FROM CORE TO APEX: TOWARDS A MULTI-FACET OPTIMIZATION OF SOFTWARE SYSTEMS

by

ANWAR GHAMMAM

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND INFORMATICS

2025

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Amine Barrak, Ph.D., Chair
Marouane Kessentini, Ph.D.
Sunny Raj, Ph.D.
Mehdi Bagherzadeh, Ph.D.
Sam Srauy, Ph.D.
Hua Ming, Ph.D.

As a token of respect, gratitude, and recognition, I dedicate this work:

To God, the most merciful.

To my father, my ideal, who saw a star in me before I ever believed it myself. Your faith in me and encouragement continue to guide me, even though you are no longer here. Your light shines within me every day, and this accomplishment is as much yours as it is mine.

O Allah, envelop him in Your boundless mercy, and make this blessed accomplishment an everlasting charity that illuminates his grave, elevates his rank, and adds weight to his scale on the Day he meets You.

To my mother, sister, and brother, whose unwavering love and support have been the cornerstone of my journey.

To my forever partner, whose unwavering support carried me through every challenge of my PhD. None of this would have been possible without you by my side. I love you more than the total word count of my thesis.

To my professors, whose wisdom and guidance have shaped me into the scholar I am today. Your mentorship has been a beacon of knowledge and possibility, and I am forever grateful for your investment in my growth.

Thank you all for the love, support, and belief I have been blessed to receive!

ACKNOWLEDGMENTS

At the end of this work, I would like to express my deep gratitude to all those who supported me and helped me accomplish it in the best conditions.

My profound gratitude goes first to my supervisor, Dr. Marouane Kessentini and my committee chair, Dr. Amine Barrak, who expertly guided me through this graduation project. Their guidance, persistent help, insightful feedback, meticulous scrutiny, and scientific approaches have helped me greatly to accomplish this work.

I want to thank the honorable committee members for agreeing to examine this report and for taking the time to analyze and attend the academic defense of this little effort. I'm hopeful they'll discover the clarity and purposefulness they're looking for.

My appreciation also extends to my lab members, post-docs, and collaborators for their insightful discussions, constructive feedback, and unwavering support throughout this journey.

Finally, I humbly thank all concerned parties who cooperated with me in this work.

Anwar Ghammam

ABSTRACT

FROM CORE TO APEX: TOWARDS A MULTI-FACET OPTIMIZATION OF SOFTWARE SYSTEMS

by

Anwar Ghammam

Adviser: Amine Barrak, Ph.D.

As software evolves and deviates from its original design, it becomes prone to increased complexity, decreased performance, and higher maintenance costs. Software optimization has thus become a critical research area in software engineering, driven by modern systems' growing complexity and scale[1, 2, 3].

Refactoring—the process of restructuring code without altering its external behavior—has been extensively studied to improve software performance and maintainability. Consequently, refactoring recommendation tools have emerged to assist developers with the process of refactoring, but these tools often fail to align with industry practical needs, resulting in a gap between theoretical recommendations by researchers and practitioners' needs in real-world applications.

Beyond source code, software optimization spans broader lifecycle processes, including software resource management and build systems. Our research addresses these challenges holistically, advancing software optimization from core resource allocation to apex-level development practices.

First, this dissertation presents a multi-objective approach to software management, focusing on resource-aware containerization. By optimizing workload balancing in cloud environments and extending these techniques to edge environments such as Software-Defined Vehicles (SDVs), this research ensures efficient resource utilization and

scalability. Using NSGA-III, our approach demonstrates superior performance in managing constraints and optimizing conflicting objectives, with practical applications validated through industrial partnership.

Second, our research proposes significant contribution to source code refactoring by addressing gaps between industry practices and existing refactoring recommendation tools. Current tools often overlook key criteria crucial to developers' decisions. By addressing the gap, we provide a broader perspective for refactoring tool developers, enabling them to enhance their tools and make them more efficient and practical for industry practitioners, thereby enhancing code maintainability and developer productivity.

Finally, this research delves into the underexplored domain of build systems refactoring. Our analysis was conducted on 725 examined build-file-related commits, where we identified 24 build-related refactorings, divided into 6 main categories. We also investigate whether these refactorings are used to tackle technical debts in build systems. Building on these insights, we introduce BuildRefMiner to detect and analyze refactorings within Buildfiles using LLMs.

This work redefines software optimization as an integrated process, combining refactoring insights, resource-aware strategies, and lifecycle improvements to ensure robust, efficient, and sustainable software systems.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

ABSTRACT v

LIST OF FIGURES xi

LIST OF TABLES xiv

CHAPTER ONE
INTRODUCTION 1

1.1. Research Context 1

1.2. Problem Statement 2

1.3. Proposed Research 5

1.4. Research Contributions 7

1.5. Publications List 10

1.6. Organization of the Dissertation 11

CHAPTER TWO
BACKGROUND 12

2.1. Docker and Container-based Projects 12

2.2. Multi-Objective Optimization Algorithms 14

2.2.1. NSGA-III: A Reference-Point-Based MOO Algorithm 15

2.3. Refactoring & Technical Debt 16

2.4. Build Systems 17

CHAPTER THREE
STATE OF THE ART 19

3.1. Software Allocation Management 19

TABLE OF CONTENTS—Continued

3.2. Software Source Code Refactoring 21

3.3. Build Code Maintenance 23

CHAPTER FOUR
DYNAMIC SOFTWARE CONTAINERS WORKLOAD BALANCING
VIA MANY-OBJECTIVE SEARCH 25

4.1. ClusterWare: Many-Objective Scheduling
Approach for Software Containers 28

4.1.1. Approach Overview 28

4.1.2. Population Representation 31

4.1.3. Objective Functions 32

4.2. Empirical Study 36

4.2.1. Research Questions 36

4.2.2. Quality Indicators 38

4.2.3. Studied Docker Projects 39

4.2.4. Parameter Settings 39

4.2.5. Statistical Tests 41

4.2.6. Results 42

4.3. Threats to Validity 53

4.4. Conclusion 54

CHAPTER FIVE
EFFICENT MANAGEMENT OF CONTAINERS FOR SOFTWARE
DEFINED VEHICLES 56

5.1. Software-related systems and Real-life use cases 61

5.2. Research Methodology 64

5.2.1. Scheduling Component 66

5.3. Validation 71

5.4. Threats to Validity 91

5.5. Conclusion 93

CHAPTER SIX
MIND THE GAP: THE DISCONNECT BETWEEN REFACTORING CRITERIA USED IN INDUSTRY AND REFACTORING RECOMMENDATION TOOLS 95

6.1. Methodology 98

6.1.1. Industry Interviews 99

6.1.2. Industry Survey 101

6.1.3. Analysis of Tools 103

6.2. RQ1: Refactoring Criteria 106

6.3. RQ2: Significance of Criteria 108

6.4. RQ3: Criteria Support in Tools 112

6.5. Closing Tool Gaps 114

6.5.1. Breadth of Criteria 115

6.5.2. Transparency of Criteria 117

6.5.3. Capturing Context 119

6.6. Threats to Validity 121

6.7. Conclusion 122

TABLE OF CONTENTS—Continued

CHAPTER SEVEN
BUILD CODE NEEDS MAINTENANCE TOO: A STUDY
ON REFACTORING AND TECHNICAL DEBT IN BUILD SYSTEMS 123

7.1. Methodology 126

7.1.1. Data Preparation 126

7.1.2. Qualitative Analysis methodology 128

7.2. Study Results 132

7.2.1. RQ1:Which refactoring types are developers applying to build code? 132

7.2.2. RQ2: How are build refactorings linked to technical debt? 142

7.2.3. RQ3: What is the effectiveness of BuildRefMiner in identifying build refactorings? 147

7.3. Threats To Validity 150

7.4. Study Implications 150

7.5. Conclusion 151

CHAPTER EIGHT
CONCLUSION AND FUTURE WORK 152

8.1. Conclusion 152

8.2. Future Work 153

LIST OF FIGURES

Figure 1.1	Proposed Research	5
Figure 1.2	Overview of the Thesis Methodology	7
Figure 2.1	Docker-compose example.	13
Figure 4.1	Proposed Approach.	29
Figure 4.2	Convert docker-compose file.	29
Figure 4.3	Solution representation.	31
Figure 4.4	Solution representation converted.	31
Figure 5.1	The Hardware Architecture of an Automotive System	62
Figure 5.2	Real-world use case: Power conservation	63
Figure 5.3	Real-world use case: Software upgrade/Failure operation	64
Figure 5.4	Architecture of the proposed framework.	65
Figure 5.5	The percentage (%) of memory usage per node (UC1).	78
Figure 5.6	The percentage (%) of CPU usage per node (UC1).	79
Figure 5.7	The percentage (%) of memory usage per node (UC2).	80
Figure 5.8	The percentage (%) of CPU usage per node (UC2).	80

Figure 5.9	Scheduling time (in seconds) per number of containers. For a more readable version, you can find it on the website [4].	81
Figure 5.10	Container Transfer Time (in milliseconds) Across Nodes for Different Container Sizes: High (e.g., Engine, Brakes), Average (e.g., Heater, Maps), and Small (e.g., Lights, Setup)	82
Figure 5.12	Containers allocated on the Manager ECU before rescheduling.	87
Figure 5.13	New distribution of containers.	88
Figure 5.14	Containers distribution after rescheduling.	90
Figure 5.15	% Of CPU consumption before and after rescheduling for the 3 ECUs.	91
Figure 5.16	% Of Memory consumption before and after rescheduling for the 3 ECUs.	92
Figure 6.1	Overview of multi-method approach.	99
Figure 6.2	Number of survey respondents (n=115) mentioning each refactoring criterion in open response comments.	109
Figure 6.3	Survey responses on the significance of refactoring criteria, showing counts of respondents (n=106) by category.	110
Figure 6.4	For each criterion, portion of survey respondents (n=106) identifying it as Significant/Very Significant against portion of refactoring recommendation tools (n=26) supporting it.	116
Figure 7.1	Approach Overview	126

LIST OF FIGURES—Continued

Figure 7.2	BuildRefMiner Prompt	131
Figure 7.3	Build-related refactorings taxonomy	133
Figure 7.4	Developer Response on Commit [5]	143
Figure 7.5	Developer Response of Commit [6]	143
Figure 7.6	Developer Response for Commit [7]	144
Figure 7.7	Developer Response for Commit [8]	145
Figure 7.8	Developer Response of Commit [6]	145
Figure 7.9	BuildRefMiner output on Commit [9]	149

LIST OF TABLES

Table 2.1	Docker-compose setup attributes.	13
Table 4.1	Docker-based projects studied.	40
Table 4.2	Parameter Settings.	41
Table 4.3	HV, IGD, and IC mean values with NSGA-III, NSGA-II, and IBEA. The results were statistically significant on 30 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%)	43
Table 4.4	HV, IGD, and IC mean values with NSGA-III, NSGA-II, and IBEA. The results were statistically significant on 30 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%)	44
Table 4.5	Average Computational time values on 30 independent runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).	47
Table 4.6	Average Computational time values on 30 independent runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).-Continued	48

Table 4.7	Number of selected nodes (NON), Average number of containers per nodes (FRQ), number of coupling (COP) mean values of NSGA-III over 30 independent simulation runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).	50
Table 4.8	Average of CPU, Memory, and Network usages comparing our approach and Spread in the cluster The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%).	51
Table 4.9	CPU, Memory, and Network usages comparing our approach and Spread for every node. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%).	52
Table 5.1	Heterogeneous environment.	74
Table 5.2	Parameter settings.	75
Table 5.3	HV, IGD, and IC mean values with Our Approach and IBEA. Results are based on 30 independent simulation runs using the Wilcoxon rank sum test with 99% confidence level (α<1%). (Highest values are in bold)	75
Table 5.4	HV, IGD, and IC mean values with Random Search (RS) and NSGAII. Results are based on 30 independent simulation runs using the Wilcoxon rank sum test with 99% confidence level (α<1%).	76

Table 5.5	Resource consumption before rescheduling.	85
Table 5.6	Resource consumption after rescheduling.	85
Table 5.7	Resource consumption before rescheduling.	87
Table 5.8	New resources consumption per node.	88
Table 5.9	Placement constraints for some containers in the cluster.	90
Table 6.1	Interviewee Demographics.	100
Table 6.2	Criteria for that focus on the refactoring process.	106
Table 6.3	Criteria that focus on the qualities of refactored code.	107
Table 6.4	Refactoring recommendation tools and the refactoring criteria that each supports.	113
Table 7.1	build refactoringTechnical Debt Categories	146
Table 7.2	BuildRefMiner Performance across the refactoring types with Zero-Shot and One-Shot Prompting.	149

CHAPTER ONE

INTRODUCTION

1.1 Research Context

In the dynamic realm of software development, the rapid evolution of technology and increasing software complexity demand continuous optimization to ensure performance, reliability, and maintainability[1, 2, 3]. As software grows and deviates from its original design, inefficiencies, complexity, and maintenance costs rise. Notably, industry trends show that 40% to 90% of software project budgets are allocated to maintenance, highlighting the pressing need for effective optimization strategies [10]. Therefore, software engineers need better ways to reduce and manage the growing complexity of software systems and improve their performance.

Managing the environmental resources needed for the software to be deployed and well executed is one aspect of software optimization. Software containers are becoming the new state of the art in the industry as they are extensively used to deploy software systems [11]. Indeed, the use of containers enables better modularity, reusability, and portability of the software compared to other traditional technologies [12]. For instance, the Docker container is considered one of the most important pillars for software deployment as it provides higher performance and flexibility in comparison with a traditional hypervisor-based virtualisation [11].

Moreover, software optimization extends beyond software resource management. Recent research within software engineering has highlighted the importance of software optimization, focusing also on the broader spectrum of software lifecycle processes: Source

code Refactoring [13, 14] is a technique that improves the design structure while preserving the overall functionality and behavior and has experienced tremendous adoption in Object-oriented systems [15, 16, 17].

Many refactoring recommendation tools—a class of tools that provide its users with a list of suggested refactorings (e.g., Move Method or Extract Interface)—have been suggested, that are standard with all major IDEs, and widely adopted in the industry to assist developers with this process.

Optimizing other aspects of the software development lifecycle, such as software build systems, is equally crucial for ensuring the overall efficiency and performance of software systems. Build systems, such as Maven [18], Ant [19] and Gradle [20] are responsible for transforming source code into executable programs by coordinating various compilation tools, including compilers and code generators. However, as software evolves, changes to source code increase the complexity of build systems, making them prone to technical debt.

This complexity of configuring build systems often leads to maintenance challenges and delays in development projects [21, 22, 23, 24]. For example, Seo et al. [23] found that up to 37% of builds at Google fail, while Kumfert et al. [24] estimate that build maintenance imposes a 12% overhead, diverting developers from core tasks. These findings emphasize the need for clean, robust, and well-maintained build systems to avoid inefficiencies and ensure smooth software development processes.

1.2 Problem Statement

Optimization is a comprehensive process that addresses multiple facets of the software lifecycle, from core aspects like software resource allocation management [25, 26, 27] to high-level tasks such as optimizing source code [28, 29, 30, 31, 32], refining CI/CD processes, and improving build system efficiency [33, 34]. Despite their critical

importance, current approaches and tools related to each optimization aspect often fall short, presenting several challenges that need to be effectively addressed to improve overall software performance and maintainability.

One topic of interest that needs to be addressed is related to the efficient management of software in different environments with different resource availability from cloud to edge. Docker and the software containerization technology are becoming the main parts of the cloud and edge strategies of most industry organizations [35]. Despite the benefits and popularity of using Docker containers, there are several challenges associated with the optimal usage of the resources consumed by this technology, as a large number of software in different sizes may need to be executed and orchestrated due to the high modularity of Docker architectures [36]. As a result, there is a critical need for efficient mechanisms to schedule and orchestrate the execution of software among many nodes in the cloud and edge environments. Thus, several software scheduling tools are proposed, such as Docker Swarm developed by Docker [25], Mesos by Apache [26], and Kubernetes by Google [27]. Despite their efficiency, existing strategies are often too simplistic to manage the complexities of modern software systems. For instance, Docker Swarm operates without prior knowledge of the workload or the specific resource requirements of containers. Its primary scheduling strategy, known as Spread, merely distributes containers evenly across all Docker hosts, requiring any additional configuration to be performed manually [37]. This limitation underscores the necessity for intelligent container management strategies. Effective resource allocation and workload balancing are essential for optimizing software performance and ensuring scalability in heterogeneous environments with varying requirements and constraints [38, 39, 40].

