimising microservice architectures by addressing granularity issues more effectively than using only traditional statistical methods.

The research has shown that annotation issues are a top developer concern [96]. Especially annotation issues related to evaluating test cases and detecting bad smells in microservices [94, 93]. Previous studies have mentioned the importance and methods of determining the appropriate granularity [41, 105, 122]. However, a systematic compilation of

operations for granularity analysis has been lacking. In this context, this approach is the first to explore and identify the average granularity values for similar operations. Furthermore, GitHub selection considered its popularity as an open-source software hosting platform that allows the collection of millions of methods for building datasets [3, 95].

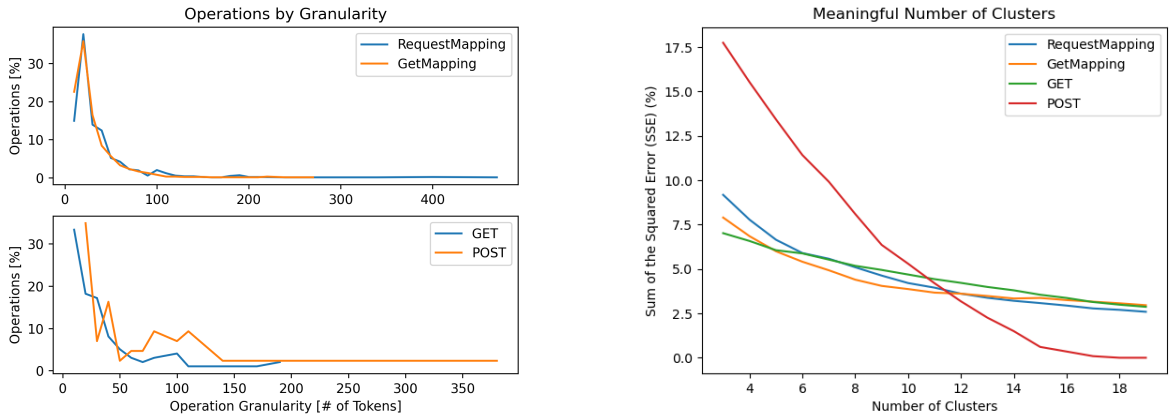
The organisation of the rest of the chapter follows: Section 5.2 explains the components of the approach. A report of the evaluation is in Section 5.3 with results and discussion, followed by an analysis of related works in Section 5.4. Finally, Section 5.5 concludes the chapter and outlines future research.

5.2 Proposed Approach

5.2.1 Operations with Annotations

Operations are units of functionality or actions within a microservice-based application. Operations encapsulate specific functionalities that microservices expose. The functionalities are often associated with handling HTTP requests, processing data or managing resources. To implement these functionalities, developers frequently employ libraries that provide annotations. These annotations support the definition and management of operations more effectively. Annotations are metadata added to source code to convey additional instructions. In the context of microservices, annotations facilitate the development of applications by defining endpoints and operations for handling incoming HTTP requests.

Microservice frameworks play a vital role in simplifying the implementation of cloud-based applications by offering annotations. These annotations support the reuse of features and software evolution, aligning with developers when following coding style guidelines [28]. Consequently, consistent usage of annotations appears in operations that share semantics



(a) Operations by Granularity

(b) Clustering Error Curves

Figure 5.1: Operation Selection and Granularity Exploration

similarities. In general, annotations enable specific features to methods, classes and fields of Java source code. To explore and identify these annotations, the Java Parser library is a valuable resource, allowing for efficient searches for annotations attached to operations.

5.2.2 Abstract Syntax Tree of Operations

An Abstract Syntax Tree (AST) is a tree representation of source code with additional syntax and semantic information [30]. ASTs show good results for clone detection by recognising code fragments with a similar sequence of tokens [90]. Java Parser library help builds the ASTs for detecting annotations. Thus, the approach can convert the sequence of tokens into a vector representation for clustering operations.

A critical task of the approach is the selection of microservice operations for training. To ensure comprehensive coverage, the selection process focuses on operations containing up to 85 tokens, a granularity limit that includes most of the operations with annotations, as shown in Figure 5.1a. Additionally, the selection process extends to annotations by checking the top ones and selecting those that publish operations like @RestController and @GetMap-

ping which creates a Web service by mapping HTTP requests to a specific operation.

Initially, the approach requires an extensive collection of Java code fragments coming from more than 30K public GitHub repositories. After converting the sequence of tokens into a vector representation, our approach creates clusters of similar operations. We choose a base of 10 clusters considering the decay in the clustering error curves, which provides insights into the performance of a clustering algorithm, as shown in 5.1b.

In general, a reading process expresses AST nodes as a statement. The most common reading process is the traversal algorithm which offers three ways to read the tree: inorder, preorder and postorder. From these, the preorder is the most employed to read an AST because it accepts (i) functions as arguments, and (ii) a variable number of arguments [143, 51]. To convert AST statement to tokens, the approach use the tokenisation task from Natural Language Processing (NLP). A token is essentially a numerical representation of an AST element.

5.2.3 Approach Components

The relation between operations and annotations is crucial for exploring granularity. In this sense, our approach involves collecting code fragments, extracting their annotations, and analysing them. Next, we categorise operations based on their associated annotations and calculate the granularity of operations. The analysis focuses on understanding the granularity utilised by the developer community when attaching annotations to operations. In particular, the semantic information of operations enhances the search for similar operations, thereby improving the analysis.

The principal components of our approach are Operation Miner, Operation Converter, and Limits Finder. Each component plays a specific role in the process:

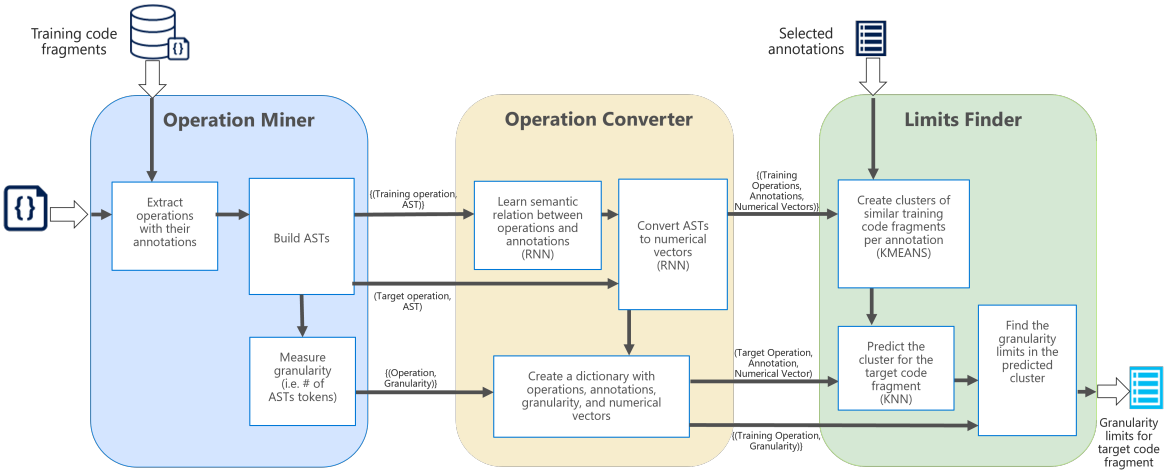


Figure 5.2: The Components of our Approach

- **Operation Miner** prepares the operations with their connections, annotations, and granularity metrics. It gathers code fragments from the input datasets and identifies the annotations associated with each operation. It calculates granularity metrics based on the number of tokens from the AST representation of each operation.

- **Operation Converter** learns semantics from operations and converts them into vector representations. This component utilises a sequence-to-sequence model to build the numerical vector of each operation. It stores operations, their annotations and their granularity. This storage is helpful for the next component to identify the closest operations with similar behaviour.

- **Limits Finder** creates clusters according to the similarity between annotated operations before searching unseen operations to the closest group. To build clusters, it utilises a K-Means algorithm, while it uses a KNN algorithm to predict the correct cluster. This choice allows the approach to dynamically and accurately assign new operations to existing clusters based on their proximity to the nearest neighbours. The Limits Finder then calculates the granularity limits for each cluster, defining the typical range of granularity values.

The approach requires code fragments as inputs: one Java dataset of microservice applications and one Java database of clone operations. The approach simplifies the inputs by requiring a string representation for dataset and database paths. Additionally, we need the two output paths for (i) the microservice operations extracted from the Java dataset; and (ii) mapping files generated after each stage. Finally, we propose the following format for the limits of granularity: (i) operation group, (ii) annotation, and (iii) granularity range.

Operation Miner

The goal of Operation Miner is to pre-process the raw files from the Java dataset. This component gets, organises, and formats the operations before continuing with the other components. The Miner executes three essential functions: (i) extracting operations with their annotations, (ii) building their Abstract Syntax Tree (AST), and (iii) measuring their granularity. These steps result in a structured dataset of operations along with an index file that specifies the source code, annotations and their connections.

Extract Operations: The initial task of the Miner is to collect the code fragments from open-source repositories. First, we need to download complete source code files from different projects. Second, the Miner must identify Java files with annotations. Third, the Miner needs to split the files into code fragments. Next, we select only code fragments that correspond to operations. Then, the Miner maps the selected operations according to their projects, packages, and classes.

Build Abstract Syntax Trees (ASTs): The complex task of the Miner is to identify the operations due to the different mechanisms and frameworks that expose functionality over an HTTP request. The Miner needs to locate the source code of each operation with their annotations. To identify the microservice operations, the Miner focuses on specific annotations that expose the functionality through an HTTP request. For instance, any code

fragment with the RequestMapping and GetMapping annotations is a microservice operation. After identifying operations, the Miner constructs ASTs for these operations to capture their structure and semantics. It utilises a Java Lang library to parser the operations, which reads the source code and constructs the corresponding AST.