Another challenge in the software optimization domain is related to software source code refactoring. Nowadays, many refactoring recommendation tools have been widely adopted in the industry [28, 29, 30, 31, 32]. However, current refactoring tools do not

always align with the criteria and practical needs of industry practitioners, resulting in a gap between theoretical recommendations by researchers and practitioners' real-world applications.

The distance between them primarily originates from the lack of insights into both worlds' recent findings and needs. For instance, developers tend to use the refactoring features provided by IDEs due to their accessibility and popularity [28, 30, 41, 32]. Most of the time, they are not informed of the benefits that can be derived from adopting state-of-the art advances in academia [28, 30, 32]. Conversely, identifying the challenges that practitioners face when refactoring received little attention from academia. Studies on when developers refactor code and which criteria they choose to refactor are very limited.

The gap between what academics offer and what industry expects in terms of software source code refactoring is not the only issue in the software optimization field. When it comes to the CI/CD aspect, Software build systems, like source code, accumulate technical debt, posing challenges to the software's maintainability and optimization. To ensure the stability and functionality of software builds, developer teams could also prioritize the ongoing refactoring of the build. Thus, understanding the refactorings that developers apply to such systems is of the utmost importance. However, the refactoring of build systems, despite their crucial role in the software development process, remains an area with limited understanding [42]. There is no holistic understanding of the quality issues of build systems that could affect different artifacts beyond just the build code quality. Furthermore, the correction of these issues via refactorings has not been explored in the literature, unlike other refactoring domains [43].

These multifaceted challenges in software optimization demand comprehensive solutions that span from the core code structure to the apex of software deployment strategies.

1.3 Proposed Research

This research aims to advance the field of software optimization by addressing multifaceted challenges that span the entire software lifecycle, from software containers scheduling to source code refactoring and build system maintenance, as shown in Figure 1.1.

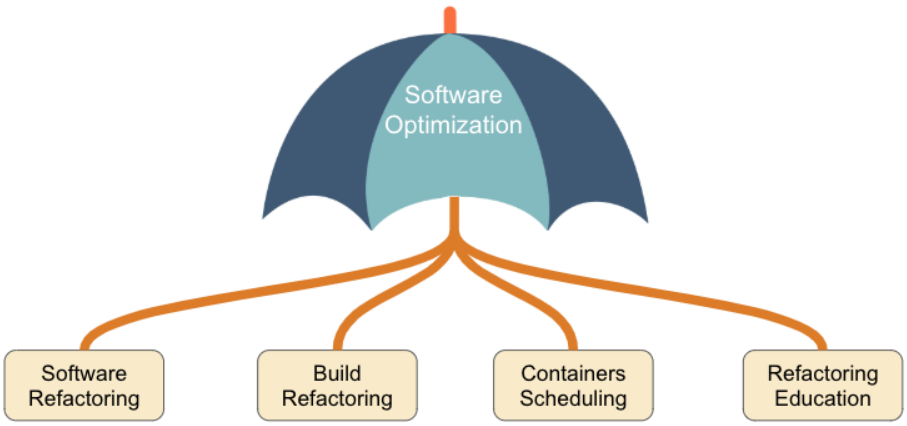

Figure 1.1: Proposed Research

The overarching objective is to enhance modern software systems performance, maintainability, and efficiency, ensuring they can meet the increasing demands of complex applications. We divided the progress of this thesis into three hypotheses answered in three phases, as detailed in Figure 1.2

Hypothesis 1:

A many-objective optimization approach using NSGA-III will significantly outperform existing software containers scheduling strategies in terms of workload balancing and resource utilization (e.g., CPU, memory) in both cloud and resource-constrained environments, such as software-defined vehicles.

Hypothesis 2: The refactoring criteria prioritized by industry practitioners differ significantly from those implemented in existing academic tools, and integrating

practitioner-informed criteria into refactoring tools will improve their perceived usefulness, practical adoption, and impact on code maintainability and developer productivity.

Hypothesis 3: Similar to source code, refactoring practices are also present within build code. These build system refactorings can be systematically classified into distinct categories, and their application is significantly correlated with the mitigation of technical debt in real-world build environments.

To validate the three hypotheses, we went through three main phases :

Phase 1 is about resource-aware container management, where a many-objective optimization approach using NSGA-III is proposed to improve workload balancing and resource utilization in cloud and edge environments. This approach addresses limitations in existing scheduling strategies, offering solutions tailored for resource-constrained scenarios like software-defined vehicles.

Phase 2 bridges the gap between academic advancements and industry practices in source code refactoring. The study identifies critical criteria used by developers in making refactoring decisions, aiming to inform the development of more practical and efficient tools. By aligning these tools with real-world needs, the research seeks to improve code maintainability and enhance developer productivity.

Finally, **phase 3** delves into build system refactoring by examining technical debt and creating a taxonomy of build-specific refactorings. It introduces BuildRefMiner, an AI-powered tool designed to automate the detection and analysis of build refactorings. This tool supports developers in maintaining efficient and reliable build systems, addressing a critical yet underexplored area of software optimization.

Together, these efforts redefine software optimization as an integrated, practical process that equips practitioners with effective tools and strategies for building robust and scalable systems.

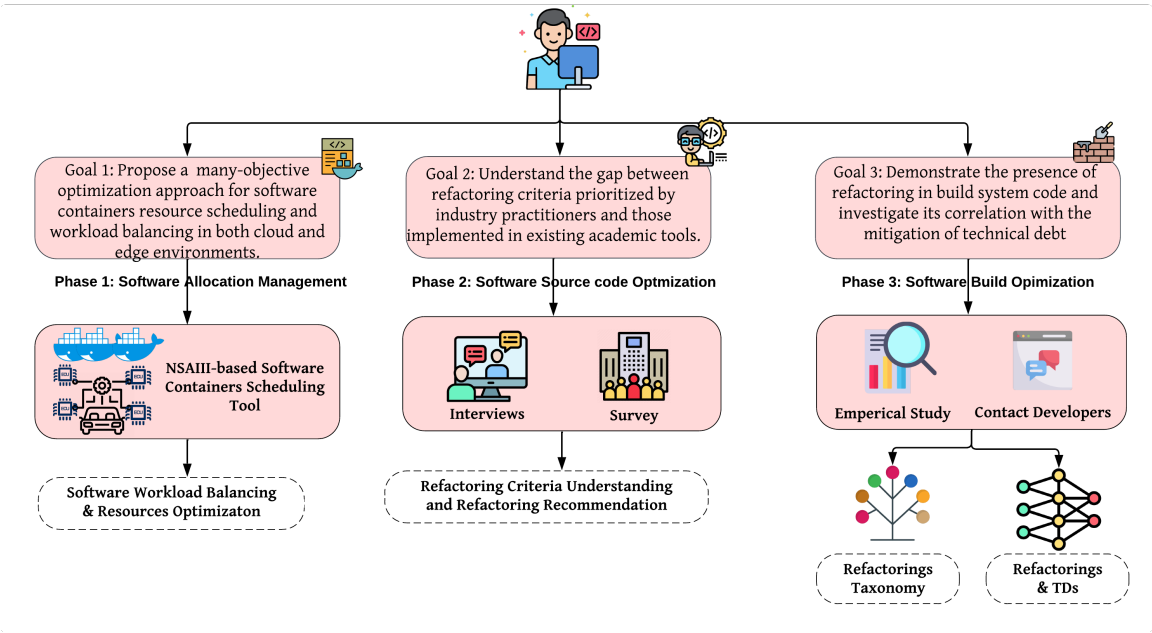

Figure 1.2: Overview of the Thesis Methodology

1.4 Research Contributions

In the following, we will summarize each contribution of this thesis:

1. **Multi-objective Optimization for Software Management (Hypothesis 1):**

   Given that software containers have become the state-of-the-art in the industry, extensively used for deploying software systems on both cloud and edge environments, it is crucial to develop an intelligent strategy for managing the allocation of these containers. Our research aims to address this need by proposing advanced methods for resource-aware container management, ensuring better performance and scalability of software systems.

   - **Contribution 1: Dynamic Software Containers Workload Balancing via Many-Objective Search**

As the complexity of software systems is dramatically increasing, it is critical to enable optimal usage of the needed resources to execute them such as memory and CPU. We proposed a scheduler based on a many-objective optimization approach for scheduling the execution of containers between multiple nodes in the cloud. The proposed approach aims at finding the best allocation for containers in nodes that leads to efficient utilization of resources. To evaluate our approach, we compared the performance of multiple many-objective and multi-objective techniques based on NSGA-II, NSGA-III, and IBEA algorithms using 48 Docker-related systems. Our results show that NSGA-III outperforms the other algorithms in quality attributes as well as in CPU, Memory and Network usage.

- **Contribution 2: Software Containers in Smart Vehicles: A Workload Management Approach**

  We propose to extend the previous contribution by considering edge environments where resources are more limited. We delve into the efficient management of containers for software-defined vehicles (SDVs). This work emphasizes the dynamic nature of SDVs and the need for robust container scheduling strategies to manage the complex constraints and resource constraints inherent in automotive environments. Using NSGA-III, we effectively optimize multiple conflicting objectives, ensuring efficient software distribution and enhanced performance of automotive software systems.

Having addressed the aspects of optimizing the workload management approach of software deployment, we now turn our focus to the optimization of the software code itself.

2. **Contribution 3: Mind the Gap: The Disconnect Between Refactoring Criteria Used in Industry and Refactoring Recommendation Tools (Hypothesis 2)**

   After addressing the aspects of optimizing the workload management approach of software deployment, we now turn our focus to the optimization of the software code itself. To contribute to the enhancement of software refactoring tools, we address the disconnect between industry practices and refactoring recommendation tools. Our study reveals that current tools fail to support many of the criteria that developers consider crucial when making refactoring decisions. By identifying these gaps, we provide a broader perspective for refactoring tool developers, enabling them to enhance their tools and make them more efficient and practical for industry practitioners, thereby enhancing code maintainability and developer productivity.

3. **Contribution 4: An Empirical Study of Refactorings and Technical Debt in Build Systems (Hypothesis 3):**

   We propose an empirical study of Builds systems-related refactorings in open-source projects. Our analysis was conducted on 725 examined build-file-related commits. We identified 24 build-related refactorings, which we divided into 6 main categories. These refactorings are organized into the first empirically derived taxonomy of build system refactorings. Furthermore, we investigate how developers employ these refactoring types to address technical debts via a manual commit-analysis and a developer survey. In this context, we identified 5 technical debts addressed by these refactorings and discussed their correlation with the different refactorings. Finally, we introduce BuildRefMiner, an LLM-powered tool leveraging GPT- 4o to automate the detection of refactorings within build systems. We evaluated its performance and found that it achieves an F1 score of 0.76 across all build systems. This study will

serve as a foundational building block for guiding future research and practice in the maintenance and optimization of build systems.

1.5 Publications List

- **Anwar Ghammam**, Dhia Rzig, Mohamed Almukhtar, Rania Khalsi, Marouane Kessentini, and Foyzul Hassan: Build Code Needs Maintenance Too: A Study on Refactoring and Technical Debt in Build Systems. **22nd International Conference On Mining Software Repositories MSR (2025) [Class A]. Acceptance rate: 27%**

- James Ivers, **Anwar Ghammam**, Khouloud Gaaloul, Ipek Ozkaya, Marouane Kessentini and Wajdi Aljedaani Mind the Gap: The Disconnect Between Refactoring Criteria Used in Industry and Refactoring Recommendation Tools. **The International Conference on Software Maintenance and Evolution (ICSME 2024) [44] [Class A], Acceptance rate 26 %**

- **Anwar Ghammam**, Rania Khalsi, Marouane Kessentini, and Foyzul Hassan: Efficient Management of Containers for Software Defined Vehicles, **IEEE Transactions on Software Engineering And Methodology (TOSEM 2024) [45] [Q1 IF 6.6]**

- Wajdi Aljedaani, **Anwar Ghammam**, Mohamed Wiem Mkaouer, and Marouane Kessentini: From Boring to Boarding: Transforming Refactoring Education with Game-Based Learning, **ICSE 2024, Games and Software Engineering Workshop (GAZ) [46]**

- Anwar Ghammam, T Ferreira, Wajdi Aljedaani, Marouane Kessentini, Ali Husain: Dynamic Software Containers Workload Balancing via Many-Objective Search, **IEEE Transactions on Services Computing Journal (TSC-22) [47] [Q1 IF 5.5]**

- **Anwar Ghammam**, Mohamed Almukhtar, Marouane Kessentini: About the relation between software Context and refactoring **(Submitted to TSE 25 (Impact Factor 6.11 - Under Review)**

1.6 Organization of the Dissertation

This thesis is organized as follows: Chapter II provides the necessary background information to understand the research contributions. Chapter III offers a summary of the related work. Chapters IV and V present our intelligent tool for software containers management in cloud and edge environments, respectively. Chapter VI seeks to understand the gap between refactoring criteria used in the industry and refactoring recommendation tools. Chapter VII presents an empirical study that leads to creating a taxonomy for build-specific refactoring practices and technical debt. Finally, Chapter VII concludes with a summary of the research and discusses potential future directions.

CHAPTER TWO

BACKGROUND

This chapter provides the essential background knowledge required to fully understand the research presented in this thesis. We first explore fundamental concepts such as Docker, container-based projects, and NSGA-III, which form the core of the first and second contributions. Next, we delve into the concepts of software refactoring and technical debt—key aspects that influence the quality and maintainability of software systems. Finally, we give an overview of the most common build systems used to build the software. By establishing a clear understanding of these topics, this chapter lays the foundation for the methodologies and analyses discussed in subsequent chapters of this thesis.

2.1 Docker and Container-based Projects

Docker [48], is one of the most popular container virtualization technologies [49, 35]. It packs the application's code and dependencies into a lightweight, standalone, and portable execution environment, aiming to deploy containerized applications in a quick process.

Docker-compose [50] provides a unified setup routine that deploys several containers using a YAML configuration file, as known as, `Docker-compose.yml` (or just Docker-compose). In the Docker paradigm, each container captures one particular component of the software (e.g., database). Thus, when creating a multi-component application using Docker, it is inevitable to combine multiple software components (containers) into a workflow. Then, Docker-compose can tackle this problem by integrating containers and running them properly.

An example of a Docker-compose file is available in Figure 2.1. The example shows that the Docker-compose file is composed of two components/containers (SERVER and DATABASE). The SERVER component is represented by a local image, and the DATABASE

component is created from the “mongo” image, hosted in DockerHub [51] (an online registry for Docker Images). Docker-compose file also provides a list of setup attributes which can be listed in Table 2.1.

```
services:
    server:
      image: api:latest
      deploy:
        placement:
          constraints:
             - node.role == manager
      depends_on:
            - database
    database:
      image: mongo:latest
```

Listing 1. Docker-compose example.

Figure 2.1: Docker-compose example.

Table 2.1: Docker-compose setup attributes.

Attribute	Description
BUILD	Setting path to the build context
IMAGE	Setting the image to start the container from
PORTS	Specify ports binding
ENVIRONMENT	Setting environment variables
DEPENDS_ON	Expressing dependency between services
VOLUMES	Setting volume bindings (host paths or named volumes)

Any typical Docker project includes the above files along with source code files written in typical programming languages, such as Java, to host the containers and enable their executions and synchronization with other features of the app that may not be containerized.

The way that tasks or containers are scheduled on a Swarm Mode cluster is governed by a scheduling strategy. Currently, Swarm Mode has a single scheduling strategy, called "Spread" [37]. The spread strategy attempts to schedule a service task based on an assessment of the resources available on cluster nodes. In its simplest form, this means that tasks are evenly spread across the nodes in a cluster. For example, if we create a service with three replicas, each replicated task will be scheduled on a different node.

2.2 Multi-Objective Optimization Algorithms

In many real-world problems, especially in software engineering, optimizing for a single objective is insufficient. Systems often face competing goals—for example, improving software performance while minimizing resource consumption, or enhancing maintainability while reducing code redundancy. Multi-objective optimization (MOO) aims to solve such problems by simultaneously optimizing multiple, often conflicting, objectives. Instead of producing a single optimal solution, MOO algorithms typically generate a set of Pareto-optimal solutions, where no one solution is universally superior to the others across all objectives [52].

Evolutionary algorithms have become a dominant strategy for solving MOO problems due to their ability to efficiently explore large, complex, and non-convex search spaces. Among these, algorithms like NSGA-II [53], IBEA [54], and MOEA/D [55] have gained wide adoption for their balance between convergence and diversity in solution spaces. These algorithms evaluate populations of solutions and iteratively refine them using mechanisms inspired by natural selection, such as crossover, mutation, and fitness-based selection. In the following subsection, we provide a more detailed overview of NSGA-III, the multi-objective optimization (MOO) algorithm employed in our contributions presented in Chapters 4 and 5.

2.2.1 NSGA-III: A Reference-Point-Based MOO Algorithm

Non-dominated Sorting Genetic Algorithm III (NSGA-III) is a more recent optimization algorithm proposed by Deb et al. [56, 57], similar to NSGA-II, but with significant changes in its selection mechanism aiming to improve the results of many-objective problems. Unlike in NSGA-II, the diversity among population members in NSGA-III is aided by supplying a number of well-spread reference points.

NSGA-III demonstrates its efficacy in solving 2 to 15-objective optimization problems, and it is also extended easily to solve constrained optimization problems, and can be used with small population size (such as a population of size 100 for a 10-objective optimization problem). The algorithm is shown in Algorithm 2.1.

Algorithm 2.1: Generation t of NSGA-III. Adapted from [56, 57]

1. H structured reference points Z_r or supplied aspiration points Z_a , parent population P_t P_{t+1} $S_t = \emptyset$, i = 1
2. Q_t = Recombination+Mutation(P_t)
3. $R_t = P_t \cup Q_t$
4. (F_1, F_2, . . .) Non-dominated-sort(R_t)
5. **repeat**
6. $\quad S_t = S_t \cup F_i$ and i = i + 1
7. **until** $|S_t| \geq N$
8. $F_l = F_i$(Last front to be included)
9. **if** $|S_t| = N$ **then**
10. $\quad P_{t+1} = S_t$, break
11. **else**
12. $\quad P_{t+1} = \bigcup_{j=1}^{l-1} F_j$
13. $\quad$ Points to be chosen from F_l : K = N - $|P_{t+1}|$
14. $\quad$ Normalize objectives and create reference set Z_r : Normalize(f^n, S_t, Z_r, Z_s, Z_a)
15. $\quad$ Associate each member s of S_t with a reference point: [$\Pi(s)$, d(s)] =Associate(S_t , Z_r) {Π(s): closest reference point, d: distance between s and $\Pi(s)$ }
16. $\quad$ Compute niche count of reference point j $\in Z_r$: φ_j=$\sum_{s \in S_t/Ft}$ ((Π(s) = j) ?1 : 0)
17. $\quad$ Choose K members one at a time from F_l to construct P_{t+1} : Niching(K, φ_j,Π, d, Z_r , F_l, P_{t+1})
18. **end**

First, same as NSGA-II, the parent population P_t is randomly initialized in the specified domain, then the binary tournament selection, crossover, and mutation operators are applied to create an offspring population Q_t (Line 1–2). Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected for the next generation.

Unlike in NSGA-II (which uses the crowding distance measure for selecting the best set of points from the last front that can be partially accepted), in NSGA-III the supplied reference points Z_r are used to select these remaining members. The chosen reference points can either be predefined in a structured manner or supplied preferentially by the user. To accomplish this, objective values and reference points are first normalized to have an identical range. Thereafter, the orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is calculated.

Next, the member is then associated with the reference point having the smallest orthogonal distance, and the niche counts φ for each reference point, defined as the number of members in S_t/F_l that is associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member in front last front F_l that is associated with the identified reference point is included in the final population. The niche count of the identified reference point is increased by one, and the procedure is repeated to fill up population P_{t+1}.

2.3 Refactoring & Technical Debt

Refactoring is defined as the process of restructuring existing code without changing its external behavior to improve readability, maintainability, and extensibility[58]. It is a routine activity in software development, with scopes ranging from small changes (sometimes called floss refactoring [59]) to broad changes with architectural implications (sometimes called large-scale refactoring [60]).

Technical debt is defined as the cost of additional work created by choosing an easy solution now instead of using a better approach that would take longer[58]. It is a metaphor that describes the consequences of poor software design and implementation decisions. Technical debt can manifest in various forms in Build files, such as design flaws and outdated dependencies. Over time, technical debt can accumulate and slow down development, increase maintenance costs, and reduce software quality[58]. Refactoring can help reduce technical debt by improving the design and structure of the codebase. In the context of build code, refactoring can involve simplifying build scripts, removing duplication, and improving maintainability [61].

2.4 Build Systems

Build automation is a critical part of modern software development[21]. Build tools provide flexible ways to model, manage, and maintain complex software projects. They rely on configuration files to define dependencies, specify build tasks, and orchestrate build process workflows[62, 63, 64, 65]. Within this work, we specifically focus on Ant, Maven, and Gradle, widely-used build tools that provide a good representation of the different approaches to build automation.

Ant uses an XML-based highly customizable build script to define tasks and dependencies. Ant does not have predefined build lifecycle phases. Instead, build targets can be connected to indicate dependencies. **Maven** also uses an XML build script, centered around its project object model (POM.xml), which defines project configuration, dependencies, and settings. Rather than customizable tasks, Maven relies on plugins bound to these phases to execute goals in a consistent sequence. Finally, **Gradle** builds on Ant and Maven using a Groovy or Kotlin domain-specific language (DSL) for its build scripts. This allows flexible modeling of build requirements in a declarative, extendable way. Gradle

build files consist of components like plugins, dependencies, configurations, tasks[1], and properties.

[1] units of work or action in a build system (e.g., compiling, packaging).

CHAPTER THREE

STATE OF THE ART

3.0 Introduction

In this chapter, we give an overview of the state of the art of software optimization from different aspects.

3.1 Software Allocation Management

In the current literature, there have been few attempts to address software management [38, 39, 40, 66, 67, 68, 69, 70, 47].

Guerrero et al. [38] suggested a container allocation strategy and elasticity management by optimizing the elasticity of presently deployed applications and maximizing the reliability of micro-services by avoiding single points of failure. Liu et al. [39] proposed a new container scheduling algorithm based on multi-objective optimization, namely Multiopt. This approach aimed to optimize the performance of docker containers using five key factors: the resource usage of every cluster node (CPU, Memory), the clustering of containers, the association between nodes and containers, and the time consumption for transmitting images on the network. Menouer et al. [40] suggested enhancing the Spread strategy by introducing a new container scheduling strategy based on the power consumption of heterogeneous cloud nodes. Their novel approach consisted of finding the optimal compromise to minimize the global energy consumption of heterogeneous cloud infrastructure. Kaewkasi et al. [66] focused on applying meta-heuristic algorithms. They adopted the Ant Colony Optimization algorithm (ACO) to implement a new container scheduler for SwarmKit, the purpose of which was to balance the use of resources so that applications in the container cluster would have better performance.

However, limited resources are rarely observed in the cloud, unlike with edge devices in cars. The majority of existing scheduling solutions for software containers are primarily developed for cloud environments with adequate resources, as their primary focus is to orchestrate the execution of the containers to accomplish specific tasks.

Other scheduling techniques [67, 68] consider the limited resources (such as CPU, memory, and energy) on edge devices and aim to efficiently schedule software containers in order to optimize resource usage. Feifei Chen et al. [69] proposed a container Scheduling Method in Edge Computing. This strategy, based on the Min-Min algorithm, tried to place a container on the physical machine with the least increase in energy consumption as a way to lower the cluster's energy usage. Thus, the approach was restricted to reducing the power consumption in edge devices. Fuming Zhang et al. [70] proposed a deep deterministic policy gradient (DDPG) based online joint task scheduling (OJTS) algorithm to solve the problem of joint scheduling of delay-sensitive and computation-oriented tasks in resource-constrained EC scenarios. However, the proposed method did not consider optimizing resource usage by software containers. These present systems, however, have different shortcomings when scheduling containers in edge e.g smart vehicles. They fail to address the physical environment needs such as the real-time requirements and complex constraints that arise in real time and find trade-offs between these competing objectives and constraints. The studies referenced above primarily focus on managing software allocation and scheduling in various environments to achieve resource balancing, ultimately enhancing software performance. In this proposal, we introduce a many-objective approach for software container management both in the cloud and at the edge. Our approach not only considers resource consumption but also addresses various constraints and real-time scenarios that may occur in these environments. The following sections provides a detailed description of the proposed approach.

3.2 Software Source Code Refactoring

Related work that addresses refactoring criteria primarily comes from research studying different ways to generate refactoring recommendations. Tools that recommend refactorings analyze code and propose refactorings, typically employing criteria oriented to removing code smells or improving code quality metrics. Though multiple studies highlight the importance of integrating developer or user preferences into refactoring tasks [71][72], the means of integrating these preferences vary across tools.

Smell-based Recommendations. Recommendation tools that focus on removing code smells (i.e., identifiable patterns in code that indicate the presence of design problems [28][29]) implement heuristics based on known solutions to the code structures that indicate a code smell [30][31]. These heuristics result in the generation of specific refactoring recommendations to resolve code smells, though the set of code smells varies across tools.

For example, JDeodorant [32], addresses five code smells (Feature Envy, Type/State Checking, Long Method, God Class, and Duplicated Code). It uses pattern matching to detect these smells and suggests targeted refactorings. To detect a Long Method, JDeodorant identifies excessively lengthy methods that usually indicates that the method is performing more tasks than it should, thereby violating the single responsibility principle. To fix this, JDeodorant suggests breaking down the lengthy method into smaller pieces through a process known as method extraction. This involves creating new methods for each distinct task or logical block of code within the original method, to improve readability and maintainability. The Duplicated Code smell is identified when JDeodorant finds identical or very similar code fragments in multiple locations. It suggests pulling the common code into a new method and replacing the duplicates with calls to this method, thus adhering to the DRY (Don't Repeat Yourself) principle.

TrueRefactor [73] also addresses five code smells (Lazy Class, Long Method, Large Class, Shotgun Surgery, and Temporary Field), only one of which overlaps with those

addressed by JDeodorant. TrueRefactor detects the Shotgun Surgery code smell by recognizing scenarios in which a single modification leads to changes across many different code locations. It suggests consolidating these dispersed modifications into a single location, thereby streamlining the maintenance and enhancing the code structure.

Smell-based recommendations tend to be prescriptive, often leading to a single, well-defined corrective action. These tools often provide little ability for users to control the criteria employed as the criteria that inform the resolution of each code smell are embedded in the tools' heuristics. User choice may be limited to choosing which code smells to remediate.

Metrics-based Recommendations. Recommendation tools that focus on improving code quality metrics typically implement optimization techniques, and many use multi-objective optimization to integrate multiple refactoring criteria [74]. Tools using multi-objective approaches generate multiple comparable recommendations on a Pareto front that represents different trade-offs among objectives. Some tools allow users to influence recommendations by selecting the objectives to be optimized. All such tools allow users to choose solutions that best match their preferences.

For example, Refbot [75] uses a multi-objective approach to recommend refactorings that optimize code quality metrics like extendibility, flexibility, and reusability. To improve flexibility, it could suggest adopting design patterns like Strategy or State, which allow the behavior of the software to be changed dynamically. To improve reusability, Refbot may identify common patterns or code blocks and suggest extracting them into reusable methods or classes, reducing redundancy and facilitating code reuse.

Another example, A-CMA [76], is a configurable refactoring recommendation tool that supports more than 20 code measures such as Coupling, Cohesion, and Dead Code, and has been used in refactoring studies by other researchers [77][78]. The selection of criteria that are optimized has a substantial impact on the final recommendations.

Metrics-based recommendations often use optimization algorithms that support multiple criteria and allow users to choose criteria directly, which provides more transparency to users. However, the rationale for inclusions of different criteria in different metrics-based tools is not clear.

3.3 Build Code Maintenance

Recent research has highlighted the significance of build code maintenance as a software system evolves. Mcintosh et al. [33] have investigated the co-change of source and test code with build files. However, they did not investigate the specific types of build changes, unlike our sharp focus on build refactorings. Macho et al. [34] have extracted detailed build changes from Maven build files. They propose a taxonomy of build changes consisting of 95 change types of 5 main categories, and study their frequency. With these results, authors provide the basis for further research, such as for analyzing the (co-)evolution of build files with other artifacts. However, all of the change types studied are CRUD (create, remove, update, delete) changes to meet new build requirements. Hence, they are not refactorings as they modify the systems behavior. Also, they are only limited to Maven without digging into other well-known build tools such as Ant and Gradle. In their study, Hardt et al. [79] have introduced Formiga for Ant, which aims to facilitate build maintenance and dependency discovery. It only includes a limited set of rename and code cleanup refactorings. Simpson et al. [61], have discussed the need for Ant build files refactoring and list 23 such refactorings. However, both these works are focused only on Ant build systems, and the taxonomy/categories provided are based on the Author's experience and subjective opinions, contrasting with our research, which incorporates a broad, empirically grounded quantitative and qualitative analysis that also includes the more modern Maven and Gradle build systems. Shridhar et al. [80] conducted a qualitative analysis of the build commit history of 18 open-source projects from the Maven and Ant

systems. The study involved the manual tagging and classification of change types and build system ownership styles into six distinct categories. Their study highlights that changes are predominantly functional, and while "Preventive" and "Perfective" improvements are mentioned, they are not a focus of this study and the different refactorings that may constitute these changes are not detailed. In their research, Xiao et al. [81] analyzed 500 commits across 291 projects' Maven build systems, identifying technical debt (TD) primarily arising from tool constraints, library limitations, and dependency management challenges. They also notified developers of TD presence and, in some cases, submitted pull requests to address rudimentary TDs related to dependency incompatibilities. However, they did not investigate or propose exhaustive and generic refactoring approaches, nor did they link specific fix and TD categories. Our study proposes a deeper investigation into build system refactorings and how they relate to technical debt. It builds on previous research by empirically examining various build refactoring changes made by practitioners. We aim to make their findings more accessible and useful by offering a tool, BuildRefMiner, which automates the detection of build refactorings to support ongoing research in this field.

CHAPTER FOUR

DYNAMIC SOFTWARE CONTAINERS WORKLOAD BALANCING VIA MANY-OBJECTIVE SEARCH

4.0 Introduction

Software containers are becoming the new state of the art in the industry as they are extensively used to deploy systems [11]. Indeed, the use of containers enables better modularity, reusability, and portability compared to other traditional technologies [12]. For instance, the Docker container is considered one of the most important pillars for software deployment as it provides higher performance and flexibility in comparison with a traditional hypervisor-based virtualisation [11]. Thus, Docker and the software containerization technology are becoming the main parts of the cloud strategies of most industry organizations [35]. Despite the benefits and popularity of using Docker containers, there are several challenges associated with the optimal usage of the resources consumed by this technology as a large number of containers may need to be executed and orchestrated due to the high modularity of Docker architectures [36].

Due to the dramatic increase in the number and size of containers needed to build systems, there is a critical need for efficient mechanisms to schedule and orchestrate the execution of containers among many nodes in the cloud clusters. Thus, several container scheduling tools are proposed, such as Docker Swarm developed by Docker [25], Mesos by Apache [26], and Kubernetes by Google [27]. Generally speaking, despite their efficiency, these strategies are too simple to handle complex container execution. It is assumed that Docker Swarm has no prior knowledge regarding the workload or the container's resource requirements. The only available scheduling strategy is called Spread, which basically schedules a service task based on spreading the number of containers equally to all Docker hosts, and all the extra configuration has to be performed manually [37]. Thus, it is

important to create more sophisticated, high-level, and adaptive allocation strategies in order to guarantee a balanced workload among devices, the service's performance requirements, efficient communications between containers, and more efficient utilization of resources in terms of CPU and memory.