Measure Granularity: Granularity measurement involves quantifying the size of each identified operation. The Miner groups the operations by their packages, each representing a microservice. Then, the component counts operations by microservice and their lines of code. Additionally, the Miner measures the lines of code per operation and holds the granularity information for the next component, the Operation Converter, when it is required to attach to the vector representation of the operations.

```
1
2  // 1. Example of annotated operation
3  @RequestMapping("/home")
4  public String welcome() {
5      return "Welcome " + user.getName();
6  }
7
8  // 2. Example of AST representation as string
9  String ast = "MethodDeclaration Annotation RequestMapping Literal ReferenceType String welcome
       ReturnStatement BinaryOperation + Literal MethodInvocation user getName";
10
11 // 3. Example of its granularity
12 String[] tokens = ast.trim().split("\\s+");
13 granularity = tokens.length;
14
15
```

Listing 5.1: Example of Annotated Operation and Its AST

Listing 5.1 shows the first example of an annotated operation that returns a welcome message with the user name. The second example corresponds to its AST representation assigned to a string variable. The third and last example calculates its granularity by splitting the string into tokens, which are words separated by blank spaces. The granularity is the number of tokens that form its AST representation. Listing 5.2 presents the pseudo code of this component.

```
1
2  class OperationMiner:
3      def __init__(self):
4          pass
5
6      def mine_operations(self, java_files):
7          operations = []
8          for file in java_files:
9              code_fragments = self.extract_code_fragments(file)
10             annotated_operations = self.identify_annotated_operations(code_fragments)
11             for op in annotated_operations:
12                 ast = self.build_ast(op)
13                 granularity = self.measure_granularity(ast)
14                 operations.append({
15                     'code': op,
16                     'ast': ast,
17                     'granularity': granularity,
18                     'annotations': self.get_annotations(op)
19                 })
20         return operations
21
22     def extract_code_fragments(self, file):
23         code_fragments = []
24         # Open file, get content, split content into code fragments
25         if fragment isMethod:
26             code_fragments.append(fragment)
27         return code_fragments
28
29     def identify_annotated_operations(self, fragments):
30         annotated_operations = []
31         for fragment in (fragments)
32             if fragment isAnnotated:
33                 annotated_operations.append(fragment)
34         return annotated_operations
35
36     def build_ast(self, operation):
37         # Use the Java Lang Parser library
38         ast_tree = parser.parse("@UnknownAnnotation \n class Unknown { " + operation + " } ")
39         ast = parser.convert_to_text(ast)
40         return ast
41
42     def measure_granularity(self, ast):
43         String[] tokens = ast.trim().split("\\s+");
44         granularity = tokens.length;
45         return len(ast)
46
47     def get_annotations(self, operation):
48         # Logic to extract annotations from an operation
49         return annotations
50
51
```

Listing 5.2: Pseudo Code of Operation Miner

Operation Converter

The goal of our Operation Converter is to store in memory the relation between the code fragments, their annotations and the granularity associated with each operation. This component gets the AST representation and converts operations into vectors. The Converter executes three essential functions, which are: (i) learn the semantic relation between operations and annotations, (ii) convert ASTs to numerical vectors, and (iii) create a dictionary that captures operations, annotations, granularity and numerical vectors. Additionally, the Converter considers the static characteristics of a code fragment. This component also considers the granularity that belongs to a single operation instead of the whole system.

Learn Semantic Relation: The primary function of this component is to learn relations between operations and annotations through semantics-driven techniques, which extract features from a text by focusing on its syntax and semantics. Our semantic learning produces a sequence-to-sequence model that can read a text word by word to learn the semantic meanings [4]. To perform this functionality, a Recurrent Neural Network (RNN) employs the AST representation as a text of code fragments to discern patterns and connections. This component requires a Java Parser, a library that reads the Java files and provides an AST structure to work with Java code in a programmatic way.

Convert to Numerical Vector: The core functionality of the converter is to transform ASTs of operations into numerical vectors through the application of semantics-driven learning. The previous function produces an encoder that can convert the AST representation of a new operation into vectors. The approach needs the conversion step, considering that similar operations could require similar annotations, and the approach intends to detect operations with similar behaviour by converting the operations into vectors and calculating the distance between vectors. Two operations are similar if the distance between their vectors is closest to zero.

Create a Dictionary: After getting the vector representation of operations, this component converts the annotation names into vectors with one dimension. Then, the converter adds the annotations as additional information to the vector representation of operations. The converter uses an unsupervised learning technique to create clusters considering the cosine similarity between vectors. Additionally, each cluster has internal groups by their annotations. This way, the converter can organise data by operation similarity, annotation, and granularity.

Listing 5.3 presents the pseudo code of the Operation Converter as a class to facilitate the understanding of this component.

```
1
2  class OperationConverter:
3      def __init__(self):
4          pass
5
6      def convert_operations(self, operations):
7          vectorized_operations = []
8          for operation in operations:
9              vector = self.convert_to_vector(operation['ast'])
10             vectorized_operations.append({
11                 'vector': vector,
12                 'annotations': operation['annotations'],
13                 'granularity': operation['granularity']
14             })
15         return vectorized_operations
16
17     def convert_to_vector(self, ast):
18         # Logic to convert AST to numerical vector
19         # Here we use the encoder from the semantics-driven learning approach.
20         return vector
```

Listing 5.3: Pseudo Code for Operation Converter

Limits Finder

The Limits Finder component is responsible for detecting operations with similar annotations and determining the granularity limits for these annotated operations. This process involves several steps and utilises both the K-Means and KNN algorithms. Using these algorithms is possible by measuring the distance between two vectors.

First, we measure the distance between vectors representing code fragments to find

similar operations. Various distance measures are available in Machine Learning, such as Hamming, Euclidean, Manhattan, Minkowski, Cosine Similarity, and others. We use Cosine similarity due to its effectiveness in clone detection of code fragments in different programming languages [150]. Cosine similarity returns a number between zero and one, where zero means identical vectors (high similarity), and one indicates completely different vectors. To ensure meaningful similarity, we filter out operations that are too dissimilar using a specific threshold (e.g., 0.70).

Using Cosine Similarity, we reduce the search scope by finding the nearest vectors representing similar operations. We use the K-Means algorithm to build clusters from a subset of the closest vectors. The K-Means algorithm partitions the data into a predetermined number of clusters, grouping annotated operations. If one operation has more than one annotation, we could join the annotations in pairs to capture their combined effect.

To check clusters, we count operations per annotation within each cluster and calculate the error as the distance from each operation to the cluster centre. We select the k number of clusters by minimising this error. This method is commonly used in unsupervised learning when the error decreases as the number of clusters increases.

After clustering, we assign each operation a unique cluster identification (cluster ID). Then, we use the KNN algorithm to predict the cluster membership of new or unseen operations based on their similarity to existing operations. We use cosine similarity to fetch a subset of operations and their corresponding cluster IDs. The KNN algorithm performs the prediction by using the K cluster members that are most similar to an unseen operation, leveraging the learned similarity patterns from the training data. This step helps generalise the clustering to operations not seen during the initial clustering phase.

Finally, we filter the operations within each predicted cluster to determine the granularity metrics. These metrics are ordered from minimum to maximum, allowing us to define

the granularity limits for each unseen operation. The granularity limits are determined by the range of granularity values observed within each cluster. We avoid the outliers and use the 0.25 and 0.75 percentiles to limit the typical granularity of operations. This process provides insights into the typical granularity of operations associated with each cluster.

The primary output of the Limits Finder is a set of granularity limits for unseen operations, which helps to guide the effective use of annotations in microservice development and maintenance. Listing 5.4 presents the pseudo code of this component as a class.

```
class LimitsFinder:
    def __init__(self):
        pass

    def find_limits(self, vectorized_operations, unseen_operation):
        vectors = [op['vector'] for op in vectorized_operations]
        kmeans = KMeans(n_clusters=self.get_k_clusters())
        kmeans.fit(vectors)
        clusters = kmeans.predict(vectors)

        cluster_dict = defaultdict(list)
        for i, op in enumerate(vectorized_operations):
            cluster_dict[clusters[i]].append(op)

        knn = KNeighborsClassifier(n_neighbors=7)
        knn.fit(vectors, clusters)

        granularity_limits = self.calculate_granularity_limits(cluster_dict)
        return knn, granularity_limits

    def get_k_clusters(self):
        # Logic to determine the number of clusters
        return k_clusters

    def calculate_granularity_limits(self, cluster_dict):
        granularity_limits = {}
        for cluster, ops in cluster_dict.items():
            granularities = [op['granularity'] for op in ops]
            limits = np.percentile(granularities, [25, 75])
            granularity_limits[cluster] = limits
        return granularity_limits

    def predict_cluster(self, knn, new_operation_vector):
        return knn.predict([new_operation_vector])[0]
```

Listing 5.4: Pseudo Code for Limits Finder

Pseudo Code of the Approach

Listing 5.5 presents a pseudo code which outlines how the approach invoke the different tasks of the Operation Miner, the Operation Converter, and Limits Finder:

```

java_dataset_path = "..."  # contains the path to the java operations
java_files = "..."  # contains the path to the java operations
clone_operations_path = "..." # contains the path of operationes for experimentation
op_miner = OperationMiner()
op_converter = OperationConverter()
limits_finder = LimitsFinder()

def execute_approach():
    java_files = load_files(java_dataset_path)
    operations = op_miner.mine_operations(java_files)
    vectorized_operations = op_converter.convert_operations(operations)

    new_operations = load_files(clone_operations_path)
    for operation in new_operations:
        ast = op_miner.build_ast(operation)
        vector = op_converter.convert_to_vector(ast)
        knn, granularity_limits = limits_finder.find_limits(vectorized_operations,
 operation)
        cluster = limits_finder.predict_cluster(knn, vector)
        limits = granularity_limits[cluster]
        print(f"Granularity limits for operation: {limits}")

def load_files(path):
    # Logic to load Java files from a given path
    return files
```

Listing 5.5: Pseudo Code of the Approach

5.3 Evaluation

The experiment measures the relation between annotations and the granularity values for a group of operations with similar behaviour. Additionally, the experiment considers the usage granularity when developers build operations with a catalogue of annotations. The granularity analysis of distances between the limits for fined-coarse and grained-coarse evaluates the approach's effectiveness by counting the percentage of operations inside the range of granularity limits.

To evaluate our approach, we build a dataset and database similar to the previous chapter and collect several annotated operations from open-source repositories. We analyse the distribution of operations by annotations to select which annotations have a representative number of operations. With those operations, we create similar operations with minor changes to build the database of annotations operations. We randomly split the database of 10674 examples to build training (45%), validation (45%) and test (10%) datasets. For experimentation, we use operations from different open-source repositories; this helps to identify if our approach has overfitting.

We mainly evaluate the Limits Finder by selecting operations with annotations and calculating their granularity using the number of tokens in their AST representation. Specifically, we consider operations that contain a maximum of 85 tokens. Using this data, we build a searchable database to find similar operations, create clusters, determine their granularity range of limits, get their granularity values and locate them within the corresponding range.

The evaluation of the approach utilises the following metrics: (i) the *Limits* evaluates the Limits Finder component by plotting the operations between the 25 and 75 percentiles (granularity range), (ii) the *F1-Score* evaluates the overall approach by calculating the precision and recall of correct and incorrect clustering prediction, (iii) the *Accuracy* evaluates the overall approach by calculating the percentage of correct clustering prediction in comparison with their total, and (iv) the *Distance Analysis* evaluates the Limits Finder component by showing the distance range of a cluster and plotting the distance between good and wrong predictions.

We chose a list of three annotations that most appears in different open-source projects. These annotations are RestTemplate, GetMapping, and RequestMapping. We use GetMapping and RequestMapping to identify operations and the RestTemplate to identify operation connections. Our scenario evaluates if the operations have their granularity

in the correct range. The range could have low, medium and high granularity where two thresholds separate the limits.