To deal with these challenges, several scheduling techniques for containers were recently proposed [66, 82, 83, 84]. Most of them are based on the use of optimization techniques due to the complexity of the problem in terms of the number of scheduling alternatives. For instance, Kaewkasi et al. [66] adopted the Ant Colony Optimization (ACO) algorithm to implement a new container scheduler for SwarmKit which spread the containers over Docker hosts to balance the overall resource usages and therefore lead to increased performance of applications. The proposed approach continuously computes the available resources in every node every time it schedules a container. Guerrero et al. [82] proposed a genetic algorithm approach to implement a container allocation strategy and elasticity management by optimizing the elasticity of the currently deployed applications and maximizing the reliability of the micro-services by avoiding single points of failure. However, the proposed strategies are limited to only two objectives while many conflicting criteria should be taken into consideration within the container scheduling problem. Indeed, an efficient scheduling approach for containers may need to consider the available resources in terms of memory and CPU but also reduce the changes in the current configuration (e.g. moving containers between nodes), the communications between containers located in different nodes (e.g. coupling) and balancing the software load between multiple nodes.

To address the above challenges, we propose a new many-objective optimization approach for the Docker containers scheduling problem. The goal is to find the best allocation of containers that can lead to a better workload balance and performance. The number of scheduling combinations to assign containers to nodes is high. Thus, the search space to explore is combinatorial which requires the use of an intelligent computational

search technique. Furthermore, the different scheduling criteria are conflicting thus, we adopted a many-objective search algorithm, based on NSGA-III [85], to find a trade-off between four conflicting objectives. Our many-objective search-based software engineering approach aims at finding the rescheduling solution that: optimizes the structure of the cluster by optimizing some metrics such as the number of selected nodes in the cluster, the average of containers per node, optimizes the communications between containers by minimizing the coupling (dependent containers allocated to different nodes), and finally minimizing the number of required changes to move from the current scheduling to the new one (e.g., move container) to guarantee a fast allocation of containers to the cluster nodes.

To evaluate our approach, we compared the performance of multiple many and multi-objective techniques based on NSGA-II, NSGA-III [56, 57], and IBEA algorithms [86] using 48 Docker-related systems. The results show that NSGA-III can generate the best scheduling solutions considering the traditional quality indicators for computational searches, such as Hypervolume, IGD, and Contribution metrics, as well as other validation metrics such as CPU, memory, and network usage. We have also created an online appendix for a demo of our platform and related experiments material [87].

The primary contributions of this work can be summarized as follows:

- We introduced a novel formulation of the containers scheduling problem as a many-objective problem that considers several conflicting objectives such as structural improvement, coupling, and the number of changes. The definition of the fitness functions was formulated based on the needs of our industry partner, the Ford Motor Company, to optimize specific objectives related to the usage of ECUs resources in the car.

- We compared three different many-objective optimization algorithms as it is the first formulation in the literature of container scheduling as a many-objective problem.

- We reported the results of an empirical study of our many-objective technique compared to the docker default approach. The obtained results provide evidence to support the claim that our proposal is, on average, more efficient than the existing techniques based on a benchmark of 48 open-source docker projects.

4.1 ClusterWare: Many-Objective Scheduling Approach for Software Containers

We present, in this section, an overview of the proposed scheduling approach for assigning software containers to the nodes, then we explain the different adaptation steps of the computational algorithm to our problem, including the solution representation and the fitness functions.

4.1.1 Approach Overview

The main goal of the proposed approach is to schedule containers by considering four conflicting objectives to be optimized. Each solution generated by the evolutionary algorithm represents a possible container scheduling by assigning the containers into nodes. Figure 4.1 shows an overview of the proposed approach composed of three main components. The first component is a parser that automatically extracts from Docker Cli (e.g., command line) the initial swarm state in the cluster, including the number of nodes, the total number of containers, their images, and their distribution per node. The docker-compose file is also parsed to extract the dependencies between containers using the parser tool. The second component takes the different information collected by the parser, including the extracted dependency graph and the swarm state, as inputs to generate a new swarm state using a many-objective optimization algorithm to find a balance between the different objectives. The third component executes the best solution found by the multi/many-objective algorithm by updating the docker-compose file to specify a new

placement for every container. Then, the docker-compose file is deployed again, and the load-balancing module reallocates the containers as suggested.

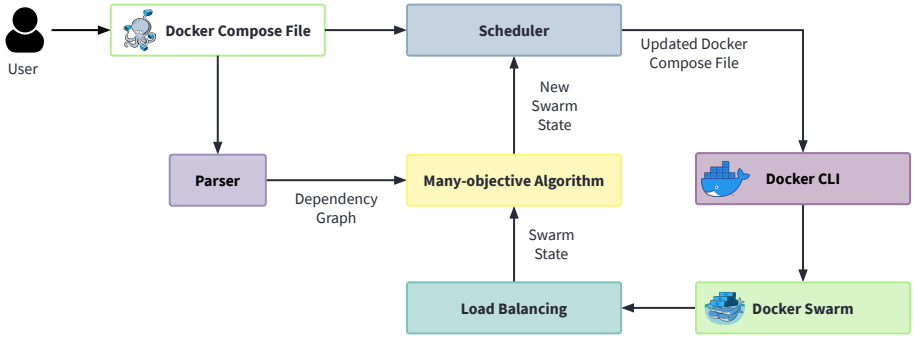

Figure 4.1: Proposed Approach.

In our approach, the user provides a docker-compose file as input, and then a Parser tool is used for generating a dependency graph $G = (V, E)$ where $V = \{v_1, v_2, v_3, \ldots, v_n\}$ means the set of containers or services and E is the set of calls or requests among them. The latter is written as a tuple $\{v_i, v_j\}$, where $v_i, v_j \in V$, and they are usually expressed in the docker-compose file as DEPENDS_ON or LINKS properties. Figure 4.2 shows an example of such conversion.

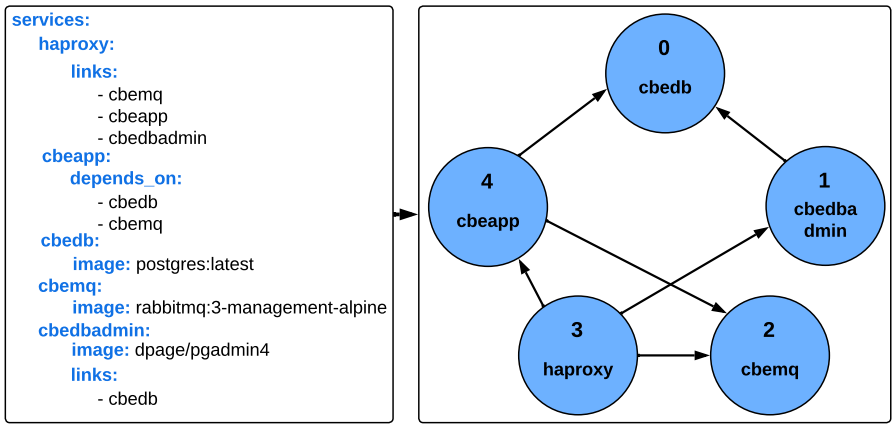

Figure 4.2: Convert docker-compose file.

In this example, five services were converted to a graph G with five nodes and six edges. Aiming to ease the node's assignment, we assign a unique identifier (id) to every container/service and node to be used in the optimization process. Thus, let's consider 0, 1, 2, 3, and 4 as the id's for the following containers, respectively: CBEDB, CBEDBADMIN, CBEMQ, HAPROXY, CBEAPP. Then, the dependency graph generated for such example is $V = \{0,1,2,3,4\}$ and $E = \{\{1,0\},\{3,1\},\{3,2\},\{3,4\},\{4,0\},\{4,2\}\}$.

The selected many-objective algorithm uses this dependency graph G and a Swarm State $P^{(t)}$ as input, where the latter means the current allocation of containers and nodes in Docker Swarm mode. Then, by taking into account the set of objectives to be optimized (details in subsection 4.1.3), a new Swarm State $P^{(t+1)}$ is generated, and the docker-compose file is changed aiming to reflect the new scheduling (see Scheduler in Figure 4.1).

Docker Swarm service is based on a declarative model, which means that once the service runs, we are not allowed to move or replace containers when some node gets started. Thus, to bypass such limitations, we generate a new docker-compose file by changing the CONSTRAINTS property from the file. The Load Balancing module in our approach is responsible for monitoring the current state of Docker Swarm by considering several metrics such as CPU and Memory usage, network metrics, and so on. In our approach, we can define some thresholds for each of them, and once such thresholds are reached, the many-objective algorithm is automatically run to reschedule the containers. Finally, if the Swarm State is unavailable (for instance, when we are running the proposed approach for the first time), all containers are randomly assigned to nodes to compose this one.

In the following subsections, we describe the different adaptation steps of the multi/many-objective algorithms described in Section 2, including mainly the solution representation and the fitness functions.

4.1.2 Population Representation

An individual (or solution) in our population consists of an integer encoding, where each gene represents the node id, and the index is the container id. By using this approach, we have the advantage of the flexibility of having more than one container per node. Thus, let $S = \{s_1, s_2, s_3, \ldots, s_n\}$ be an individual with n containers. Figure 4.3 shows an example of an individual considering the containers available in Figure 4.2.

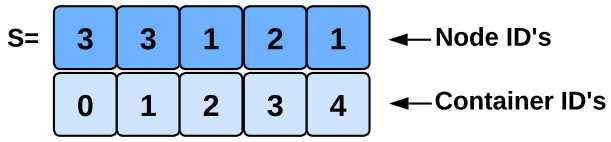

Figure 4.3: Solution representation.

In this example, the chromosome representation uses five containers ($n = 5$) and a total of three nodes. The individual S represented in figure Figure 4.3 assigns containers 2 and 4 to node 1, container 3 to node 2, and containers 0 and 1 to node 3. Figure 4.4 shows a visual representation considering all containers and nodes (consider the dependency graph available in Figure 4.2).

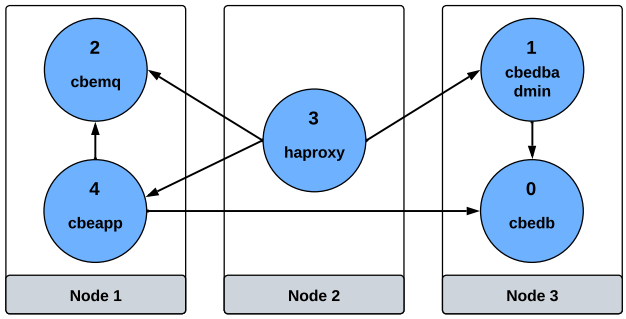

Figure 4.4: Solution representation converted.

4.1.3 Objective Functions

Even though different proposed scheduling techniques for containers [66, 82, 83, 84]. The proposed strategies are limited to at most two objectives, while many conflicting criteria should be taken into consideration within the container scheduling problem. We believe that our chosen objectives are extensively constructed based on preliminary research [85, 88] to obtain the optimal design that considers the most essential container attributes and scheduling limitations. We expect these functions to be valuable in future software container management efforts.

Consider $C = \{c_1, c_2, c_3, \ldots, c_n\}$ the set of all available containers, and $N = \{n_1, n_2, n_3, \ldots, n_m\}$ the set of all available nodes. The objective functions proposed in this work are described as follows:

First Objective: Minimizing the number of selected nodes (Equation 4.1).

The first objective corresponds to the number of selected nodes when rescheduling containers. Software containerization in many domains, such as smart automobiles [89], connected vehicles [90, 91], or different other domains [92], becomes critical, particularly in highly constrained environments For example, best practices [93] recommend that the load associated with one docker cluster node be lightweight to minimize congestion problems while running the applications, which explains the usage of the objective: Minimizing the number of containers per node. This would avoid exceeding the resource consumption limits that might affect the behavior of the node that deploys the software.

This objective is described using (Equation 4.1). ***NON*** computes the ratio of the number of selected nodes over the total number of available nodes. It is computed as follows:

$$NON(S) = \frac{|distinct(S)|}{|N|} \tag{4.1}$$

Second Objective: Minimizing the average of containers per node (Equation 4.2)

Minimizing the number of containers per node can avoid exceeding the resource consumption limits. However, minimizing the resource consumption of each node leads to activating a large number of nodes and spreading the load across them, on the other hand, would considerably increase the cluster's resource utilization, so adding considering minimizing the average number of containers per node in the cluster as a conflicting objective to the first is important in our context. For example, if we have 10 containers and 10 nodes, the worst solution would be to allocate every container to a single node; thus, the nodes will consume more resources. An optimal solution would be to choose a smaller number of nodes to distribute the containers while respecting the load between them and their resource limits. At the same time, we don't want to allocate all containers to one node so we can prevent violating the resource limits of the nodes and try to balance the node between the different nodes. Thus, minimizing the number of containers per node is a second objective to optimize despite conflicting with the first objective.

This objective is related to the average number of containers per node. The objective is calculated based on the normalized standard deviation taking into account the number of containers for each node. The objective is defined as:

$$FRQ(S) = \sqrt{\frac{1}{|F|} * \sum_{f_i \in F} (f_i - \mu)^2} \tag{4.2}$$

where $F = \{f_1, f_2, f_3, \ldots, f_m\}$ is the number of containers for each node m divided by $|C|$ and μ is the mean of F. For example, if $S = \{2, 3, 2, 1, 2\}$, then $F = \{0.2, 0.6, 0.2\}$.

Third Objective: Minimizing the nodes coupling (Equation 4.3)

The third objective, "minimizing the coupling between containers allocated to different nodes", can be considered a security objective that helps save the data and the good performance of applications deployed in containers in case one of the nodes has been shut down for a software upgrade or operational failure: Containers sharing data or depending

from each other are better to be running in the same physical ecus. (This reduces the risks of losing data or performance when for example, a container running in a different node and necessary for the work of another important container is shut down because of node failure), and also reduces the network transmission between the nodes. Although important, this objective conflicts with the objectives related to minimizing the number of containers per node.

This is expressed as a ratio between the number of inter-edges (calls or requests) in different nodes and the total number of edges E. The objective is defined as follows:

$$COP(S) = \frac{\sum_{\{a,b\} \in E} OP(S, a, b)}{|E|} \tag{4.3}$$

where $OP(S, a, b) = 1$ if $s_a \neq s_b$ for $s_a, s_b \in S$, otherwise, $OP(S, a, b) = 0$. If $|E| = 0$, then $COP(S) = 0$.

Fourth Objective: Minimizing the number of changes (Equation 4.4)

Finally, the fourth objective corresponds to the number of changes required to reschedule the containers. The fourth objective is considered a very important objective that previous works on container optimization did not consider. To ensure on-demand usage of the applications running in the containers, it is important to use a scheduler that is not only efficient but also fast to reallocate the containers to different nodes depending on the objectives without taking too much time that can affect the performance of the software, Thus a scheduler that does the minimal needed number of changes to balance the load between the nodes and respect the resources constraints of each of them is the one that we worked on in this project.

To this end, we compare the Swarm State $P^{(t)}$ and the solution S aiming to count the number of changes required to move a container to another node. The objective is defined as follows:

$$CHG(S) = \frac{hamming(S,P)}{|P|} \tag{4.4}$$

where $hamming(S,P)$ is the hamming distance between P and S and $|P| = |S|$. If $|P| = 0$, then $CHG(S) = 0$.

Therefore, the goal of our proposed approach is the following:

$$\underset{S}{\text{minimize}} \quad NON(S), FRQ(S), COP(S), CHG(S) \tag{4.5}$$

where all objective functions are normalized in the range [0, 1] where 0 is the best value and 1 the worst one.

To clarify how the objective functions are computed, consider $C = \{0,1,2,3,4\}$ as the set of available containers, $N = \{1,2,3,4\}$ as the set of available nodes, $P^{(t)} = \{3,3,3,1,2\}$ as the current Swarm State and G as the dependency graph available in Figure 4.2.

Now, consider that a solution $S = \{3,3,1,2,1\}$ (the same available in Figure 4.3) was generated by an optimization algorithm to be evaluated, the objective functions are calculated as follows:

$$\begin{aligned} NON(S) &= \frac{3}{4} = 0.75 \\ FRQ(S) &= \sqrt{\frac{0.10}{3}} = 0.18 \\ COP(S) &= \frac{4}{6} = 0.66 \\ CHG(S) &= \frac{3}{5} = 0.60 \end{aligned} \tag{4.6}$$

Therefore, the objective values are S (0.75, 0.18, 0.66, 0.6).

4.2 Empirical Study

To evaluate our approach for software container scheduling using NSGA-III, we conducted a set of experiments based on 48 containers. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed with the aim of comparing our NSGA-III proposal with a variety of existing approaches. In this section, we first present our research questions and then describe and discuss the obtained results. Finally, we discuss the various threats to the validity of our experiments.

4.2.1 Research Questions

In our study, we assess the performance of our approach by finding out whether it could generate meaningful scheduling solutions for the software containers that improve the usage of resources. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. We define in the following the two main research questions that we are addressing:

RQ1. To what extent can the proposed NSGA-III approach provide efficient scheduling solutions based on different multi-objective (NSGA-II) and many-objective algorithms (IBEA)? This question aims to investigate the efficiency of our many-objective NSGA-III approach for container scheduling to find trade-offs between the different conflicting objectives compared to other multi/many-objective algorithms.

RQ2. To what extent can the proposed NSGA-III approach minimize the resources consumption in the cluster and balance the software workload (i.e., CPU and memory usage, the network I/O of each node) compared to the deterministic Docker Swarm's default scheduler? Since it is not sufficient to validate the outperformance of our approach compared to other search-based algorithms, this question evaluates the ability of our approach compared to the deterministic by default

scheduler of the Docker Swarm in terms of the resources consumption (i.e., CPU and memory usage).

To answer **RQ1**, we considered the widely-used quality indicators in multi-objective optimization (described in subsection 4.2.2) to evaluate the different search algorithms such as Hypervolume (HV), Inverted Generation Distance (IGD), Contributions (IC). These metrics validate the quality, spread, and diversity of the generated scheduling solutions on the Pareto front. Thus, we can determine which search algorithm performs better to find the best trade-offs between the conflicting scheduling objectives. Furthermore, we have also considered the execution time to compare the different algorithms since we are considering a large number of objectives. We did not compare our algorithm to random search as it is evident that the space to explore is too large, requiring an intelligent search. We have also did not compare with mono-objective search (aggregating all the objectives into one fitness function) as it is evident that the different objectives are conflicting: In a cluster where containers are connected to each other, minimizing the coupling between them will automatically increase the number of containers per node; and if we aim to decrease the number of containers per node we will automatically increase the number of selected nodes. Thus, we aimed in this research question to focus only on comparing our NSGA-III adaption and two other algorithms: IBEA and NSGA-II. We selected NSGA-II to evaluate its performance with a larger number of objectives than two, which may justify the need to use many-objective algorithms. We have also selected IBEA as it is known to be widely used in the current many-objective optimization literature after NSGA-III. We used the same adaptation for all three algorithms to enable a fair comparison.

We believe that the quality metrics results discussed in item **RQ1** would affect the results regarding resource consumption. The algorithm giving the best set of solutions will be able to give the best resource consumption compared to the other algorithms. Therefore,

we only compared the default scheduler with the NSGA-III algorithm for resource consumption.

As a result, to answer **RQ2**, we compared the results from the default Docker Swarm's scheduling algorithm against the best search algorithm from item **RQ1**,

by considering ApacheBench [94] as a stress testing tool. Using ApacheBench, we set a total of 100000 requests that should be made when running each project and 100 requests concurrently (simultaneously) at a time, ensuring scalable testing settings. We also used Node-exporter [95] and Cadvisor [96] to monitor the status and performance of each working node in the experiment, including CPU and memory usage and network I/O. We used these evaluation metrics instead of the objective functions to avoid any bias when comparing the search-based and deterministic techniques. We have also created an online appendix for a demo of our platform, and related experiments material [87].

4.2.2 Quality Indicators

Aiming to compare the search-based algorithms, we considered the following sets of solutions [97]: i) PF_{approx}: set of non-dominated solutions obtained by one algorithm execution; ii) PF_{known}: set of non-dominated solutions of an algorithm obtained by the union of all the PF_{approx} from all the executions, removing the non-dominated and repeated solutions; and iii) PF_{true}: formed by all sets PF_{known} obtained from different algorithms by removing dominated solutions and repeated ones. The analysis was conducted by using the widely used quality indicators in the computational search field to evaluate both the quality and spread/diversity of the solutions:

- **Hypervolume** (HV) [98] measures the volume covered by members of a Pareto-front in objective space delimited by a reference point. An important feature of this metric is its ability to capture the diversity and convergence of solutions. A higher hypervolume value is desirable.

- **Inverted Generational Distance** (IGD) [99] is a convergence measure that corresponds to the average Euclidean distance between the approximate Pareto-front provided by an algorithm and the reference Pareto-front. Small values are desirable.

- **Contributions** (IC) [100] measures the proportion of solutions that lie on the reference front (RS) [101]. The higher this proportion, the better the quality of solutions.

4.2.3 Studied Docker Projects

In the experiments, we selected 48 Docker-based projects available on GitHub. Table 4.1 provides some descriptive statistics about all of them, such as the number of stars, contributors, services, and containers. We selected these projects for our validation because they range from medium to large-sized open-source projects, which were actively developed over the past 10 years, they are widely used. They are based on several programming languages. Regarding the number of containers, the figure shows that the smallest project has two containers (e.g., RAMMYGIT/MEWBASE), and the largest one has eleven containers to be scheduled (e.g., MARINANIEROD/DOCKER_PRESTASHOP). Furthermore, the list of projects contains containers coded in several programming languages such as JavaScript, Python, Ruby, PHP, etc.

4.2.4 Parameter Settings

The parameter setting significantly influences the performance of a search algorithm on a particular problem. For this reason, for each multi/many-objective algorithm and for each project, we perform a set of experiments using several population sizes [102, 103, 104]. Each algorithm is executed 30 times with each configuration, and then the comparison between the configurations is done based on IGD using the Wilcoxon test. In order to have significant results for each couple (algorithm, project), we use the trial and error method to

Table 4.1: Docker-based projects studied.

#	Name	Size (Kb)	Star	Contrib.	# of Serv.	# of Cont.
1	vegasbrianc/prometheus	3133	2.9k	33	5	5
2	Zappelphilipp/docker-graylog-kibana-nginx-stack	23	2	1	5	5
3	joelanman/e-petitions	7.9k	0	38	7	7
4	Semprini/cbe-utilities	64	1	2	5	5
5	pjuu/pjuu	5.1k	55	7	5	5
6	blacktop/docker-bro	79.7k	124	3	3	3
7	miso-lims/miso-lims	129.3k	127	14	4	4
8	Screenly/screenly-ose	19.2k	1.1k	50	5	5
9	hamburml/docker-flow-letsencrypt	80	94	8	3	4
10	LaVestima/hanna-agency	5.2k	0	2	6	6
11	ucan-lab/docker-laravel	400	360	7	3	3
12	FedorSelitsky/eventrack	1.5k	5	3	5	5
13	MassDistributionMedia/rc-ca-blinds	43.4k	2	113	3	3
14	stencila/hub	28.1k	23	13	11	11
15	dockerwest/compose-magento	77	2	3	6	6
16	dotnet-architecture/eShopModernizing	133.6k	706	11	4	4
17	sameersbn/docker-gitlab	5.9k	6.6k	167	4	4
18	hcxp/hcxp	576	5	1	6	6
19	changami/froghouse-lightning-talk-docker	10k	0	1	3	3
20	Hygieia/Hygieia	75.6k	3.5k	140	4	4
21	bartTC/dpaste	1.9k	304	22	3	3
22	znly/docker-druid	13	25	2	7	7
23	benjefferies/gogo-garage-opener	16.1k	35	3	3	3
24	Merrick28/delain	48.5k	8	7	2	2
25	Djacket/djacket	352	60	1	3	3
26	p6spy/p6spy	59.6k	1.3k	27	5	5
27	memodir/cv	2.7k	0	2	3	3
28	marinanierod/docker_prestashop	79	2	4	11	11
29	jfrog/artifactory-docker-examples	2.7k	291	30	2	2
30	zentby/dockerspace	22	0	2	3	3
31	zengoma/docker-magento2-apache-dev	25	1	1	2	2
32	envato/double_entry	722	267	19	3	3
33	camd67/moebot	1.2k	1	3	3	3
34	eclectic-boy/rhodonea_mapper	62	0	2	2	2
35	shin-/sakuya-blog	49	2	1	2	2
36	zebresel-com/mongodm	73	179	3	2	2
37	josephspurrier/gowebapi	648	55	1	2	2
38	RailsEventStore/rails_event_store	8.3k	923	58	4	4
39	labpositiva/pyworkplace	5k	1	2	2	2
40	busino/dj_farm_example	30	0	1	3	3
41	azerothcore/azerothcore-wotlk	359.6k	1.2k	152	3	3
42	phillc/rosterbater	503	3	1	4	4
43	vipulasa/laravel-meetup-v.2.0	201	1	1	2	2
44	tootsuite/mastodon	115.2k	23.5k	416	4	4
45	rodrigogs/github-metrics	295	6	2	2	2
46	netresearch/timetracker	42.7k	8	5	3	3
47	chiefy/terraform-linode-nextcloud	29	0	1	9	9
48	rammygit/mewbase	500	0	5	2	2

obtain a good parameter configuration. Since we compare different search algorithms, we classify the parameters into common parameters and specific parameters. If the results are similar for a given combination of parameters, the execution time was considered. As evolutionary operators, we adopted Integer SBX crossover, Integer Polynomial mutation, and binary tournament for selecting the individuals [105] because they have been designed to work with integer solutions. Therefore, the list of selected parameters used to answer the stated research questions is described in Table 5.2.

Table 4.2: Parameter Settings.

Parameter	**NSGA-III**	**NSGA-II**	**IBEA**
Population Size	200	200	100
Maximum Number of Generations	2500	2500	2500
Crossover Probability	0.95	0.9	0.9
Mutation Probability	0.05	0.01	0.01
Crossover Operator	Integer SBX		
Mutation Operator	Integer Polynomial		

As we have also measured the execution time, the algorithms were executed in a machine with an Intel(R), Core(TM) i7-5930K, CPU 3.50GHz with 40Gb RAM.

4.2.5 Statistical Tests

Since meta-heuristic algorithms are stochastic ones, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 30 independent simulation runs for each problem instance. The obtained results are statistically analyzed using the Wilcoxon rank-sum test [106] with a 99% confidence level (α=1%). This non-parametric test is used to evaluate the null hypothesis (H) that the results from the two algorithms come from continuous distributions with equal medians against the alternative hypothesis H1 that the medians differ.

The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value less than or equal to α (<=0.01) means that we accept H1 and reject H0. However, a p-value that is strictly greater than α (>0.01) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing NSGA-II and IBEA search results with NSGA-III ones. In this way, we determine whether the performance difference between NSGA-III and one of the other approaches is statistically significant or just a random result.

4.2.6 Results

4.2.6.1 Results for RQ1

Table 4.3 and Table 4.7 summarize the results of mean values and standard deviations for HV, IGD, and IC indicators over 30 independent simulation runs, where the bold values represent the best ones. The results are based on the consideration of all 4 objectives for the evolutionary algorithms. The objectives values were normalized between 0 and 1 and set to be minimized; the order of the objectives is not important and has no impact on the results. The users can select the best solution based on their preferences (fitness function values) and programming behavior from the non-dominated (trade-off) set of solutions. All the results were statistically significant on the 30 independent simulations using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

When comparing NSGA-III against NSGA-II and IBEA using all three performance indicators, it is clear that NSGA-II has the weakest performance. On small-scale docker projects including up to 4 containers (e.g., docker-bro, miso-lims, docker-flow-letsencrypt, docker-laravel, rc-ca-blinds) all algorithms present similar results for IGD, HV, and IC. For example, for the rc-ca-blinds, both algorithms give the best results in terms of the three quality metrics (0 for the IGD, 0.065 for the HV, and 1 for the IC). On medium-scale docker projects with up to 7 containers (e.g., e-petitions, compose-magento, hcxp/hcxp), NSGA-III and IBEA present similar results, and both provide better results than NSGA-II. For

Table 4.3: HV, IGD, and IC mean values with NSGA-III, NSGA-II, and IBEA. The results were statistically significant on 30 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%)

Docker Project	NSGAIII			IBEA			NSGAII		
	IGD	HV	IC	IGD	HV	IC	IGD	HV	IC
terraform-linode-nextcloud	**0.028 ± 0.013**	0.123 ± 0.013	**0.942 ± 0.013**	0.051 ± 0.010	**0.124 ± 0.010**	0.858 ± 0.010	0.089 ± 0.013	0.123 ± 0.013	0.725 ± 0.013
docker-grocery	**0.007 ± 0.000**	**0.150 ± 0.000**	**0.987 ± 0.000**	0.013 ± 0.000	**0.150 ± 0.000**	0.980 ± 0.000	0.141 ± 0.000	0.150 ± 0.000	0.800 ± 0.000
cbe-utilities	**0.000 ± 0.000**	**0.063 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.063 ± 0.000**	**1.000 ± 0.000**	0.118 ± 0.000	0.031 ± 0.000	0.500 ± 0.000
docker-bro	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
miso-lims	**0.000 ± 0.000**	**0.125 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.125 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.125 ± 0.000**	**1.000 ± 0.000**
screenly-ose	**0.000 ± 0.000**	**0.111 ± 0.000**	**1.000 ± 0.000**	0.008 ± 0.000	**0.111 ± 0.000**	0.987 ± 0.000	0.128 ± 0.000	0.111 ± 0.000	0.793 ± 0.000
pju	**0.000 ± 0.000**	**0.089 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.089 ± 0.000**	**1.000 ± 0.000**	0.098 ± 0.000	0.044 ± 0.000	0.625 ± 0.000
docker-flow-letsencrypt	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
hanna-agency	**0.000 ± 0.000**	**0.150 ± 0.000**	**0.987 ± 0.000**	0.013 ± 0.000	0.150 ± 0.000	0.980 ± 0.000	0.141 ± 0.000	0.150 ± 0.000	0.800 ± 0.000
stencila/hub	**0.023 ± 0.001**	**0.144 ± 0.001**	**0.644 ± 0.001**	**0.021 ± 0.007**	0.138 ± 0.007	0.612 ± 0.007	0.054 ± 0.020	0.108 ± 0.020	0.355 ± 0.020
eventrack	**0.000 ± 0.000**	**0.074 ± 0.000**	**1.000 ± 0.000**	0.002 ± 0.000	**0.074 ± 0.000**	0.993 ± 0.000	0.176 ± 0.000	**0.074 ± 0.000**	0.600 ± 0.000
docker-laravel	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
rc-ca-blinds	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
compose-magento	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
eShopModernizing	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
docker-gitlab	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
hcxp/hcxp	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
froghouse-lightning-talk	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
Hygieia/Hygieia	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
bartTC/dpaste	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
znly/docker-druid	**0.000 ± 0.000**	**0.146 ± 0.000**	**1.000 ± 0.000**	0.004 ± 0.004	0.144 ± 0.004	0.978 ± 0.004	0.103 ± 0.024	0.135 ± 0.024	0.806 ± 0.024
gogo-garage-opener	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
delain	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
Djacket/djacket	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000

Table 4.4: HV, IGD, and IC mean values with NSGA-III, NSGA-II, and IBEA. The results were statistically significant on 30 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%)

Docker Project	NSGAIII			IBEA			NSGAII		
	IGD	HV	IC	IGD	HV	IC	IGD	HV	IC
p6spy/p6spy	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
memodir/cv	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
docker_prestashop	**0.091 ± 0.000**	**0.210 ± 0.000**	**0.696 ± 0.000**	**0.084 ± 0.000**	**0.210 ± 0.000**	0.654 ± 0.000	0.108 ± 0.000	0.210 ± 0.000	0.583 ± 0.000
artifactory-docker	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
magento2-apache-dev	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
double_entry	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
camd67/moebot	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
rhodonea_mapper	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
dockerspace	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
sakuya-blog	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
mongodm	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
gowebapi	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
rails_event_store	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
pyworkplace	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
azerothcore-wotlk	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
dj_farm_example	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
rosterbater	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
laravel-meetup-v.2.0	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
mastodon	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
timetracker	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
graylog-kibana	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.100 ± 0.000	**0.078 ± 0.000**	0.714 ± 0.000
e-petitions	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.095 ± 0.000	**0.078 ± 0.000**	0.714 ± 0.000
vegasbrianc/prometheus	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
github-metrics	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**

hcxp/hcxp project, NSGA-III and IBEA output as results 0, 0.078, and 1 for the IGD, HV, and IC metrics compared to 0.056, 0.714, and 0.833 for NSGA-II respectively with the same metrics. Furthermore, the project compose-magento shows that both many objectives algorithms output 0 for the IGD, 0.078 for the HV, and 1 for the IC compared to 0.056, 0.071, and 0.833, respectively, when NSGA-II is used. For large-scale docker projects, NSGA-III is significantly better than NSGA-II and IBEA on most projects with a large number of containers (e.g., hanna-agency, Terraform-linode-nextcloud, docker-prestashop, and stencila/hub). Considering the example of the project "znly/docker-druid", NSGA-III outperformed both other algorithms by providing as results 0, 0.146, and 1 for IGD, HV, and IC compared to 0.004, 0.144, 0.978, and 0.103, 0.135 AND 0.806 respectively for both IBEA, NSGAII. This outcome is consistent with existing studies in other domains where NSGA-II is not able to handle more than 2-3 objectives.