5.3.1 Experiment Design

We prepare a Java dataset by cloning the source code from GitHub open-source repositories selected with the following criteria: (i) Java projects; (ii) microservices; (iii) more than 300 stars. We search with Java Parser for RequestMapping, GetMapping and RestTemplate annotations to detect operations and connections between them. Our text parser reads the selected Java files and splits their content into operations. We choose the operations with annotations, generate their AST representation and calculate the length of operations. Additionally, our learning model needs three datasets of selected operations for training, validating and testing its encoder. Then, we add to our dataset the vector after converting the operations with the encoder.

We explore the granularity metrics of operations for the evaluation by grouping them according to their package and class. We elaborate a Java database with a subset of operations that were modified. Our database will have successful operations within a granularity range. We generate the range by adding new sentences based on System.out.print. Thus, we increase the length and keep the actual behaviour of the operations at the same time. Other mechanism to increase the length is adding print lines inside new conditions.

We evaluate the scenario where developers follow development guides for building microservice applications. Thus, new operations with similar features should follow the specifications, such as the operation length when implementing (ABC behaviour). We identify first the operation behaviour and then its annotation. Then, we search inside our database for other similar operations to know their granularity range. Thus, if our new operation has a granularity inside the range is a successful operation; otherwise, it is a unsuitable operation.

5.3.2 Hyper-parameters

Hyper-parameters are variables to adjust the overall performance of the three main components of our approach. The Operation Miner requires the number of cores to speed the processing of the dataset and build the corresponding database. The Operation Converter requires the dimensions for setting the deep learning with the hidden states, continuous vector embedding, and mini-batch. Additionally, the Limits Finder requires the lower and higher thresholds for the granularity limits.

We optimise these hyper-parameters using standard tuning techniques such as random search, genetic algorithms, and grid search. Random search and genetic algorithms are efficient in high-dimensional space and consume high computer processing to achieve reasonable accuracy. Thus, we chose grid search due to the parallel execution and the promising results for clone detection and multi-label classification. Table 5.1 shows specific values for hyper-parameters of our approach.

Table 5.1: Limits of Granularity Settings

Component	**Hyper-parameter**	**Range**	**Value**
Operation Miner	Cores	5-35	35
Operation Miner	Vector dimensions	85x[32-256]	85x128
Operation Converter	Hidden states	N/A	100
Operation Converter	Mini-batch	N/A	100
Operation Converter	Vector dimensions	85x[32-256]	85x128
Limits Finder	Vector dimensions	85x[32-256]	85x128
Limits Finder	Low thresholds	N/A	below 25%
Limits Finder	High thresholds	N/A	above 75%

5.3.3 Experiment Setup

We implement the Miner with Java and Python libraries to extract the operations. Specifically, the Java Parser Lang library helps identify connections between operations, while Python Java Parser builds the AST representation. Additionally, we implement semantics-driven learning by extending PyTorch, a Python-based library for NLP and deep neural networks. We train the learner for the Operation Converter and then build the vectors using its encoder.

We perform the experiments mainly on a laptop with a Core i7-4700MQ CPU at 2.4GHz and 16GB RAM. We also speed the training of semantics-driven learning in the environment of the nodes, which provides several CPUs. We set up our scripts to run up to 35 CPUs in the nodes. The process of parsing the dataset runs in the laptop environment. The process with a subset of 25 unseen operations runs in the laptop environment, which loads the vectors database partially due to the limitation of 16GB RAM.

We provide a replication package for interested readers [1]. The replication package has a zip file with all the open-source projects related to microservices from GitHub. The package also contains the datasets, databases and scripts to run the experiments. There are four main scripts: (i) mining the dataset; (ii) training the learning model; (iii) running the experiments; and (iv) plotting the charts. The source code of our Operation Miner, Operation Converter and Limits Finder is available as a zip file with the secret key: miningLIMITS2022. Our replication package also includes more detail and examples of the critical steps for our approach: operation miner, operation converter and limits finder.

[1] https://bitbucket.org/mining-granularity-limits/replication-package/

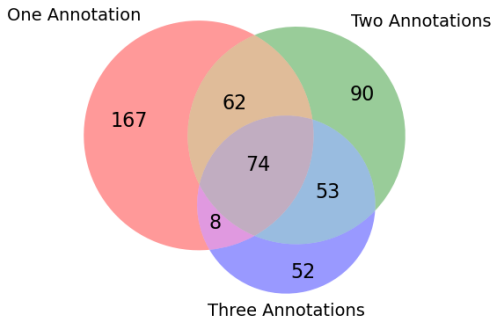


Figure 5.3: Isolated and Linked Annotations

5.3.4 Results and Discussion

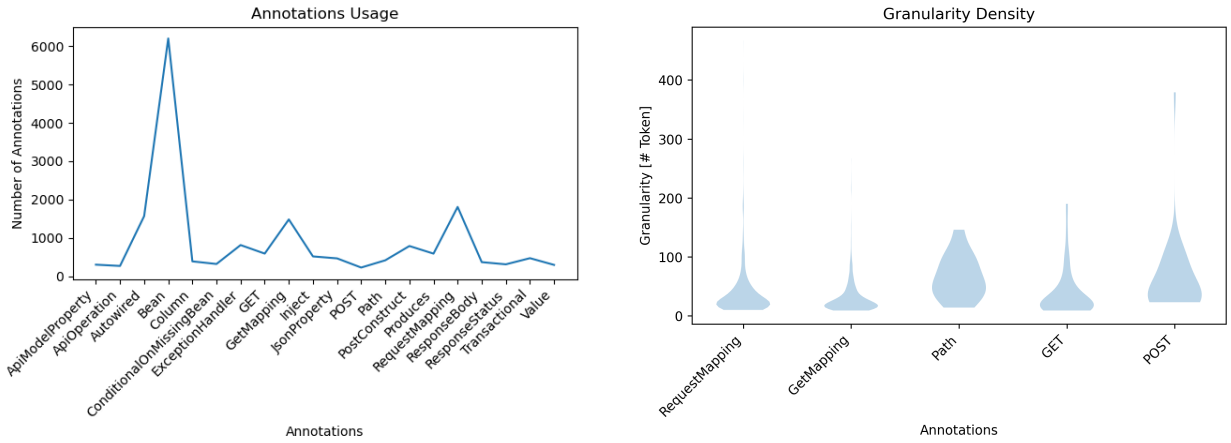
We draw Venn diagrams to show code fragments with one, two and three linked annotations. A plot scatters of the annotation usage counts the number of code fragments for each annotation. A violin plot presents the range of granularity and compares the relation between the number of lines and tokens. The distribution of tokens usage allows us to select operations based on their number of tokens. The clustering error curves and the box-and-whisker plot help observe the granularity limits per cluster. We plot scatter to show the performance of our approach when matching operations with their corresponding cluster.

After collecting open-source repositories from GitHub, we identify 143,341 Java files containing 264,531 potential code fragments. From this dataset, only 20,540 code fragments contain 506 unique annotations. Figure 5.3 illustrates how developers connect those unique annotations. The data shows that developers use annotations in various combinations. This figure is a Venn Diagram depicting the number of unique annotations attached to operations, categorised into One Annotation, Two Annotations or Three Annotations. For instance, the *RequestParam* annotation is often utilised alongside the *RequestMapping* annotation, demonstrating how multiple annotations are combined in practice.

Software Engineers(SEs) generally utilise annotations for various microservices-related tasks, like publishing services, consuming services, accessing database structure, and returning processed information. In this context, SEs may reuse the collected annotated operations to detect similar operations for different purposes. Here are three possible scenarios where SEs could apply our approach to the collected operations: (i) improving recommendation systems by suggesting code fragments to developers for completing operations; (ii) enhancing code refactoring tools by detecting similar operations and automatically merging them, and (iii) optimising code review processes by highlighting similar patterns across different projects. The fourth scenario involves determining the limits of the granularity of operations, which results are discussed in this section.

We observe that 33% are unique in the group *One Annotation*; while the groups *Two Annotations* and *Three Annotations* has 18% and 10% exclusive linked annotations. Developers combine 24% of annotations: (i) 12% of annotations belong to the combination between *One Annotation* and *Two Annotations*; (ii)10% represents the combination between *Two Annotations* and *Three Annotations*; and (iii) 2% corresponds to the combination between *One Annotation* and *Three Annotations*. The 15% remnant corresponds to the combination of the three groups. Thus, we can say that developers may link 49% of annotations in pairs of two. We focus on operations with one annotation because it has the most representative number of operations.

Figure 5.4a shows the usage of the top 20 annotations according to the number of code fragments where they appear. These 20 annotations appear near 69% of code fragments. Although *Bean* is the most common annotation with more than 6,200 code fragments, we exclude it because our scope focuses on operations. According to our expectation, *RequestMapping* and *GetMapping* are the next common annotations with an usage of 1,809 and 1,483 code fragments. Thus, we can say that developers commonly expose the operations as *HTTP requests* when methods have either *RequestMapping* or *GetMapping*. Finally,



(a) Annotations Usage

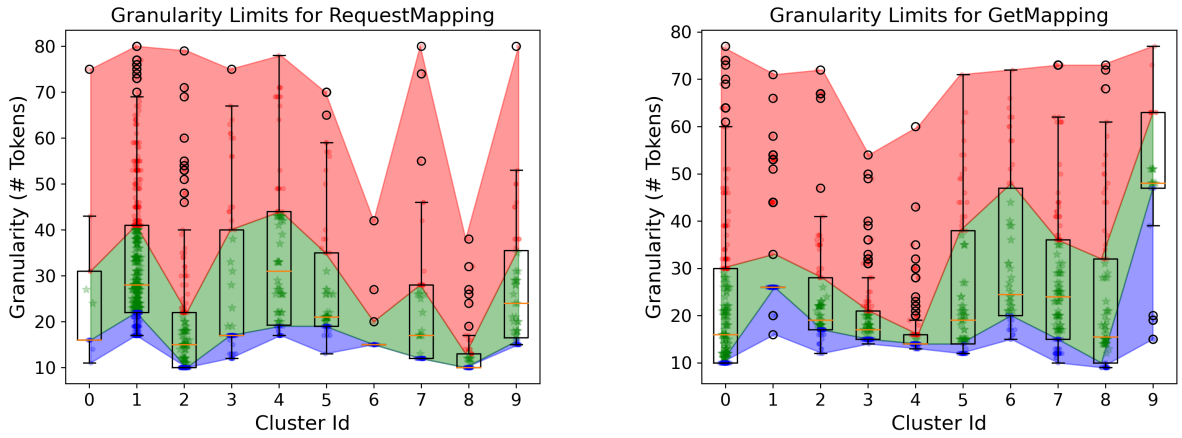
(b) Granularity Density

Figure 5.4: Mining Operations

Path is another annotation to expose operations with 418 code fragments that appears when a class has the *RequestMapping* annotation.