For most of the test results, IBEA evaluation was consistent with the NSGAII algorithm and presented similar results, whereas, for the other docker projects, NSGA-III outperformed both algorithms. This could be explained by the interaction between (1) Pareto dominance-based selection and (2) reference point-based selection, which is the distinguishing feature of NSGA-III compared to other existing many-objective algorithms. For a better comparison between NSGA-III, and IBEA, since they showed similar results in different projects, we studied the execution time of all many/multi-objective algorithms used in our experiments. The execution time is critical when using evolutionary algorithms. This metric is important to compare the algorithms regarding the spread of identifying scheduling solutions. It is important to give not only efficient scheduling of the containers but also a fast and smooth reallocation required for normal behavior of the applications deployed in the containers when rescheduling.

Table 4.5 and Table 4.6 show the average running times of the different algorithms, over 30 runs, on the different projects used in our experiments. It is clear that NSGA-III is

the fastest, on average, compared to NSGAII and IBEA. For hcxp/hcxp project, NSGA-III output ran in 122 seconds compared to 145 seconds for 145 and 133 respectively, for IBEA and NSGA-II. Furthermore, the project compose-magento shows the out-performance of NSGA-II with 107 seconds compared to 145 seconds for 152 and 143, respectively, for IBEA and NSGA-II. Also, NSGA-III is significantly better than NSGA-II and IBEA on most projects with a large number of containers (e.g., hanna-agency, Terraform-linode-nextcloud, docker-prestashop, and stencila/hub). Considering the example of the project "znly/docker-druid", NSGA-III outperformed both other algorithms by providing results 92 seconds compared to 126 and 132 seconds, respectively, for both IBEA and NSGAII.

This observation could be explained by the computational effort required to compute each solution's contribution (IGD) when using IBEA. Furthermore, NSGA-II may take longer time to find relevant solutions than many-objective algorithms due to the limited spread of the solutions in the Pareto front when using more than 3 objectives. We note that the experiments were conducted on a single machine (i7 – 2.70 GHz, 8.0 GB – DDR3, SSD - 520MB/s); thus, the different algorithms will run faster on better hardware configurations.

Key findings: NSGA-III outperforms the different search algorithms based on NSGA-II and IBEA regarding the quality and spread of identified scheduling solutions in the Pareto front.

4.2.6.2 Results for RQ2 In this research question, we compare the NSGA-III results against the Docker Swarm's default scheduler on 30 independent runs.

We used the first three objectives NON, FRQ, and COP. CHG is not considered in this experiment because CHG corresponds to the number of changes required to reschedule the containers from a swarm state P generated by the default scheduler to a new swarm state generated by our tool. We believe that it is important to consider only the required changes when moving containers between nodes. Thus, we find the best and fastest solution that

Table 4.5: Average Computational time values on 30 independent runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).

Project	NSGAII	IBEA	NSGAIII
hcxp/hcxp	122	145	133
docker-grocery	122	145	133
delain	117	127	131
double_entry	117	127	131
graylog-kibana	116	123	124
screenly-ose	111	117	138
rc-ca-blinds	111	129	147
compose-magento	107	152	143
hanna-agency	107	152	143
sakuya-blog	103	126	128
docker-gitlab	103	126	128
docker_prestashop	103	144	167
dockerspace	103	124	167
mongodm	103	144	167
gowebapi	103	126	128
rails_event_store	103	144	167
cbe-utilities	103	126	128
laravel-meetup-v.2.0	103	144	137
e-petitions	103	144	137
froghouse-lightning-talk	103	126	128
bartTC/dpaste	102	131	142
screenly-ose	102	131	142
pyworkplace	101	123	123
rosterbater	101	128	137
mastodon	101	123	114
eventrack	99	105	124
pju	99	104	143
Djacket/djacket	99	141	153
camd67/moebot	99	141	153
docker-laravel	98	120	167
eShopModernizing	98	120	167
artifactory-docker	98	123	124
azerothcore-wotlk	97	108	124
miso-lims	97	121	140
magento2-apache-dev	97	108	143
timetracker	96	108	133

Table 4.6: Average Computational time values on 30 independent runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).-Continued

Project	NSGAII	IBEA	NSGAIII
p6spy/p6spy	93	121	138
docker-bro	93	104	124
Hygieia/Hygieia	93	104	124
rhodonea_mapper	93	121	138
pju	92	126	132
znly/docker-druid	92	126	132
memodir/cv	91	94	146
docker-flow-letsencrypt	84	105	165
stencila/hub	84	105	165
gogo-garage-opener	84	105	165
github-metrics	83	124	117
vegasbrianc/prometheus	81	123	114
dj_farm_example	11	123	143

reallocates the containers for better resource usage while keeping the normal behavior of the applications deployed in the containers. However, we cannot compare the docker default scheduler with our new scheduler using this metric since we are calculating these changes when moving from the default state presented by the docker.

The default Docker Swarm's scheduler is based on a deterministic adhoc approach based on filters and strategies. Filters are used to narrow the domain of nodes for scheduling by taking the node and container properties as inputs, among other parameters. Strategies are used to decide on which node the next container runs using three alternatives: random, spread, and binpack.

As described in Table 4.7, our proposed approach provides significant improvements in terms of the number of selected nodes (NON), the average number of containers per node (FRQ), and the node's coupling values (COP) compared to Docker Swarm's default scheduler. This is an interesting result confirming that NSGA-III can find very good

compromises between the different conflicting objectives and outperform those produced by the Docker default scheduler. In some cases, applying NSGA-III solutions give slightly higher values for the average number of containers per node (e.g., docker-grocer, cbe-utilities, screenly-ose, pju and examples shown in Table 4.7). This can be explained by the fact that decreasing the number of depending containers allocated on different nodes and decreasing the number of selected nodes will automatically increase the number of containers per node. Overall, the NSGA-III algorithm was able to find a good trade-off between all four objectives since most of them were significantly decreased comparing the initial state of the cluster before rescheduling.

Since comparing the performance of NSGA-III with the default scheduler using similar evaluation metrics to the fitness functions is not sufficient, we considered three evaluation metrics in terms of CPU usage, memory usage, and network transmission. For this purpose, we selected ten projects with different sizes and numbers of containers in a cluster composed of 3 nodes. The results described in Table 4.8 provide for every project the % of CPU usage, memory usage (average of the resources consumption for the three nodes), and the network usage (received and transmitted values in bytes per second) in the cluster using NSGA-III versus the default Docker swarm scheduler. Table 4.9 describes also in more details the % of resource consumption per node.

We found interesting results as described in Table 4.8, including a decrease in all resource usage for almost all projects using our approach compared to the Docker Swarm scheduler. Considering the example of project "cbe-utilities", the percentage of CPU usage for the cluster decreased from 16.89% to 13.90%, and the decrease in the memory usage exceeded 6% (from 40.3% to 33.90%) after applying our new many-objective approach. Further details in Table 4.9 about the results per node are described. We notice that the CPU usage for both node 2 and node 3 decreased from 14.860% and 15.130% to 11.5% and 5.650%, respectively, with a little increase within node 1 which can be explained by an

Table 4.7: Number of selected nodes (NON), Average number of containers per nodes (FRQ), number of coupling (COP) mean values of NSGA-III over 30 independent simulation runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

Docker Projects	**Our Approach**			**Default Scheduler**		
	NON(S)	FRQ(S)	COP(S)	NON(S)	FRQ(S)	COP(S)
terraform-linode-nextcloud	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
docker-grocery	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
cbe-utilities	0.66666	0.09999	0.33333	1.00000	0.09428	0.83333
docker-bro	0.66666	0.09999	0.80000	1.00000	0.09428	1.00000
miso-lims	0.33333	0.00000	0.00000	1.00000	0.00000	1.00000
screenly-ose	0.66666	0.00000	0.33333	1.00000	0.11785	0.66666
pju	1.00000	0.13608	0.71428	1.00000	0.00000	0.85714
docker-flow-letsencrypt	1.00000	0.09428	0.60000	1.00000	0.09428	1.00000
hanna-agency	0.66666	0.16666	0.00000	1.00000	0.00000	0.00000
docker-laravel	1.00000	0.06734	0.25000	1.00000	0.06734	0.50000
rc-ca-blinds	0.66666	0.16666	0.00000	1.00000	0.00000	0.00000
eventrack	0.33333	0.00000	0.00000	1.00000	0.00000	1.00000
stencila/hub	0.66666	0.09999	0.40000	1.00000	0.09428	0.60000
compose-magento	1.00000	0.04285	0.46153	1.00000	0.08570	0.53846
eShopModernizing	0.66666	0.07142	0.00000	1.00000	0.06734	0.00000
docker-gitlab	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
hcxp/hcxp	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
froghouse-lightning-talk	1.00000	0.18856	0.00000	1.00000	0.09428	0.50000
Hygieia/Hygieia	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
bartTC/dpaste	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
znly/docker-druid	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
gogo-garage-opener	0.66666	0.00000	0.00000	1.00000	0.00000	0.00000
delain	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
Djacket/djacket	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
p6spy/p6spy	0.33333	0.00000	0.00000	1.00000	0.11785	0.00000
memodir/cv	1.00000	0.09428	0.00000	1.00000	0.09428	0.00000
docker_prestashop	1.00000	0.11785	0.00000	1.00000	0.11785	0.00000
artifactory-docker	1.00000	0.08570	0.10000	1.00000	0.04285	0.30000
magento2-apache-dev	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
double_entry	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
camd67/moebot	0.66666	0.00000	0.00000	1.00000	0.11785	0.50000
rhodonea_mapper	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
dockerspace	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
sakuya-blog	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
mongodm	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
gowebapi	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
rails_event_store	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
pyworkplace	0.33333	0.00000	0.00000	0.66666	0.00000	0.00000
azerothcore-wotlk	0.66666	0.00000	0.66666	1.00000	0.11785	0.66666
dj_farm_example	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
rosterbater	0.66666	0.04545	0.00000	1.00000	0.04285	0.00000
laravel-meetup-v.2.0	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
mastodon	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
timetracker	1.00000	0.11785	0.00000	1.00000	0.11785	1.00000
graylog-kibana	0.33333	0.00000	0.00000	1.00000	0.09428	1.00000
e-petitions	0.33333	0.00000	0.00000	1.00000	0.09428	1.00000
vegasbrianc/prometheus	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
github-metrics	1.00000	0.00000	0.33333	1.00000	0.00000	0.50000

Table 4.8: Average of CPU, Memory, and Network usages comparing our approach and Spread in the cluster The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

Docker Project	CPU Usage		Memory Usage		Network Usage (bytes/s)			
					Received		Transmitted	
	OurAppr.	Spread	OurAppr.	Spread	OurAppr.	Spread	OurAppr.	Spread
graylog-kibana	**10.15%**	10.60%	**43.36%**	50.09%	**1164.92**	1603.96	**1161.17**	1652.10
e-petitions	**6.88%**	9.22%	**47.63%**	48.35%	**846.70**	1294.86	**1160.09**	1340.18
cbe-utilities	**13.90%**	16.89%	**33.90%**	40.30%	**1179.04**	1539.35	**1117.37**	1661.11
docker-bro	**14.97%**	17.65%	**44.31%**	46.63%	**1172.26**	1610.29	**1162.84**	1415.64
miso-lims	**2.34%**	2.38%	**15.36%**	20.60%	**521.93**	940.86	**102.54**	340.15
screenly-ose	**6.77%**	7.68%	**29.39%**	29.79%	**1099.01**	1505.93	**1334.49**	1429.68
pju	**20.18%**	20.78%	**36.84%**	42.53%	**719.01**	1459.63	**895.31**	1424.31
docker-flow-letsencrypt	**14.97%**	17.65%	**44.31%**	46.63%	**664.90**	1006.96	**720.01**	868.74
hanna-agency	**2.31%**	2.38%	**15.36%**	20.69%	**1319.59**	2158.03	**1386.06**	1975.92
docker-laravel	**20.60%**	23.65%	**45.63%**	52.50%	**664.90**	1006.96	**720.01**	868.74

increase in the number of containers in node1. Regarding memory usage, we notice a decrease for node 3 to half (from 40.5% to 20.210%) while keeping approximately the same memory usage for both other nodes. Also, in other projects such as docker-flow-letsencrypt, e-petitions, and docker-laravel, we notice a decrease of approximately 3% in the CPU usage for the hole cluster when using our approach and a memory decrease that exceeds 2%, 2%, and 7% respectively.

The results for each node for these projects detailed in Table 4.9 are promising. Continuing with the example docker-flow-letsencrypt, the CPU usage for both node1 and node2 was decreased from 15.620% and 19.520% to 13.140% and 10,260%, respectively, and the memory usage for node 1 was decreased from 55.23%51.850 to 55.230%. Also, for e-petition, we observe a decrease that exceeded 9% for node 2 in terms of CPU (from 11.930% to 2.7% and from 4.6 to 2.070% for node 3) and approximately the same average of memory consumption for both nodes. The balance in our approach between decreasing the number of nodes and the average of containers per node succeeded in distributing equally

Table 4.9: CPU, Memory, and Network usages comparing our approach and Spread for every node. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

Docker Project	Nodes	CPU Usage		Memory Usage		Network Usage (bytes/s)			
						Received		Transmitted	
		Our Appr.	Spread	Our Appr.	Spread	Our Appr.	Spread	Our Appr.	Spread
graylog-kibana	1	19.700%	15.140%	53.390%	51.090%	1537.520	2632.610	3366.580	4830.660
	2	5.700%	11.930%	40.850%	40.500%	997.570	1228.150	59.120	60.610
	3	5.070%	4.600%	35.830%	40.500%	959.680	960.120	57.820	65.050
e-petitions	1	15.890%	11.140%	62.090%	62.390%	1542.520	1752.320	3366.580	3892.660
	2	2.700%	11.930%	40.500%	41.850%	997.570	1260.150	59.850	62.420
	3	2.070%	4.600%	40.300%	40.830%	658.680	872.120	53.840	65.444
cbe-utilities	1	24.560%	20.660%	41.560%	40.470%	1609.970	1832.780	3233.000	4830.660
	2	11.500%	14.860%	40.120%	40.500%	966.870	1825.150	58.660	91.610
	3	5.650%	15.130%	20.210%	40.621%	960.290	960.120	60.442	61.050
docker-bro	1	21.520%	17.820%	51.850%	55.230%	1456.520	2242.61	3366.580	4120.66
	2	13.140%	15.620%	40.500%	41.850%	1100.570	1628.15	59.120	63.610
	3	10.260%	19.520%	40.600%	42.830%	959.680	960.12	62.820	62.650
miso-lims	1	5.454%	2.200%	25.090%	21.390%	809.720	1100.320	210.970	892.660
	2	0.256%	2.500%	11.500%	19.850%	97.410	960.150	35.450	62.780
	3	1.233%	2.450%	9.500%	20.830%	658.68	612.250	62.200	65.000
screenly-ose	1	11.500%	11.620%	38.090%	38.500%	1409.970	1632.520	3830.660	4125.650
	2	10.700%	5.820%	37.000%	36.850%	1266.780	1925.150	99.800	75.125
	3	1.120%	5.600%	5.300%	12.830%	620.290	960.120	73.000	88.254
pju	1	24.820%	21.337%	41.120%	40.470%	1229.520	1856.610	2563.000	4120.660
	2	20.320%	21.023%	25.200%	42.500%	1100.570	1562.150	60.120	89.610
	3	17.200%	20.130%	44.200%	44.620%	450.680	960.120	62.820	62.650
docker-flow-letsencrypt	1	21.520%	17.820%	51.850%	55.230%	537.320	1132.610	2100.100	2530.560
	2	13.140%	15.620%	40.500%	41.850%	937.700	1228.150	32.120	40.610
	3	10.260%	19.520%	40.600%	42.830%	459.680	660.120	27.820	35.050
hanna-agency	1	5.454%	2.200%	25.090%	21.390%	2001.520	3500.810	3366.580	4730.660
	2	0.256%	2.500%	11.500%	19.850%	997.570	1428.150	478.120	650.610
	3	1.233%	2.450%	9.500%	20.830%	959.680	1545.120	313.487	546.500
docker-laravel	1	26.560%	25.230%	66.360%	62.500%	537.320	1132.610	2100.100	2530.560
	2	12.450%	24.180%	25.250%	47.000%	937.700	1228.150	32.120	40.610
	3	22.780%	21.550%	45.300%	48.000%	459.680	660.120	27.820	35.050

the containers achieving a balanced load between nodes that leads to reduced resource usage and more efficient utilization of these resources in the whole cluster. For a few projects (e.g., graylog-kibana, miso-lims, hanna-agency), the results show unremarkable differences between the two approaches in terms of CPU. These results can be explained by the low number of containers used in such projects. Thus, we obtained a very similar allocation of nodes for both approaches that do not affect resource usage by keeping almost the same number of containers per node.

We have noticed a significant decrease in network transmission for all projects using our approach compared to the Docker swarm scheduler. For instance, the project

"cbe-utilities" has an average of received and transmitted network in the cluster that decreased from 1539.350 and 1661.106 bytes per second while the docker strategy provided values of 1179.043 and 1117.367 (bytes per second). Table 4.9 summarizes the results for each node separately. For the project graylog-kibana, for example, the received network was decreased from 2632.610, 1228.150, and 960.120 for node 1, node 2, and node 3 using the default scheduler to 1537.520, 997.570, 959.680 respectively using our approach. The Transmitted Network was decreased from 4830.660, 60.610, and 65.050 to 4830.660, 60.610, and 65.050 for each node, respectively (node 1, node 2, and node 3). The decrease in the network I/O values is explained by the fact that we are decreasing the container coupling in different nodes for better communication between nodes.

Key findings: Our new many-objective approach outperforms the docker default scheduler regarding resource usage: the CPU, memory, and network transmission in the docker cluster.

4.3 Threats to Validity

Conclusion Validity. Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 99% confidence level (α <1%). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the search-based software engineering community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [107] for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Construct Validity. Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solutions at the knee point when we compared our approach with fully-automated scheduling approaches, but the users may select a different scheduling solution based on their preferences to give different weights to the objectives when selecting the best solution. To mitigate this threat, we have to use the quality indicators of the Pareto fronts when comparing the different search algorithms and also the average values of the resource usage metrics.

External Validity. We selected such Docker projects because they are developed considering several programming languages and have been developed in the past 10 years. However, we cannot state that this is enough to generalize the results since the site may not reflect real-world projects. To minimize such threats, we tried to evaluate docker projects from different domains and sizes. The use of larger docker projects should be evaluated in a future experiment. Another threat is related to the generalizability of our findings.

Another threat is that we did not include existing approaches, including meta-heuristic algorithms and deep learning models, in the validation because they use assumptions different from ours. For instance, deep learning models require a very large dataset that is not available in practice. The other search algorithms use fewer objectives. Thus, they were considered as part of the multiple formulations that we proposed in our benchmark. We highlighted the lack of existing tools for container scheduling beyond Kubernetes and a few other commercial tools. None of the existing approaches provided their tools for the community, and they are very hard to replicate.

4.4 Conclusion

In this chapter, we proposed a new dynamic workload balancing for containers that target more complex objectives than the typical default scheduler of existing Docker

technologies. Our new approach aimed to achieve a balanced workload between the clusterś nodes and more efficient utilization of resources. Therefore, we considered in our approach to minimize the number of selected nodes to reduce resources consumption, minimize the average of containers per node for a balanced workload between them, taking into consideration the dependencies between containers and try to reduce the coupling (dependent containers allocated to different nodes) to minimize the network transmission and finally minimize the number of changes between the current scheduling and our approach (e.g., move container). The experiments performed on 48 docker projects provides strong evidence that our approach can significantly reduce resource consumption (CPU, memory, network I/O) compared with the default scheduler and other existing techniques.

CHAPTER FIVE

EFFICENT MANAGEMENT OF CONTAINERS FOR SOFTWARE DEFINED VEHICLES

5.0 Introduction

Docker, the leading containerization framework, is utilized by 79% of IT organizations [49]. Consequently, there is a noticeable trend toward containerizing software applications in diverse domains, with the automotive industry being no exception [108, 109]. The importance of software in today's vehicles has increased tenfold and it plays a prominent role in the development of SDVs. Recent advances in connected and smart cars have significantly increased the complexity of the software applications running in the car [110, 111, 112]. Smart and connected vehicles [113, 114, 115, 116] include many applications to enable different features such as driving assistance (e.g., ADAS [117, 118]), intelligent path planning (e.g., ITS [119]), monitoring of the cameras and sensors, media, intelligent speed control, detecting traffic signals, and so on. With the current shift in the automotive industry towards a software-defined based model [120], it is critical to enable cars to host and execute a large number of applications with limited resources (number of ECUs) with limited memory, CPU, and energy) [121]. Recent trends in automotive are focusing on the containerization of the software applications running in cars [89, 90, 91]. Consequently, the efficient management of these containers is the key to enabling the on-demand use of software based on the available resources while considering different requirements and constraints within the software-defined vehicle. For instance, for energy and cost efficiency purposes, a low-energy mode of the vehicle may require the intelligent selection of containers or ECUs to turn off based on container priorities and available resources.

To handle the management of software containers in different applications, various approaches [122, 123, 124] have been provided to orchestrate their execution based on demand. Docker Swarm [122] is an open-source technology that allows users to manage several Docker hosts and orchestrate a set of containers on these hosts via an easy-to-operate, high-availability, and failover mechanism called Spread. Spread, basically schedules a service task based on spreading the number of containers equally to all Docker hosts, and all the extra configuration has to be performed, manually [37].

Kubernetes [123] favors nodes with a balanced resource usage rate. A cluster manager that has been recently developed that provides efficient resource isolation is Apache Mesos [124]. It allows users to have clusters of containers. Each cluster is made of one master that allocates tasks to workers using frameworks. However, limited resources are rarely observed in the cloud, unlike with edge devices in cars. The present commercial systems for software containers are primarily developed for cloud environments with adequate resources, as their primary focus is to orchestrate the execution of the containers to accomplish specific tasks rather than the emerging edge computing paradigm.

Other scheduling techniques [67, 68] consider the limited resources (such as CPU, memory, and energy) on edge devices and aim to efficiently schedule software containers in order to optimize resource usage. These present systems, however, have different shortcomings when scheduling containers in software-defined vehicles. Software-defined vehicles offer challenges in the form of many requirements that need to be satisfied. Software reliability is critical since smart vehicles are software-driven (e.g. drive-by-wire systems) and any kind of software failure can put the lives of the passengers at risk [125]. Next, the software has to be secure to prevent unauthorized access to the in-vehicle network and with modern vehicles “knowing” more about their passengers than ever before, it needs to ensure privacy protection [126]. Vehicles can swiftly switch to power-saving modes and prioritize applications in emergencies. They may falter in seamlessly handling software

security issues such as ECU failures or upgrades and often struggle with the intricate real-time constraints, and challenges of the automotive edge environment from software priorities to placement constraints and resource efficiency, to adaptation to the driver demands, enabling higher personalization, and software and hardware updates. In the realm of container management in software-defined vehicles, these challenges are multifaceted, where the existing scheduling techniques for edge, often designed for static environments grapple with the dynamic management of SDVs. In essence, while software-defined vehicle challenges are pressing, many existing techniques might be ill-equipped and lack the comprehensive adaptability and sophistication needed for this dynamic landscape. The execution of software containers for SDVs in automotive requires flexible, dynamic, and effective scheduling that can not only balance several conflicting objectives such as optimizing the energy, CPU, and memory consumption, and balancing the load among cluster nodes (e.g., ECUs), but also, conforme to the automotive constraints and real-life scenarios within the SDV.

To tackle the above challenges, we proposed in this chapter a dynamic software container scheduling approach based on NSGAIII for embedded devices in software-defined cars. To deal with the conflicting objectives and real-time requirements and constraints, our search-based approach is based on a many-objective search to find an adequate management solution, i.e., the allocation of containers to the ECUs that not only delivers optimized usage of the resources while balancing the load between the ECUs of SDVs but also adapts to the different real-time scenarios that can happen within the car and respects the environment needs and constraints. The adoption of a many-objective search algorithm proves beneficial given the vast search space of possibilities for allocating containers to different ECUs while addressing conflicting objectives and constraints. Our approach enables the move, shut-off, and execution of software containers between ECUs dynamically based on the continuous monitoring of resource usage (energy, memory, and CPU).

The proposed approach considers multiple conflicting objectives, encompassing structural improvement (reducing the number of nodes and the average of containers per node), resource consumption minimization (CPU, memory, Power), and coupling between containers while adapting to different use cases (ECUS Failure/Upgrade, Low Energy Mode, Security attacks, Introduction of new ECUs in the cluster) and respecting different constraints including container priorities, the dependency between containers and container placement conditions. Even though this work appears directly related to the automotive industry, its applications extend beyond those initial boundaries. It applies to a wider array of devices, encompassing phones. The key focus lies in optimizing load distribution among ECUs, not confined solely to cars. The approach aims at minimizing CPU, memory, and power consumption during power-saving mode while addressing constraints, with the overarching goal of making a meaningful impact on various constrained devices.

In collaboration with our industry partner, the Ford Motor Company, we evaluated our approach using different real-world scenarios in a heterogeneous cluster of Raspberry Pi devices, which simulate ECU devices in the vehicle. The scenarios were based on original software containers provided by our partner. Results also show that our approach is more efficient when compared to the widely used management solutions, such as the Docker Swarm Strategy (Spread), in effectively balancing the load between ECUs in the car, as well as minimizing the CPU, memory, and power consumption when in power-saving mode while dynamically dealing with the real-time requirements and constraints in the car. Also, experiments show that many-objective search (NSGA-III) generates solutions that improve the quality indicators for computational search (i.e., Hypervolume (HV), Inverted Generational Distance (IGD) and Contribution (IC) metrics) and reduces the CPU, memory, and power usage better than other many-objective algorithms (NSGA-II [56], and IBEA [86] algorithms).

Contributions. The main contributions of this work are summarized as follows:

1. Based on the continuous real-time monitoring of the vehicle environment, we introduce a novel formulation of software container management in SDVs as a many-objective problem on top of a dominant container orchestration tool, Docker. It not only considers balancing several conflicting objectives such as the number of utilized ECUs, the load between the ECUs, energy, CPU, and memory consumption but also conforming to diverse use cases (ECUS Failure/Upgrade, low energy mode, security attacks, introduction of new ECUs in the cluster). Additionally, it adheres to various constraints, including container priorities, the dependency between containers (network and data transmission), and container placement conditions when the containers are executing inside the vehicle.

2. We compare the performance of the proposed algorithm to a baseline of deterministic method and search techniques (i.e., random search and mono-objective optimization), three existing techniques using many-objective optimization (NSGA-II, NSGA-III, and IBEA), and docker strategy Spread, in reducing the CPU and memory usage in heterogenous environment and improving the values of the quality indicators (i.e., Hypervolume, IGD and Contribution metrics).

3. We report the results of applying our proposed algorithm to critical real-world scenarios in collaboration with the Ford Motor Company and we evaluate the usefulness of our approach in reducing resource consumption while coping with the requirements and constraints within smart SDVs.

4. We develop and utilize open-source SDV-related scenarios that are deeply rooted in rigorous academic research and align with industry-standard practices. These scenarios were carefully crafted through a meticulous and iterative process, in close collaboration with our partner in the automotive industry.

Data sharing and availability The studied scenarios and results used in our study, as well as a demo for the framework and the dashboard, are available on our website [4]. Any other data used in our experiments (e.g. containers) cannot be disclosed publicly due to proprietary constraints imposed by our industrial collaborator.

5.1 Software-related systems and Real-life use cases

SDVs such as Smart and connected vehicles [113, 114, 115, 116] include many applications to enable different features such as driving assistance (e.g., ADAS [117, 118]), intelligent path planning (e.g., ITS [119]), monitoring of the cameras and sensors, media, intelligent speed control, detecting traffic signals, and so on. The automotive industry is focusing on the containerization of the software applications running in cars. To run these containers, The hardware model of an automotive embedded system is composed of a distributed set of **Electronic Control Units (ECUs)**, which have different and limited capacities of memory, processing power, and CPU. Each ECU has access to different sensors, which impose localization constraints for some software containers. In other words, software containers that read from a specific sensor or write to a particular actuator must be deployed to an ECU that has access or near to the sensor or actuator. An example is depicted in Figure 5.1, where ECU3 and ECU6 are close to the Engine sensor and Brakes sensor respectively. In this case, containers designed for processing engine data or braking data will need to be run on the appropriate ECU. This is where the "placement constraints" come in. A software container designed for the engine can't just run on any ECU; it needs to run on ECU3, which is connected to the Engine sensor. Similarly, a container for brakes needs to run on ECU6.

We present in the following an overview of the main motivations related to the problems addressed in this work. They are illustrated based on real-world scenarios reflecting the critical needs of the automotive industry to find efficient solutions to enable

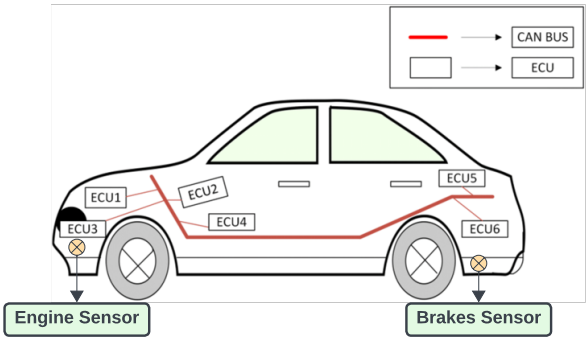

Figure 5.1: The Hardware Architecture of an Automotive System

the deployment and usage of software containers in the vehicle under different real-time constraints. We explain how these use cases have been developed in section 7.2 specifically in **RQ2**.

Figure 5.2 and Figure 5.3 depict real-world scenarios in the context of software-defined vehicles as defined in collaboration with researchers and engineers from our partner, the Ford Motor Company, a prominent manufacturer of cars, systems, and services in the automotive industry. Figure 5.2 shows a real-world scenario in which a software-defined car in normal mode is unable to reach the driver's destination because the available battery power is insufficient. In this situation, a monitoring system determines if the current power consumption by the software applications running in the car allows for reaching the targeted destination or at least a nearby charging station. In either situation, the car needs to switch into a power conservation mode, the promoted ECUs are placed in Low Power Mode (LPM) by lowering their power consumption boundaries, and critical applications are relocated to the designated ECUs by the management/scheduler solution. When the driver finds a nearby gas station, the battery is charged, and the car may resume regular operation. The same scenario applies as well when full memory or CPU is about to be achieved due to the large number of software applications running by the driver in the car. Indeed, the current shift in automotive towards a subscription-based model for the

applications running in the car dramatically increases the resources needed to run many apps.