Figure 5.4b represents the probability density of different granularity values. We observe that three annotations, *RequestMapping*, *GetMapping* and *GET* have a similar density. They show that developers prefer operations with less than 75 tokens on average After 100 tokens, the number of operations almost disappears. There are different shapes for the Path and POST annotations. Operation Granularity tends to decrease slowly between the 75 and 150 tokens in both cases. Although POST looks to disappear after 200 tokens, it slightly reappears near the 400 tokens. Thus, clusters of the operations with similar behaviour may have their own granularity density.

We reduce the scope of our experimentation by focusing on the operations with a higher percentage of token usage. We count the number of operations for every ten tokens and calculate the usage as a percentage against the total number of operations per annotation. Figure 5.1a shows the percentage of token usage for the top annotations. A decay appears in all cases, meaning that operations with a granularity above 80 tokens have a small usage.



(a) RequestMapping C10

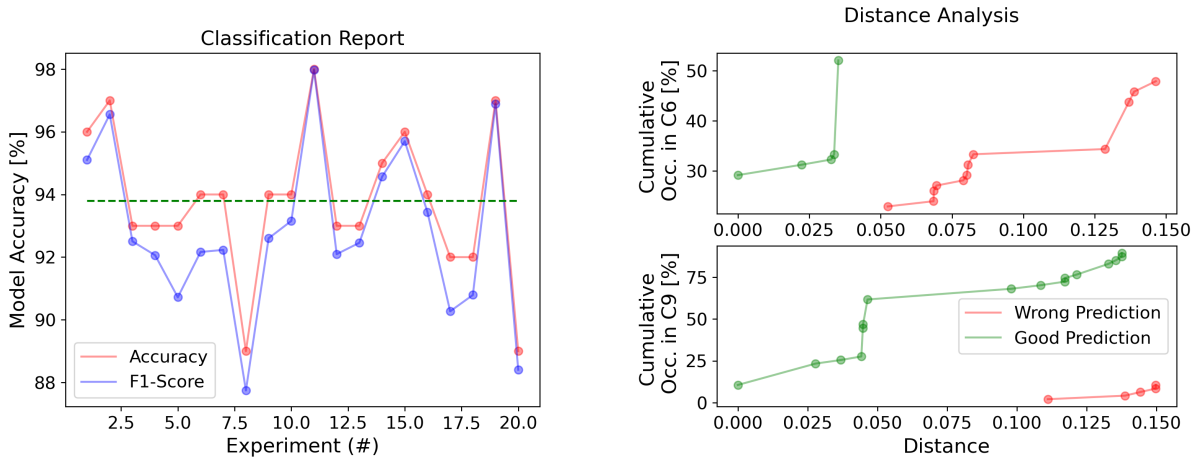
(b) GetMapping C10

Figure 5.5: Granularity Limits

Thus, we focus on operations with less than 85 tokens for the rest of the experimentation. *RequestMapping* and *GetMapping* have similar behaviour; they increase until 20 tokens and then decrease. On the other hand, *GET* and *POST* start with a high percentage of usage and then decay. *POST* has ups and downs, probably due to operations with too different behaviour. We exclude Path because its shape in Figure 5.4b differs from others.

We identify the granularity limits by clustering the vector representation of operations per annotation. Figure 5.1b shows the error curves for RequestMapping and GetMapping after clustering the operations between 3 and 100 clusters. Clustering operations have errors under 4% with a significant decay before 20 clusters. Then, we consider 10 clusters as a base to continue the exploration. Figures 5.5a and 5.5b show the granularity limits of both RequestMapping and GetMapping, respectively. The whisker boxes present the lower and high granularity values for each cluster, and each operation is a coloured data point. Thus, the blue data points (below 25%) have low, while the red data points (above 75%) have high granularity. The green data points contain 50% of data points, including the mean.

Our approach achieves an accuracy above 89%, with an average of 94%. Figure



(a) Overall Accuracy

(b) Distance Analysis

Figure 5.6: Overall Results

5.6a shows that our approach has 55% of experiments above the average. We also notice a slight difference between the Accuracy and F1-Score. The difference of 2% means low false positives and negatives for Precision and Recall. Additionally, we group the results of good and wrong predictions by cluster. For instance, Figure 5.6b shows the cumulative percentage of distance occurrence for two different clusters. We observe two possible scenarios: C6 is a cluster with a similar cumulative percentage but a gap between the distance of good and wrong predictions (0.04 to 0.05). On the other hand, C9 has a more significant cumulative percentage for good predictions. However, there is an overlapping with wrong predictions after a distance of 0.11.

5.3.5 Threats to Validity

We collected code fragments from open-source repositories and built a database of Java to analyse their granularity. Then, we selected queries randomly and evaluated the suggestion of linked annotations of our approach. Thus, we present the threats to the validity of our evaluation according to the guidelines in [130].

External validity: A limited number of repositories may reduce the successful per-

centage of operations with good granularity. We mitigate it by splitting the source code into operations to increase the number of records for exploring their granularity.

Internal validity: A subset of operations that share similar features may introduce noise and reduce the probability of finding correct granularity limits. We mitigate it by (i) reducing the size of the subset and (ii) cloning real-world operations. Using a single split for training, validation, and test datasets risks performing well on training data but failing to generalise the new data. This single split may provide an unreliable evaluation of model performance. To mitigate this, we create the validation set randomly and build a pool of examples accessed in batches. This mechanism ensures that the model is validated on diverse data samples, thereby enhancing its generalisation ability and reliability.

Construct validity: We mitigated the bias in the dataset of repositories by selecting projects related to microservices with more than 300 stars. We also mitigated the bias in Java operations created manually by introducing small changes without changing the simple behaviour of operations.

5.4 Related Work

This chapter focuses on the mining granularity of annotated operations in the area of microservices. Table 5.2 shows how our work fits the state-of-the-art techniques by comparing key features between our work and the closely related work.

Annotations detection. A few previous works consider the *annotations detection* and *microservices detection* in the context of static analysis. Our research shows one empirical study catalogues developer issues and identifies annotations as the top category for components settings (30.3%) [96]. One systematic mapping study consolidates activities for detection and transition to microservices [48]. The research also shows previous studies that

Table 5.2: Related Work

Features	**Related Work**									**Our Work**
	[96]	[94]	[48]	[93]	[41]	[28]	[105]	[123]	[122]	
Annotations detection	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓
Microservice detection	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓
Granularity importance	✗	✗	✓	✗	✓	✓	✓	✗	✓	✓
Granularity metrics	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Source code of samples	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓
Semantics similarity	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Operations collection	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Granularity exploration	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

take advantage of the annotations to detect bad smells on microservice architecture [93] and evaluate the quality of the software through mutation testing [94]. Unlike those works, our approach identifies microservices and extracts annotations from a dataset of microservice implementations from real open-source projects. We catalogue the annotations used between connected operations.

Granularity importance. Several studies mentioned the *granularity importance* of determining the size of microservices and proposed methodologies to identify suitable sizes for microservices when splitting monolith applications [48, 28]. Refactoring and Domain-Driven Design are used to find the optimal modularity of microservices [41, 105, 122]. Five works propose fourteen *granularity samples*, and only 22% have third-parties *source code* to calculate the *granularity metrics*. Unlike those works, our approach explores the different operation lengths among their semantics similarity to propose limits for good granularity.

Previous approaches do not consider the granularity by annotations. Then, we focus on the *operations collection*, *semantics similarity* and *granularity exploration* of size limits for fine-grained and coarse-grained operations. Although, other works show good results with a limited number of projects. Vural et al. [123] suggest evaluating approaches with

more projects. Thus, we collect a suitable amount of operations to reduce the threats to validity and increase the feasibility of our approach.

5.5 Summary

We elaborated a semantics-driven learning approach to suggest the granularity limits according to the similarities between operations. Our approach implements a Recurrent Neural Network (RNN) to learn the semantic relation between operations and their annotations. Then, a K-Nearest-Neighbour (KNN) classifier predicts the granularity limits. We are the first to propose a mechanism that detects the granularity limits of similar operations with their annotations.

We conclude that a database of operations is good enough to identify the granularity limits, specifically for unseen operations published with annotations. Moreover, the analysis of overall results shows that increasing the unique annotations with overlapped operations would slightly reduce the overall accuracy. Increasing the distance between operations would minimize the impact on the overall accuracy.

In our ongoing research, we are including other granularity metrics and introducing the analysis of multiple dimensions like complexity. Additionally, we are incorporating other clustering mechanisms, such as Hierarchical Clustering, for advanced features. Finally, although our approach does not improve granularity, practitioners may opt to use it to improve microservice granularity in the future.

Chapter Six

Reflection and Appraisal

6.1 Overview

This chapter aims to (i) revisit the research questions posed in Chapter 1 by reviewing how we addressed the research questions and (ii) reflect on the approach and evaluation considering different design aspects presented in previous chapters. The reflection encompasses the annotated operations, evaluation mechanism, database integrity and overhead presented in the approach.

The remainder of this chapter is organised as follows. Section 6.2 elaborates on how the proposed studies and approaches answer the research questions. Section 6.3 highlights the reflection on the different design aspects. The chapter concludes in Section 6.4.

6.2 How the Research Questions are Addressed

This section discusses how the previous chapters have addressed each research question mentioned in the first chapter of this thesis.

RQ1.1 and RQ1.2: What are the purposes of using annotations in microservice construction? How do the static analysis-based techniques support the purpose of using annotations?

In Chapter 2, we delved into the relation between annotations and microservice construction [99]. We recognised that construction of microservices poses unique concerns, requiring practical tools and techniques to address them using annotations. A systematic literature review presented the different purposes and specific uses of annotations in microservice construction. Through this review, a wide range of purposes for using annotations emerged. The found purposes were defect prediction, architecture evaluation, vulnerability detection, microservice identification, among others.

From the perspective of specific use of annotations, we identified the following: (i) identifying annotations; (ii) adding annotations; (iii) modifying annotations; (iv) verifying annotations; and (v) predicting annotations. Additionally, the study presented their relation with the different static analysis-based techniques categorised as: (i) graph theory; (ii) genetic algorithm; (iii) machine learning; (iv) deep learning; (v) syntax-based method; (vi) rule-based method; and (vii) tools. The combination of these characteristics showed us the specific uses of annotations that require more attention from industry and academia.

The results presented in this chapter guided us to identify pending challenges for research as follows: (i) consider the usage of annotations for energy consumption in microservices; (ii) discover microservice structures changing over time; (iii) incorporate deep reinforcement learning for the evolution of microservices; (iv) verify annotations for security constraints; (v) identify technical concerns from the perspective of microservice software development; (vi) incorporate a learning technique to detect missing annotations; and (vii) incorporate a semantic learning technique to improve microservice granularity. From these pending challenges, we decided to address the last three challenges.

RQ1.3: To what extent the use of annotations is one of these concerns?

In Chapter 3, we investigated microservice concerns from the perspective of developers by collecting posts from Stack Overflow [96]. Our exploration into the posts considered that developers usually publish their concerns to find a solution from online communities. Thus, this study revealed the posts with suggested answers and coding practices. The research extracted the bug symptoms and root causes and provided a classification of posts based on microservice life cycle activities at runtime, namely service routing, service discovery, service authentication & authorization, and service invocation.

From the perspective of bug symptoms, we identified the following top activity tasks: (i) client discover for service routing; (ii) configuration for service discovery; (iii) authentication for service authentication & authorization; and (iv) asynchronous for service invocation. Additionally, the study analysed the issue categories and found the following top categories: (i) wrong configuration of frameworks; (ii) wrong usage of parameters; (iii) wrong version of dependencies; (iv) missing parameters; (v) missing operations; (vi) missing web tokens; (vii) missing patterns for distributed systems; and (vi) missing annotations. The results of our study indicated that missing parameters and operations are the most common concerns in service routing. At the same time, the misuse and wrong usage of annotations are the most common concerns in service discovery.

Based on our findings, we identified the following implications for developers: (i) understanding missing operations and fixing missing parameters to reduce bugs at the compilation stage; (ii) using static code analysis and project management tools to recognise misuses of annotations; (iii) choosing a decentralised solution with short timelife tokens to minimise security breaches; and (iv) using configuration parameters and transaction patterns to keep data consistency. Moreover, from the implication list, annotations belong to the top concern, and we decided to address the usage of static code analysis to recognise similar

operations when detecting misuses and missing annotations.

RQ2: How can we leverage semantic connections between code fragments and microservice annotations to predict annotations?

In Chapter 4, acknowledging that developers follow style guidelines to coding annotated operations, we developed a semantics-driven learning approach to investigate the standard annotations that expose operations in microservice applications [98]. We considered missing annotations scenarios and proposed a novel approach for capturing the relation between code fragments and annotations, leveraging a Recurrent Neural Network (RNN) and a K-Nearest-Neighbour (KNN) classifier. Our investigation focused on discerning whether the semantic information between operations and annotations can detect misuse and missing annotations.

Our proposed approach offered a mechanism to analyse code fragments and understand the utilisation of annotations by extracting the Abstract Syntax Tree (AST) representation of each code fragment. This AST representation facilitated the detection of semantically similar code fragments through the following components: (i) a pre-processor for transforming raw data into AST representation; (ii) a learner for capturing the semantic information using a sequence-to-sequence model that encodes vectors; (iii) a similarity finder, informed by cosine similarity, for returning a subset of annotated operations from a database; (iv) the predictor for training a KNN classifier to predict annotations using this subset; and (v) the adviser for suggesting actions that keep or add annotations.

In this chapter, we concluded that using a database of rules based on code fragments with annotations is good enough to identify missing annotations. The performance evaluation of our approach focused on overall Accuracy and the BLEU (Bilingual Evaluation Understudy) Score. Notably, the approach exhibited high accuracy in actions and annotations, with a BLEU Score that validated the quality of the subset returned by the

similarity finder. The confusion matrix analysis provided insights into wrong suggestions, revealing possible considerations to improve the database of code fragments. Overall, the results strongly support the affirmative answer to this RQ2.

RQ3: How can annotations contribute to the understanding of typical granularity degree within existing microservices?

In Chapter 5, we delved into various GitHub open-source projects to find operations with different granularity values [97]. The chapter proposed a semantics-driven learning approach to mining the granularity limits of operations with their annotations according to the developer community. Moreover, the learning process pursued to build a database of similar operations for clustering by their annotations. Thus, our investigation focused on discerning whether microservice granularity from previous projects provides a foundation for assessing the granularity of new operations.

Our approach delved into the exploration of microservice granularity, using annotations as key indicators. Microservices often employ annotations to expose operations aligned with coding style guidelines. We extended the semantics-driven learning approach using the following components: (i) operation miner for extracting operations with specific annotations and generating their AST representation; (ii) operation converter for utilising semantic relations to convert ASTs into numerical vectors and calculate their granularity metrics; (iii) limits finder for clustering an operation database and predicting where new operations belong. The approach excelled in its ability to learn and map the nuances of microservice granularity, providing a valuable tool for detecting whether a new microservice operation aligns with the usual granularity degrees observed in the developer community.

Our findings revealed that some operations combine two annotations, and their probability density of granularity demonstrated favourable patterns. The granularity limits, revealed through clustering and illustrated in box-and-whisker plots, showed detailed patterns

for the selected annotations: *RequestMapping* and *GetMapping*. These annotations formed distinct clusters with different granularity traits. The analysis of overall results showed that increasing the unique annotations with overlapped operations would reduce the overall accuracy. Additionally, increasing the distance between operations would minimise the impact on the overall accuracy. In summary, results strongly indicate a positive answer for RQ3.

6.3 Reflection on the Research

This section presents a reflection on the thesis approach and evaluation. The reflection considers the design aspects concerning the operations, annotations, natural language processing, integrity of the dataset and overhead.

6.3.1 Selection of Operations and Annotations

In this thesis, selecting operations and annotations was crucial to the research journey and experiments. This process serves as a pivotal component of the overall research, contributing to producing a dataset for repeatable and reliable experiments. The decisions made in this regard were crucial to ensure that different features of this process align with the broader objective of the thesis. These features included class split, similarity of behaviour, keywords to create clones, operation size, and distribution. The alternative, adapting experiments to different real-world operations, would have posed a considerable time investment.

The selection process unfolded diverse methods, each influenced by several factors, including the feasibility of implementation, the relevance of the chosen method to the research objective, and its suitability within the microservice construction. The selected method combined a manual creation of annotated and automatic replication. The former serves as a

seed for the latter. Additionally, the selection of annotations benefited from this mechanism; when one annotation had a few operations, we increased the number of operations by creating new ones based on clone techniques.

The effectiveness of our chosen approach manifested in the achieved results. Moreover, the limitations inherent to the selection and creation of annotated operations were addressed and mitigated to ensure the credibility of the research findings. However, applying different mechanisms based on ground truth extraction would present another strategy to overcome the challenges associated with selecting operations and annotations.

6.3.2 Evaluation Using Natural Language Processing

We applied Natural Language Processing (NLP) using a semantics-driven learning approach to pursue the thesis objective. The primary focus of this approach was twofold: (i) detecting missing annotations and (ii) determining the granularity limits of annotated operations. The reliability and effectiveness of these techniques were instrumental in shaping the quality and applicability of the approach within real-world scenarios.

In this thesis, we evaluated two dimensions to demonstrate the accomplishment of our semantics-driven learning approach. One dimension is the assessment of missing annotations, a key concern in microservice software development. Simultaneously, we delved into determining the granularity limits, a critical dimension in optimising the performance of microservice-based applications. These evaluations validated the effectiveness of the approach. They contributed to fortifying the applicability of NLP in the realm of microservice construction.

The calibration of various parameters was indispensable to conduct these evaluations. The key parameters were vector dimensions, subset size and the number of k-neighbours.

These parameters influenced the credibility and efficacy of our findings and results. We would choose a sequential parameter tuning that allows a trade-off between efficiency and exhaustive exploration; however, we chose a comprehensive grid search due to its parallel execution. Additionally, a fitness function would return more efficient values; however, this avenue requires further investigation to assess its feasibility and impact on the overall evaluation process. Thus, we opted to evaluate an expected range of values.

A central characteristic of our evaluation process was utilising a simulation environment. We would evaluate the scope of the approach in a high-performance computing environment. However, we chose it only for the training of the learning model. The experiments run in a simulation environment similar to a standard laptop available for developers. Additionally, this laptop environment served as a controlled and repeatable mechanism, enabling the execution of multiple experiments. The benefits of using this environment were ensuring reproducible findings and saving resources.

6.3.3 Integrity in the Operation Database

Our semantic-driven learning approach relies an operation database containing many annotated operations. The matching process can generally start using a code fragment as an input without annotations. The goal of the approach was straightforward: identify code fragments with similar behaviour from the database. In our experiments, annotated operations were extracted from GitHub repositories to guarantee the applicability of semantics-driven learning in real-world scenarios. However, the approach lacks a mechanism to validate the correctness of operations. For instance, we would classify which operations include a bug.

A central characteristic of our operation database was the collection of different code fragments that contained patterns, rules, and style guidelines provided by developers. Bug

propagation occurred when specific bug patterns were copied along the projects, and other developers followed the same bug patterns without acknowledging which part of the code had a bug. We would clean the initial code fragments of bugs before doing the automatic replication. However, that process would be time-consuming and require applying techniques out of the scope of our approach.

Considering the accuracy of our results, this limitation would be independent of the effectiveness of the approach. However, instead of using free bug operations, measuring the number of bugs in the subset for the classifier training would help analyse the adverse effects of bugs inside operations. This bug-detection mechanism would provide insight into how extended approaches would improve their accuracy. Moreover, further exploration of this limitation would enhance the robustness of the approach.

6.3.4 Overhead

In the pursuit of the research objective, we delved into the intricate domain of semantic-driven learning applied to microservices. A crucial aspect that merited careful consideration was the presence of software overhead in the experiments. The experiments presented overhead due to the application of natural language processing techniques with machine learning classifiers. The overhead observed in the experiments can come from (i) building the database based on annotated operations and (ii) searching the subset to train the classifier.

In the first source of overhead, each annotated operation carries critical semantic information, contributing to the overall effectiveness of the applied natural language processing techniques. Thus, curing the operations is time-consuming. Additionally, searching and selecting the subset requires substantial computational resources and time in the second source of overhead. In both cases, the implications of overhead in the experiments are profound.

Overhead may lead to challenges related to the efficiency and accuracy of the approach. Excessive overhead can hinder the execution time of the experiments.

To mitigate and optimise the identified overhead sources, we used the K neighbours parameter to enhance the efficiency of building the database and searching the subset. By addressing the overhead, we aim to minimise its impact on the experiments, thereby ensuring the quality of the thesis results. However, other valuable mechanisms that we could apply to optimise the overhead would be the usage of algorithms based on hash code and running in parallel to reduce the execution time.

6.4 Concluding Remark

In this chapter, we summarise the outcome of our research questions. We also reflect retrospectively on the research limitations. Moreover, the reflection focused on what we would have done differently to make a better approach. Specifically, we reflected on (i) the process of selecting operations and annotations, (ii) the way we evaluate the usage of natural language processing in the approach, (iii) the possible impact of classifying operation database with bugs, and (iv) the sources of overhead in the solution. While there is room for improvement in our work, this reflective analysis shows an adequate addressing of research questions during the elaboration of the thesis.

Chapter Seven

Conclusion and Future Work

7.1 Overview

In the context of microservice architecture research, this journey depicts various dimensions, challenges and solutions associated with annotations in microservice construction. Chapter 2 provides the conceptual framework for annotations and highlights their purposes and specific uses in academia. Chapter 3 reveals the importance of annotations in the industry with implications for developers using static code analysis for annotation recognition. Moreover, Chapters 4 and 5 reveal something new about how new approaches could detect missed annotations and cluster operations with similar behaviour by using natural language processing. This thesis has addressed open concerns in microservices by evaluating the proposed contributions to solve the research questions stated in Chapter 1.

The remainder of this chapter is organised as follows. Section 7.2 elaborates on the contributions and implications of our work. Section 7.3 highlights the possible future research directions. The thesis concludes by the closing remarks in Section 7.4.

7.2 Contributions of the Thesis

This thesis promotes a semantic-driven learning approach for microservice construction. The approach provides a mechanism for detecting missing annotations and exploring of granularity limits by addressing a critical concern in microservice development. Specifically, this thesis adds to the field the following contributions:

- A Systematic Literature Review (SLR) of annotations in microservices [99]. We performed the SLR of annotations in microservice construction to build a catalogue of purposes of using annotations through static analysis techniques based on the existing state-of-the-art approach. Our view presents a nuanced understanding of the diverse purposes of annotations and the challenges developers face. Additionally, we outlined future directions to guide the development of advanced tools and techniques to predict annotations.

- An empirical study on microservice software development [96]. Based on SLR findings, an empirical study was conducted to identify the relevance of annotations in the microservice life cycle. Collecting posts from Stack Overflow and extracting information related to bugs, symptoms and causes of concerns help classify the issues into common concerns in microservice software development. Additionally, we recognise missing and misused annotations as part of common concerns that mainly affect practitioners and developers.

- A semantics-driven learning for microservice annotations [98]. We developed a semantic-driven learning approach for microservice construction that gets knowledge based on the semantic connections between code fragments and annotations. The learning approach builds on the natural language processing for clone detection and machine learning classifier to analyse code fragments and predict missing annotations. The researchers can benefit from the results of this contribution by enhancing the accuracy and efficiency of annotation usage. Overall, this contribution advances the field by introducing a novel approach that allows developers to use annotations more effectively to reduce complexity of microservice applications.

- A mining approach to limit granularity of annotated operations [97]. We extended a semantic learning approach to mine the granularity of operations. The learning approach aimed at identifying the granularity limits of operations by clustering them according to their behaviour. The approach explores the semantic similarity and proposes limits for fine-grained and coarse-grained operations. The approach relies on the premise that operations with similar behaviour and different lengths provide boundaries for new operations. Practitioners and researchers can benefit from these results by making informed decisions about granularity. This contribution advances the field by introducing an innovative approach to improve practices in microservice development.

7.3 Future Work

In addition to the future work in Section 2.4.1, which presents the importance of specific uses of annotations, this section further discusses the areas for potential future research direction. Specifically, it explores the possibilities offered by annotations and semantics-driven learning.

7.3.1 Enhanced Annotations Semantics.

Chapter 2 provides promising avenue for future research and delve deeper into semantics of annotations in microservices by addressing the limitations of current approach. One mechanism for enhancement is the incorporation of Deep Reinforcement Learning (DRL) to benefit from rewards to predict possible structure changes. Deep learning algorithms have shown remarkable capabilities in learning patterns and making predictions [38]. Reinforcement learning is a type of learning in which an agent interacts with an environment to achieve a specific goal by taking actions that give a reward [43]. In microservice evolution, the environment represents the architecture, the actions are the possible changes of structure, and the reward involves quality attributes of a system. DRL trains the agent to predict the most effective actions based on the changes in the structures of open-source projects.

7.3.2 Annotation Impact on Non-Functional Aspects.

Chapter 3 provides valuable insights into the development concerns and impact of annotations on microservices. Additional, Chapter 4 provides an approach that use semantics of operations to identify missing annotations. These concerns are primarily focused on functional aspects of software development. Future research could incorporate non-functional aspects, such as security and performance [142, 22]. This non-functional aspects means additional features that needs for exploration of other mechanisms for vector representations such as Flow2vec for these advanced features. Flow2vec considers the control flow as part of the vector encoding [115]. Thus, the approach could identify portions of the control flow related to performance and security, and incorporate them for getting vector representation.

7.3.3 Dynamic Granularity Adjustment.

Chapter 5 provides a new perspective on granularity in microservices, which relies on static analysis of operations. Future research could extend this analysis to the dynamic nature of microservices, considering the runtime changes in microservices [108]. These runtime changes are related to operational patterns over time. Then, the approach needs to learn the recognition of dynamic patterns provided by runtime monitors. These runtime monitors generally capture information in real-time and provide log records for posterior manual analysis [147, 141]. The analysis results could serve as a training dataset for pattern recognition in a machine learning algorithm. Additionally, researchers could evaluate different clustering mechanisms, such as Hierarchical Clustering, for detecting granularity limits and matching them with the performance limitations to launch a warning for a granularity adjustment.

7.4 Closing Remarks

This section summarises the outcome of the research conducted throughout the elaboration of this thesis. The primary conclusions extracted from this research are:

- This thesis demonstrated that annotations can significantly impact microservices and specific uses of annotations benefits from static analysis techniques, emphasizing the potential for enhancing the overall microservice construction through advanced analysis of annotation usage.

- Through empirical analysis and computational modelling, this thesis has established that annotations play a crucial role in shaping microservices. It provides valuable insights for researchers and practitioners seeking effective solutions for microservices concerns. Additionally, this thesis has contributed learning models to predict annotations and determine typical granularity values.

- Chapter 3, after its empirical analysis of Stack Overflow posts, identified and categorised the primary concerns related to annotations in microservices development. Misuse of annotations emerge as a top concern, highlighting the significance of carefully selecting and applying annotations to ensure the intended functionality and quality of microservices architecture.

- Chapter 4 contributed a novel approach for detecting missing annotations in microservices based on the semantic analysis of annotations. By leveraging neural language processing techniques, this research offered a more nuanced understanding of how annotations correlate with fault-prone areas in code.

- Chapter 5 further enriched the landscape by presenting a semantics-driven learning approach for exploring the typical granularity values of microservices operations. By clustering similar operations based on annotations, the research introduced a method for refining operations design by determining their granularity limits, which affects their performance and maintainability.

References

[1] [n.d.] *Query Stack Overflow*. 2020. URL: https://data.stackexchange.com/stackoverflow/queries.

[2] [n.d]. *Common components used in Microservices*. 2020. URL: https://bit.ly/3o2uHEJ.

[3] Carlos M. Aderaldo and et al. "Benchmark Requirements for Microservices Architecture Research". In: *Proceedings of the 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. Buenos Aires, Argentina: IEEE, 2017, pp. 8–13.

[4] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Computing Surveys* 51.4 (July 2018), p. 81.

[5] Eman Abdullah AlOmar et al. "On preserving the behavior in software refactoring: A systematic mapping study". In: *Information and Software Technology* 140 (Dec. 2021), p. 106675.

[6] Eman Abdullah Alomar et al. "Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox". In: *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, May 2021, pp. 348–357.

[7] Bader Alshemaimri et al. "A survey of problematic database code fragments in software systems". In: *Engineering Reports* 3 (10 Oct. 2021), e12441.

[8] Nuha Alshuqayran and et al. “A Systematic Mapping Study in Microservice Architecture”. In: *Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. Macau, China: IEEE, 2016.

[9] Nuha Alshuqayran, Nour Ali, and Roger Evans. “Towards Micro Service Architecture Recovery: An Empirical Study”. In: *Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA)*. Institute of Electrical and Electronics Engineers Inc., July 2018, pp. 47–56.

[10] Daniel R.F. Apolinário and Breno B.N. de França. “A method for monitoring the coupling evolution of microservice-based architectures”. In: *Journal of the Brazilian Computer Society* 27 (1 Dec. 2021).

[11] Thiwankan C Kapugama Arachchi and K P Hewagamage. “Process of Conversion Monolithic Application to Microservices Based Architecture”. Master. University of Colombo, 2021.

[12] Elena A. Araujo et al. “Applying a multi-platform architectural conformance solution in a real-world microservice-based system”. In: *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse*. Association for Computing Machinery, Oct. 2020, pp. 41–50.

[13] Muhammad Ilyas Azeem et al. “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis”. In: *Information and Software Technology* 108 (Apr. 2019), pp. 115–138.

[14] Alexander Bakhtin et al. “Survey on tools and techniques detecting microservice api patterns”. In: *Proceedings of the 2022 IEEE International Conference on Services Computing (SCC)*. IEEE. Barcelona, Spain: IEEE, 2022, pp. 31–38.

[15] Alan Bandeira et al. “We need to talk about microservices: an analysis from the discussions on stackoverflow”. In: *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, p. 5.

[16] Marcello M. Bersani et al. “Verifying big data topologies by-design: a semi-automated approach”. In: *Journal of Big Data* 6 (1 Dec. 2019), pp. 1–23.

[17] Farzana Ahamed Bhuiyan, Md Bulbul Sharif, and Akond Rahman. “Security bug report usage for software vulnerability research: a systematic mapping study”. In: *IEEE Access* 9 (2021), pp. 28471–28495.

[18] Ekaba Bisong. “Introduction to Scikit-learn”. In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 215–229.

[19] Raffaele Bolla et al. “Enhancing energy-efficient cloud management through code annotations and the green abstraction layer”. In: *Proceedings of the 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2015, pp. 534–539.

[20] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3*. IEEE Computer Society, 2014.

[21] Miguel Brito, Jácome Cunha, and João Saraiva. “Identification of microservices from monolithic applications through topic modelling”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. Association for Computing Machinery, Mar. 2021, pp. 1409–1418.

[22] Tomáš Bureš et al. “Towards performance-aware engineering of autonomic component ensembles”. In: *Proceedings of the Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I 6*. Springer. Oct. 2014, pp. 131–146.

[23] Vincent Bushong, Dipta Das, and Tomas Cerny. "Reconstructing the holistic architecture of microservice systems using static analysis". In: *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER)*. Online Streaming: Scitepress, May 2022, pp. 149–157.

[24] John Carnell and Illary Huaylupo Sánchez. *Spring microservices in action*. Simon and Schuster, 2021.

[25] Tomas Cerny et al. "Microvision: Static analysis-based approach to visualizing microservices in augmented reality". In: *Proceedings of the 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2022, pp. 49–58.

[26] Istehad Chowdhury and Mohammad Zulkernine. "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities". In: *Journal of Systems Architecture* 57 (3 Mar. 2011), pp. 294–313.

[27] Maria Christakis and Christian Bird. "What developers want and need from program analysis: an empirical study". In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. Association for Computing Machinery (ACM), Aug. 2016, pp. 332–343.

[28] Michel Cojocaru, Alexandru Uta, and Ana Maria Oprescu. "MicroValid: A validation framework for automatically decomposed microservices". In: *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2019, pp. 78–86.

[29] Antonio Cuomo, Antonella Santone, and Umberto Villano. "CD-Form: A clone detector based on formal methods". In: *Science of Computer Programming* 95 (2014), pp. 390–405.

[30] Hoa Khanh Dam et al. "Lessons learned from using a deep tree-based model for software defect prediction in practice". In: *Proceedings of the 2019 IEEE/ACM 16th*

International Conference on Mining Software Repositories (MSR). Montreal Quebec, Canada: IEEE Press, May 2019, pp. 46–57.

[31] Mohamed Daoud et al. "A multi-model based microservices identification approach". In: *Journal of Systems Architecture* 118 (Sept. 2021).

[32] Dipta Das et al. "On automated RBAC assessment by constructing a centralized perspective for microservice mesh". In: *PeerJ Computer Science* 7 (2021), pp. 1–24.

[33] Jose M Del Alamo et al. "A systematic mapping study on automated analysis of privacy policies". In: *Computing* 104.9 (2022), pp. 2053–2076.

[34] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. "Research on architecting microservices: Trends, focus, and potential for industrial adoption". In: *Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 21–30.

[35] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. "Using annotations to check structural properties of classes". In: *Proceedings of the Fundamental Approaches to Software Engineering: 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 8*. Springer. 2005, pp. 237–252.

[36] Daniel Escobar et al. "Towards the understanding and evolution of monolithic applications as microservices". In: *Proceedings of the 2016 XLII Latin American computing conference (CLEI)*. Institute of Electrical and Electronics Engineers Inc., Jan. 2017.

[37] Christian Esposito and et al. "Challenges in delivering software in the cloud as microservices". In: *IEEE Cloud Computing* (2016), pp. 10–14.

[38] Runhan Feng et al. "Automated Detection of Password Leakage from Public GitHub Repositories". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. Pittsburgh Pennsylvania: ACM, 2022.

[39] Pietro Ferrara et al. “Static analysis for discovering IoT vulnerabilities”. In: *International Journal on Software Tools for Technology Transfer* 23 (1 Feb. 2021), pp. 71–88.

[40] Francisco Freitas, André Ferreira, and Jácome Cunha. “Refactoring java monoliths into executable microservice-based applications”. In: *Proceedings of the 25th Brazilian Symposium on Programming Languages*. Association for Computing Machinery, Sept. 2021, pp. 100–107.

[41] Jonas Fritzsch et al. “From monolith to microservices: A classification of refactoring approaches”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11350 LNCS (2019), pp. 128–141.

[42] Yu Gan and et al. “Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices”. In: *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019.

[43] Samira Ghodratnama, Amin Behehsti, and Mehrdad Zakershahrak. “A personalized reinforcement learning summarization service for learning structure from unstructured data”. In: *Proceedings of the 2023 IEEE International Conference on Web Services (ICWS)*. 2023, pp. 206–213.

[44] Qiwen Gu, Stefan Wagner, and Jonas Fritzsch. “A meta-approach to guide architectural refactoring from monolithic applications to microservices”. Master. Universität Stuttgart, 2020.

[45] Wang Guan, Ivan Smetannikov, and Man Tianxing. “Survey on automatic text summarization and transformer models applicability”. In: *Proceedings of the 2020 1st International Conference on Control, Robotics and Intelligent System*. Xiamen, China: ACM, 2020, pp. 176–184.

[46] Haryadi S. Gunawi et al. "What bugs live in the cloud? A study of 3000+ issues in cloud systems". In: *Proceedings of the 5th ACM Symposium on Cloud Computing (SOCC)*. Association for Computing Machinery, Nov. 2014.

[47] Arun Gupta. *Microservice Design Patterns*. 2015. URL: https://goo.gl/pd5d1x.

[48] Sara Hassan, Rami Bahsoon, and Rick Kazman. "Microservice transition and its granularity problem: A systematic mapping study". In: *Software: Practice and Experience* 50.9 (2020), pp. 1651–1681.

[49] Victor Heorhiadi and et al. "Gremlin: Systematic resilience testing of microservices". In: *Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. Nara, Japan: IEEE, 2016.

[50] Wolfgang Hobmaier. *Improving the Quality of OpenAPI Specifications Using TypeScript Types and Annotations*. 2020.

[51] Xing Hu et al. "Deep code comment generation". In: *Proceedings of the 26th conference on Program Comprehension (ICPC)*. ACM Press, 2018, pp. 200–210.

[52] James Ivers, Chris Seifried, and Ipek Ozkaya. "Untangling the knot: Enabling architecture evolution with search-based refactoring". In: *Proceedings of the 2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, 2022, pp. 101–111.

[53] Pooyan Jamshidi et al. "Microservices: The journey so far and challenges ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35.

[54] Jeff Atwood and Joel Spolsky. *StackOverflow*. 2008. URL: https://stackoverflow.com/.

[55] Jeff Atwood and Joel Spolsky. *StackOverflow Comment Everywhere*. 2008. URL: https://stackoverflow.com/help/privileges/comment.

[56] Kamil Jezek, Jens Dietrich, and Premek Brada. "How Java APIs break - An empirical study". In: *Information and Software Technology* 65 (Sept. 2015), pp. 129–146.

[57] Paul Jorgensen. *Software testing, a craftsman approach*. CRC Press, 2014.

[58] Nikhil Ketkar. "Introduction to pytorch". In: *Deep learning with python*. Springer, 2017, pp. 195–208.

[59] Myeongsoo Kim et al. "Automated test generation for REST APIs: no time to rest yet". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, July 2022, pp. 289–301.

[60] B. Kitchenham and P. Brereton. "A systematic review of systematic review process research in software engineering". In: 55.12 (2013), pp. 2049–2075.

[61] B. Kitchenham and S. Charters. *Guidelines for performing systematic literature reviews in software engineering*. 2007.

[62] Barbara Kitchenham. "Procedures for performing systematic reviews". In: *Keele, UK, Keele University* 33.2004 (2004), pp. 1–26.

[63] Barbara Kitchenham et al. "Trends in the quality of human-centric software engineering experiments–A quasi-experiment". In: *IEEE Transactions on Software Engineering* 39.7 (2012), pp. 1002–1017.

[64] Emna Ksontini et al. "Refactorings and technical debt in docker projects: An empirical study". In: *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2021.

[65] Rodrigo Laigner et al. "Cataloging dependency injection anti-patterns in software systems". In: *Journal of Systems and Software* 184 (Feb. 2022).

[66] Ha Thanh Le et al. "Automated reverse engineering of role-based access control policies of web applications". In: *Journal of Systems and Software* 184 (Feb. 2022).

[67] Xing Li et al. "Automatic policy generation for Inter-Service access control of microservices". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

[68] Guangtai Liang et al. “Automatic construction of an effective training set for prioritizing static analysis warnings”. In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Antwerp, Belgium: ACM, Sept. 2010, pp. 93–102.

[69] Fábio Lopes et al. “Automating orthogonal defect classification using machine learning algorithms”. In: *Future Generation Computer Systems* 102 (Jan. 2020), pp. 932–947.

[70] Pablo Loyola and Yutaka Matsuo. “Learning feature representations from change dependency graphs for defect prediction”. In: *Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 361–372.

[71] Shang Pin Ma et al. “Version-based and risk-enabled testing, monitoring, and visualization of microservice systems”. In: *Journal of Software: Evolution and Process* 34.10 (2022).

[72] Shang Pin Ma et al. “Version-Based Microservice Analysis, Monitoring, and Visualization”. In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 2019-December. IEEE Computer Society, Dec. 2019, pp. 165–172.

[73] Senthil Mani et al. “AUSUM: Approach for unsupervised bug report summarization”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE ’12)*. Cary, North Carolina: ACM Press, 2012, pp. 1–11.

[74] Kollegger Manuel and Johannes Kepler. “Continuous Architecture Evaluation in the Context of Microservices Computer Science”. PhD thesis. Universität Linz, 2018.

[75] Diego Marcilio et al. “SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings”. In: *Journal of Systems and Software* 168 (Oct. 2020).

[76] Manuel Mazzara and Bertrand Meyer, eds. *Present and ulterior software engineering*. Springer, 2017.

[77] Mostafa Mehrabi, Nasser Giacaman, and Oliver Sinnen. “@ PT: Unobtrusive parallel programming with Java annotations”. In: *Concurrency and Computation: Practice and Experience* 31.1 (2019), e4831.

[78] Andrea Melis. “Cybersecurity issues in software architectures for innovative services”. PhD thesis. AlmaDL University of Bologna Digital Library, 2020.

[79] Na Meng and et al. “Secure Coding Practices in Java: Challenges and Vulnerabilities”. In: *40th ICSE*. Gothenburg, Sweden: IEEE, 2018.

[80] Loup Meurice, Csaba Nagy, and Anthony Cleve. “Static analysis of dynamic database usage in Java systems”. In: *Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016)*. Ed. by Selmin Nurcan et al. Vol. 9694. Springer, June 2016, pp. 491–506.

[81] Leticia Montalvillo and Oscar Díaz. “Requirement-driven evolution in software product lines: A systematic mapping study”. In: *Journal of Systems and Software* 122 (Dec. 2016), pp. 110–143.

[82] Seyed Mehdi Nasehi and et al. “What makes a good code example? A study of programming Q&A in StackOverflow”. In: *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sept. 2012, pp. 25–34.

[83] Carlos Noguera and Laurence Duchien. “Annotation framework validation using domain models”. In: *Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications*. Springer Berlin Heidelberg, June 2008, pp. 48–62.

[84] Batyr Nuryyev et al. "Mining annotation usage rules: A case study with MicroProfile". In: *Proceedings of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2022, pp. 553–562.

[85] Inah Omoronyia et al. "A review of awareness in distributed collaborative software engineering". In: *Software - Practice and Experience* 40 (12 Nov. 2010), pp. 1107–1133.

[86] Oracle. *Lesson: Annotations (The Java™ Tutorials > Learning the Java Language)*. URL: https://docs.oracle.com/javase/tutorial/java/annotations/ (visited on 11/17/2020).

[87] Claus Pahl and Pooyan Jamshidi. "Microservices: A systematic mapping study". In: *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER)*. Rome, Italy: SCITEPRESS, 2016, pp. 137–146.

[88] Ya Pan et al. "A systematic literature review of android malware detection using static analysis". In: *IEEE Access* 8 (2020), pp. 116363–116379.

[89] Aurojit Panda, Mooly Sagiv, and Scott Shenker. "Verification in the age of microservices". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. Vol. 7. Whistler, BC, Canada: ACM, May 2017, pp. 30–36.

[90] Daniel Perez and Shigeru Chiba. "Cross-language clone detection by learning over abstract syntax trees". In: *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE Press, May 2019, pp. 518–528.

[91] K. Petersen and et al. "Guidelines for conducting systematic mapping studies in software engineering: An update". In: *Information and Software Technology* 64 (2015), pp. 1–18.

[92] Ilaria Pigazzini. "Automatic detection of architectural bad smells through semantic representation of code". In: *Proceedings of the 13th European Conference on Software Architecture (ECSA '19)*. ACM, Sept. 2019, pp. 59–62.

[93] Ilaria Pigazzini et al. "Towards microservice smells detection". In: *Proceedings of the 3rd International Conference on Technical Debt*. Association for Computing Machinery, June 2020, pp. 92–97.

[94] Pedro Pinheiro et al. "Mutation operators for code annotations". In: *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing (SAST 2018)*. Association for Computing Machinery, Sept. 2018, pp. 77–86.

[95] Md Rafiqul Islam Rabin et al. "Towards demystifying dimensions of source code embeddings". In: *RL+SE and PL 2020 - Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages, Co-located with ESEC/FSE 2020*. New York, NY, USA: Association for Computing Machinery, Inc, Nov. 2020, pp. 29–38.

[96] Francisco Ramírez et al. "An empirical study on microservice software development". In: *Proceedings of the 2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES)*. IEEE. 2021, pp. 16–23.

[97] Francisco Ramírez et al. "Mining the limits of granularity for microservice annotations". In: *Proceedings of the 20th International Conference on Service-Oriented Computing (ICSOC 2022)*. Springer. Nov. 2022, pp. 273–281.

[98] Francisco Ramírez et al. "Semantics-driven learning for microservice annotations". In: *Proceedings of the 20th International Conference on Service-Oriented Computing (ICSOC 2022)*. Springer. Nov. 2022, pp. 255–263.

[99] Francisco Ramírez et al. "Systematic review of annotations in microservice construction (under review for publication)". In: *ACM Computing Surveys (CSUR)* (2024).

[100] Zhongshan Ren et al. "Migrating web applications from monolithic structure to microservices architecture". In: *Proceedings of the 10th Asia-Pacific Symposium on Internetware*. Association for Computing Machinery, Sept. 2018, pp. 1–10.

[101] RFC6749. *OAuth*. 2006. URL: https://oauth.net/2/.

[102] Christoffer Rosen and et al. "What are mobile developers asking about? A large scale study using stack overflow". In: *Empirical Software Engineering* 21 (2016), pp. 1192–1223.

[103] Anees Saba. *4 Challenges you need to address with Microservices adoption*. 2016. URL: https://bit.ly/35Mfr4O.

[104] Caitlin Sadowski et al. "Tricorder: Building a program analysis ecosystem". In: *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*. Vol. 1. IEEE Computer Society, Aug. 2015, pp. 598–608.

[105] Ana Santos and Hugo Paula. "Microservice decomposition and evaluation using dependency graph and silhouette coefficient". In: *Proceedings of the 15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*. Sept. 2021, pp. 51–60.

[106] Stefanie Scherzinger and Sebastian Sidortschuck. "An empirical study on the design and evolution of NoSQL database schemas". In: *Proceedings of the 39th International Conference on Conceptual Modeling*. Vol. 12400 LNCS. Springer Science and Business Media Deutschland GmbH, Nov. 2020, pp. 441–455.

[107] Micah Schiewe et al. "Advancing static code analysis with language-agnostic component identification". In: *IEEE Access* 10 (2022), pp. 30743–30761.

[108] Khaled Sellami et al. “Combining static and dynamic analysis to decompose monolithic application into microservices”. In: *Proceedings of the Service-Oriented Computing*. Ed. by Javier Troya et al. Cham: Springer Nature Switzerland, 2022, pp. 203–218.

[109] Dharmendra Shadija and et al. “Towards an understanding of microservices”. In: *Proceedings of the 2017 23rd International Conference on Automation and Computing (ICAC)*. IEEE, 2017, pp. 1–6.

[110] Rongrong She, Liping Zhang, and Fengrong Zhao. “A method for identifying and recommending reconstructed clones”. In: *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*. Association for Computing Machinery, Jan. 2019, pp. 39–44.

[111] Zhidong Shen and Si Chen. “A survey of automatic software vulnerability detection, program repair, and defect prediction techniques”. In: *Security and Communication Networks* 2020 (Sept. 2020), pp. 1–16.

[112] Ian Sommerville. *Software Engineering (tenth edn. global edition)*. 2016.

[113] Fausto Spoto. “The Julia static analyzer for Java”. In: *Proceedings of the Static Analysis: 23rd International Symposium (SAS 2016)*. Vol. 9837 LNCS. Springer Verlag, 2016, pp. 39–57.

[114] Niels Streekmann. “Software evolution and modernisation”. In: *Clustering-Based Support for Software Architecture Restructuring*. Wiesbaden: Springer, 2011, pp. 23–44.

[115] Yulei Sui et al. “Flow2Vec: value-flow-based precise code embedding”. In: *Proceedings of the ACM on Programming Languages 4, No. OOPSLA (2020)*. Vol. 4. Nov. 2020, pp. 1–27.

[116] D. Taibi and V. Lenarduzzi. “On the definition of microservice bad smells”. In: *IEEE Software* 35 (3 2018), pp. 56–62.

[117] Yiming Tang et al. "An empirical study of refactorings and technical debt in machine learning systems". In: *Proceedings of the 2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE Computer Society, May 2021, pp. 238–250.

[118] Rafik Tighilt et al. "On the study of microservices antipatterns: A catalog proposal". In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. Association for Computing Machinery, July 2020, pp. 1–13.

[119] Shreya Tiwari et al. "A review on green computing implementation using efficient techniques". In: *Proceedings of the 2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*. IEEE. Greater Noida, India: IEEE, 2021, pp. 1496–1501.

[120] Harsh Upreti. *4 Essential strategies for testing microservices*. 2018. URL: https://smartbear.com/blog/4-essential-strategies-for-testing-microservices/.

[121] Fredy H. Vera-Rivera, Carlos Gaona, and Hernán Astudillo. "Defining and measuring microservice granularity—a literature overview". In: *PeerJ Computer Science* 7 (2021).

[122] Fredy H. Vera-Rivera et al. "Microservices Backlog - A genetic programming technique for identification and evaluation of microservices from user stories". In: *IEEE Access* 9 (2021), pp. 117178–117203.

[123] Hulya Vural and Murat Koyuncu. "Does domain-driven design lead to finding the optimal modularity of a microservice?" In: *IEEE Access* 9 (2021), pp. 32721–32733.

[124] Andrew Walker, Dipta Das, and Tomas Cerny. "Automated code-smell detection in microservices through static analysis: A case study". In: *Applied Sciences (Switzerland)* 10 (21 Nov. 2020), pp. 1–20.

[125] Andrew Walker, Dipta Das, and Tomas Cerny. "Automated microservice code-smell detection". In: *Proceedings of the Information Science and Applications (ICISA)*. Springer Singapore, 2021, pp. 211–221.

[126] Andrew Walker, Ian Laird, and Tomas Cerny. "On automatic software architecture reconstruction of microservice applications". In: *Proceedings of the Information Science and Applications (ICISA 2020)*. Springer Singapore, 2021, pp. 223–234.

[127] Jie Wang et al. "Scaling static taint analysis to industrial SOA applications: A case study at Alibaba". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, Inc, Nov. 2020, pp. 1477–1486.

[128] AWS Whitepaper. *Challenges of Microservices*. URL: https://amzn.to/3a0XRgI.

[129] Claes Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 2014, pp. 1–10.

[130] Claes Wohlin et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[131] Li Wu, Johan Tordsson, Jasmin Bogatinovski, et al. "MicroDiag: Fine-grained performance diagnosis for microservice systems". In: *Proceedings of the International Workshop on Cloud Intelligence (CloudIntelligence 2021)*. IEEE. 2021, pp. 31–36.

[132] Xia Xin et al. "Measuring program comprehension: A large-scale field study with professionals". In: *IEEE Transactions on Software Engineering* 44.10 (Oct. 2018), pp. 951–976.

[133] Baowen Xu et al. "A brief survey of program slicing". In: *ACM SIGSOFT Software Engineering Notes* 30.2 (2005), pp. 1–36.

[134] Yanming Yang et al. “A survey on deep learning for software engineering”. In: *ACM Computing Surveys* (Jan. 2021).

[135] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. “Coacor: Code annotation for code retrieval with reinforcement learning”. In: *Proceedings of the The world wide web conference (WWW 2019)*. 2019, pp. 2203–2214.

[136] Dongjin Yu et al. “A survey on security issues in services communication of Microservices-enabled fog applications”. In: *Concurrency and Computation: Practice and Experience* 31.22 (2019).

[137] Hao Yu et al. “Neural detection of semantic code clones via tree-based convolution”. In: *Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC 2019)*. Montreal, Quebec, Canada: IEEE, May 2019, pp. 70–80.

[138] Zhongxing Yu et al. “Characterizing the usage, evolution and impact of Java annotations in practice”. In: *IEEE Transactions on Software Engineering* 47.5 (2021), pp. 969–986.

[139] Pascal Zaragoza et al. “Leveraging the layered architecture for microservice recovery”. In: *Proceedings of the 2022 IEEE 19th International Conference on Software Architecture (ICSA)*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 135–145.

[140] Uwe Zdun et al. “Microservice security metrics for secure communication, identity management, and observability”. In: *ACM Transactions on Software Engineering and Methodology* (May 2022).

[141] Chenxi Zhang et al. “DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning”. In: *Proceedings of the 44th International*

Conference on Software Engineering. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 623–634.

[142] Haibo Zhang and Kouichi Sakurai. "A survey of software clone detection from security perspective". In: *IEEE Access* 9 (2021), pp. 48157–48173.

[143] Jian Zhang et al. "A novel neural source code representation based on abstract syntax tree". In: *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, Quebec, Canada: IEEE, May 2019, pp. 783–794.

[144] Man Zhang and Andrea Arcuri. "Adaptive hypermutation for search-based system test generation: A study on REST APIs with EvoMaster". In: *ACM Transactions on Software Engineering and Methodology* 31 (1 Jan. 2022), pp. 1–52.

[145] Xiang Zhou et al. "Delta debugging microservice systems". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Vol. 18. ACM, Sept. 2018, pp. 802–807.

[146] Xiang Zhou et al. "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study". In: *IEEE Transactions on Software Engineering* 14.8 (2018), pp. 1–1.

[147] Xiang Zhou et al. "Latent error prediction and fault localization for microservice applications by learning from system trace logs". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019, pp. 583–694.

[148] Yu Zhou et al. "Analyzing APIs documentation and code to detect directive defects". In: *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina: IEEE Press, 2017, pp. 27–37.

[149] Jun Zhu et al. “Mitigating access control vulnerabilities through interactive static analysis”. In: *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. Vol. 2015-June. Association for Computing Machinery, June 2015, pp. 199–209.

[150] Meital Zilberstein and Eran Yahav. “Leveraging a corpus of natural language descriptions for program similarity”. In: *Onward! 2016: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Amsterdam, Netherlands: ACM, Nov. 2016, pp. 197–211.