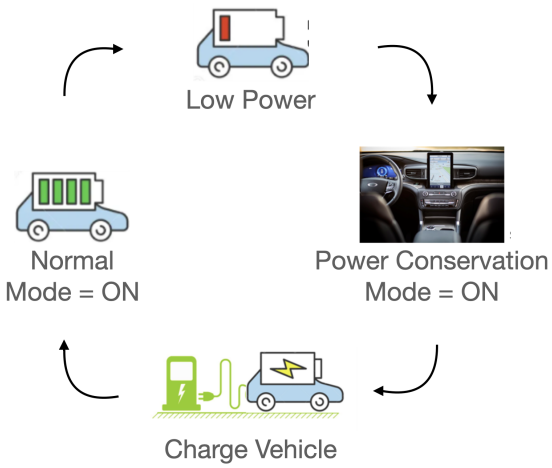

Figure 5.2: Real-world use case: Power conservation

Figure 5.3 illustrates another use case scenario in which the leader node is swapped in response to an upgrade request, a failure (e.g., bug, security attack, etc.), or when the ECU is consuming a lot of energy and has to be completely shut down. To deal with these situations, the following steps should be followed: The targeted ECU should be first isolated, then replaced with a promoted node as the leader; Furthermore, the containers of the isolated ECU should be allocated to another designated ECU while taking into account all of the constraints including containers priorities, placement constraints, available CPU, memory resources, and energy per ECU. For example, if applications are assigned various priorities and the resources are limited, the containers with lower priorities should be deactivated to allow the containers with higher priorities to run once the leader ECU fails successfully, and its isolation is necessary. When the isolated ECU returns to normal (e.g., finished upgrade, normal power level, failure tolerance etc.), the manager of the containers would be able to rejoin the cluster and re-balance the distribution of the containers accordingly.

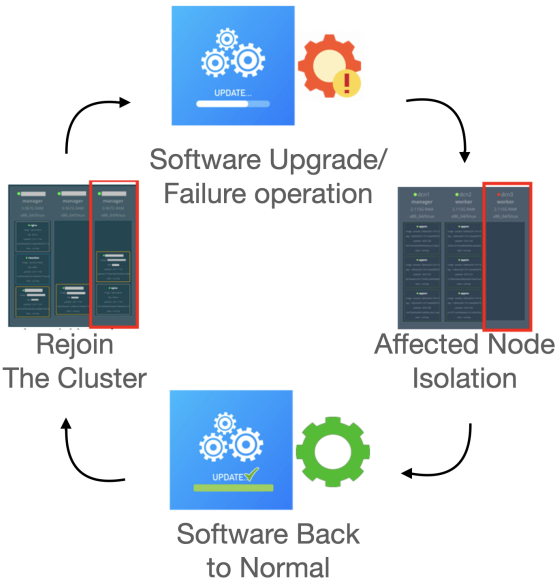

Figure 5.3: Real-world use case: Software upgrade/Failure operation

The existing scheduling techniques aren't optimized for such intensive utilization and constraints leading to potential inefficiencies and overloads. They fall short when faced with the unique challenges and requirements of the automotive edge environment, such as real-time constraints and requirements, failure tolerance, and security that can be present within the vehicle.

5.2 Research Methodology: A Many-Objective Scheduling Approach for Software Containers

Figure 5.4 presents an overview of the framework architecture. The first component is the *Docker Cluster*, composed of various in-vehicle applications that facilitate various functionalities, including driving assistance (e.g., ADAS), intelligent path planning (e.g., ITS), camera and sensor monitoring, media capabilities, intelligent speed control, traffic signal detection, and management of the engine and brakes, etc. These containers are running in the different ECUs in the car. The scheduler requires three primary inputs: 1) The current status of the ECUs in the vehicle, 2) the current status of the containers, and 3)

real-time requirements and constraints specific to the vehicle. To pull metrics related to the containers' status, Cadvisor [96] is responsible for providing an understanding of the default distribution of the running containers between the ECUs in the car as well as their resource usage and performance. nodeExporter [127] helps us collect the vehicle status metrics, such as the number of ECUs per cluster, memory, CPU, and Power (Energy) utilization per each node. *Prometheus* [95], an open-source system monitoring and alerting toolkit that collects and stores both containers and ECUs metrics as time series data and updates them for every period of time to feed them to the Scheduling component.

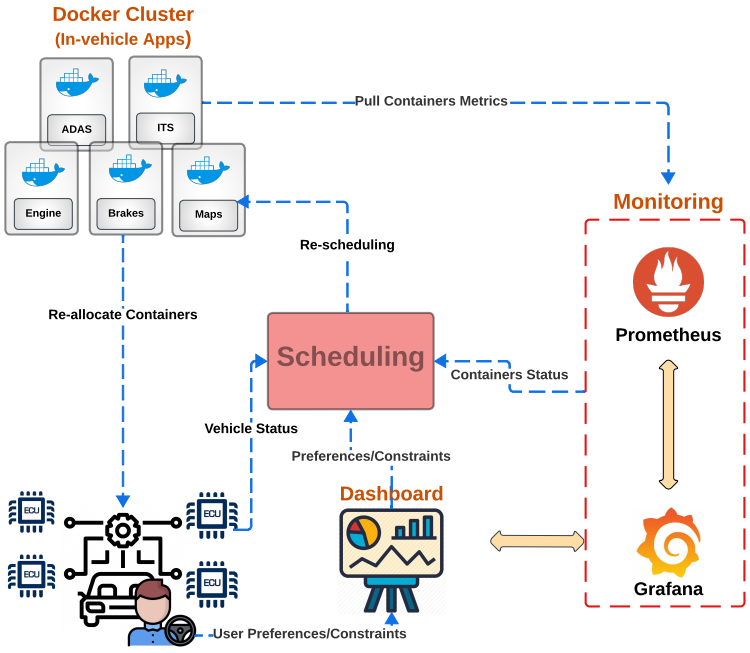

Figure 5.4: Architecture of the proposed framework.

The scheduler receives as input the above-mentioned real-time data: containers status, vehicle status, and preferences and constraints. These inputs are periodically updated to monitor any changes in their values. The scheduler is activated in two use cases: when

users choose to optimize specific objectives or add new constraints and require the reallocation of vehicle applications based on the updated input. The second use case is when an actual practical scenario occurs: The cluster's resource usage is beyond its limit, resulting in the deactivation of one of the ECUs. This deactivation can occur for security reasons, during an upgrade or failure, or when the car's battery capacity is insufficient and it needs to go to power conservation mode. Alerts appear on the dashboard display, providing specific information about the alerts and recommending scheduling options. After executing the scheduler, a potential solution can be implemented to redistribute the containers among the various ECUs. Finally, The *Monitoring Component*, uses PromQl, the Prometheus query language, to scrape the real-time data and display it on our dashboard, such as the real-time distribution of containers per node, activated nodes in the cluster, etc. Users, via the dashboard, are allowed to change some configurations related to the constraints such as container priorities (When in a power-saving usage mode), placement constraints or assign weights to the objectives to optimize based on their relative relevance. The scheduling component is updated with any modifications made to the dashboard before executing the scheduler. Examples of real-time resource visualization are shown in [4].

5.2.1 Scheduling Component

Since we are using the same multiobjective optimization approach as the first contribution, to avoid redundancy, all the sections including the DCF Parser, Optimization, and Representation of the Individuals are detailed in subsection 4.1.1.

Also, all the four objectives presented in subsection 4.1.1 are used in the context of SDVs. Therefore, in the following, we will focus on the rest of objective functions we included in this contribution.

5.2.1.1 **Objective Functions** In the development of our dynamic software container management system for vehicles, we have anchored our optimization objectives in rigorous studies and industry-standard practices. These objectives were meticulously curated through an iterative process in collaboration with our automotive industry partner. This process involved a series of consultations and refinements to ensure that the objectives were tailored to a wide array of use cases within the vehicle's ecosystem, drawing from a rich tapestry of automotive references and scheduling-related works in embedded devices [121, 89, 90, 91, 93, 128, 68, 129, 130, 131, 132]. To solidify the integrity and applicability of these objectives, they were subjected to a stringent peer-review process by seasoned automotive engineers at Ford who were not directly involved in the project. This critical review served to fine-tune the objectives, ensuring they are not only theoretically sound but also pragmatically viable. This meticulous approach to defining our objectives underscores our commitment to quality and relevance in addressing the computational challenges posed by the execution of containers in the constrained environments of ECUs within software-defined cars.

Let $C = \{c_1, c_2, c_3, \ldots, c_n\}$ be the set of available containers, and $N = \{n_1, n_2, n_3, \ldots, n_m\}$ be the set of available nodes/ECUs. We define and compute the objective functions that rely on C and N as follows.

The **first, second, third, and fourth** objectives align with those outlined in subsection 4.1.2 under the first contribution, as this chapter expands on the software containers approach for scheduling containers in the cloud (discussed in detail in Chapter IV).

The ***fifth***, and ***sixth and seventh*** objectives are related to optimize the resource efficiency by minimizing the CPU, Memory, and Power consumption per node. Balancing the load between nodes in the cluster while respecting resource limits is one of the most important objectives considered in embedded devices with limited constraints [133, 39], and

it is one of the main goals of our study. Efficient resource management and a balanced load between the ECUs will ensure the vehicle software's reliability and quality, enabling the vehicles to continually adapt to the needs of their passengers and guarantees its safe for drivers, passengers, and other road users [131]. ***MemConsumption*** (Equation 5.1) computes the average of memory consumption per node. This objective relies on the standard deviation of the memory consumption per node. The objective is defined as follows:

$$MemConsumption(S) = \sqrt{\frac{1}{|M|} * \sum_{m_i \in M} (m_i - \mu)^2} \tag{5.1}$$

where $M = \{m_1, m_2, m_3, \ldots, m_n\}$ is the percentage of memory consumption for each node for a specific period of time. For this project, we calculate the average for the last five minutes n divided by the total percentage of memory consumption by all containers in the cluster, and μ is the mean of M.

CC (Equation 5.2) computes the average CPU usage per node. This objective relies on the standard deviation of the percentage of CPU consumption per node. The objective is defined as follows:

$$CC(S) = \sqrt{\frac{1}{|CPU|} * \sum_{c_i \in c=CPU} (c_i - \mu)^2} \tag{5.2}$$

where CPU is the sum of the percentage of CPU usage for each node at time t (the last five minutes of t) over the total percentage of CPU consumption of the containers, and μ is the mean of CPU.

CC (Equation 5.3) computes the average Power usage per node. This objective relies on the standard deviation of the percentage of Power consumption per node. The objective is defined as follows:

$$PowerConsumption(S) = \sqrt{\frac{1}{|Power|} * \sum_{p_i \in p = Power} (p_i - \mu)^2} \tag{5.3}$$

, where $Power$ is the sum of the percentage of Power usage (in watts) for each node at time t (the last five minutes of t) over the total percentage of Power consumption of the containers, and μ, is the mean of $Power$

The ***eighth objective*** emphasizes the importance of prioritizing containers based on their significance in a specific use case, particularly in a power usage mode as referenced in section 5.1. In scenarios where power conservation is crucial, it becomes imperative to manage container operations judiciously. The objective suggests that containers with lower priorities should be temporarily deactivated to ensure that those with higher priorities continue to function optimally. This approach ensures that essential functionalities, like enabling a driver to reach the next power station, are not compromised. ***Pr*** (Equation 5.4) computes the container's priorities as the ratio of the sum of running containers' priorities over the sum of the priorities assigned to the set of containers.

$$Pr(S) = \frac{\sum_{c_r \in RC} Priority(c_r)}{\sum_{c \in C} Priority(c)} \tag{5.4}$$

The goal of our Many-Objective Optimization algorithms is to minimize the objective functions described above. We formulate our problem as follows:

$$\underset{S}{\text{minimize}} NON(S), FRQ(S), COP(S), PC(S), Pr(S), MC(S), CC(S), CHG(S) \tag{5.5}$$

Note that all the objective functions are normalized in the range [0, 1], and the lower the value, the better the solution. To clarify how the objective functions are computed, consider C={0, 1, 2, 3, 4, 5} as the set of available containers, *Priorities*={1, 1, 2, 3,4, 2}, as their priorities (1 is the highest priority), *AveragePowerConsumption*={52%, 24%, 45%, 32%,

14%, 80%} as the average power consumption per container for the last 5 minutes, *AverageCPUConsumption*={0.3%, 5%, 0.45%, 0.11%, 0.21%, 0.35%} and *AverageMemoryConsumption*={14%, 24%, 15%, 33%, 44%, 18%} and G as the dependency graph available in Figure 4.2. $N = \{1, 2, 3, 4\}$ is the set of available nodes, and $P^{(t)} = \{3, 3, 3, 1, 2, 2\}$ as the current cluster State. Now, consider that a solution $S = \{3, 3, 1, 2, 1, -1\}$ was generated (the same available in Figure 4.3) by an optimization algorithm to be evaluated, the objective functions are calculated as follows:

1. NON(S) = Three nodes were selected in the candidate solution (node of id 1, 2, and 3) of a total of 4 nodes = 0.75
2. FRQ(S) = $\sqrt{\frac{0.10}{3}} = 0.18$
3. COP(S) = In a total of 6 dependencies, 4 of them are inter-dependencies (dependencies between containers allocated on different nodes) = $\frac{4}{6} = 0.66$
4. MC(c) = $\sqrt{\frac{0.5}{3}} = 0.408$
5. CC(c) = $\sqrt{\frac{0.10}{3}} = 0.67$
6. Pr(S) = $\frac{11}{13} = 0.67$
7. CHG(S) = 4 containers were suggested to move in a total of 6 containers= $\frac{4}{6} = 0.66$

5.2.1.2 **Constraints** The software container scheduling problem in software-defined cars is often constrained by various factors:

1. Containers should be assigned to active ECUs throughout the scheduling process. In case of a failure/upgrade of a specific ECU, the scheduler must get notified and promptly allocate the containers on activated nodes.

2. Every node/ECU has a maximum CPU and memory utilization that should not be exceeded.

3. Every ECU has a maximum power consumption that should not be exceeded.

4. Some software applications should run close to some sensors in the vehicle as detailed in section 5.1. Therefore, some containers with these placement limitations should be assigned to specific nodes.

5. When containers have interdependencies, the operational state of one can directly influence the functionality of another. In the context of a power-saving mode, even if a particular container is deemed essential and is kept running, it's imperative that all its dependent containers also remain active. This is regardless of their individual priorities. The reason is, that if a container relies on others for its functionality, deactivating its dependencies could disrupt its normal behavior. Thus, while it might seem counterintuitive to keep lower-priority containers running in a power-saving mode, it's a necessary measure to ensure the proper functioning of higher-priority containers that depend on them. This underscores the importance of understanding and managing container dependencies effectively, especially in scenarios where resources are constrained or are being optimized.

5.3 Validation

In this section, we introduce our research questions (RQs), provide an overview of the experimental design, and present results that support our responses to the two main addressed RQs detailed in the subsequent content:

RQ1: How does the scheduling performance of our NSGA-III-based approach for containers compare to commonly used state-of-the-art search-based algorithms and the default scheduler in Docker?

This question aims to compare the effectiveness and efficiency of our many-objective optimization approach with a selected set of state-of-the-art search-based algorithms, including default Random search, IBEA, NSGA-II, and Mono-Objective. These algorithms were chosen because they represent some of the most popular optimization techniques, each belonging to a different category. Random Search is often used as a baseline in optimization studies [134] since it provides a point of reference to assess the improvement offered by more sophisticated approaches. Our inclusion of Mono-Objective algorithms is strategic because it highlights the contrast between single-objective and multi-objective optimization approaches and the limitation of the single-objective approach in simultaneously addressing competing objectives [135]. IBEA and NSGAII are Known for their ability to handle multi-objective optimization problems and are frequently employed in analogous contexts [54, 53]. We also conducted a comparison with the docker default scheduler Spread [136], which is utilized by our industrial partner. In this work, we did not compare our approach with existing scheduling approaches in the current literature mentioned in Section 3.1 for two key reasons. Firstly, these schedulers cannot frequently grant access to their artifacts. Furthermore, they lack the necessary tools and expertise to dynamically tackle the precise automotive goals and limitations that our study is centered around.

To answer RQ1, three quantitative experiments were conducted: First, To evaluate both the quality and spread/diversity of their solutions, we compared the algorithms with the same quality indicators used in subsection 4.2.2

These experiments were conducted with three simulated ECUs (In contemporary software-related vehicles, they can go with as low as three ECUs [132], which is the average

of ECUs used by our industrial partner in their cars) and with a number of containers that vary from 25 to 100. All the containers used in our experiments are automotive-related, each with distinct characteristics such as type, size, and resource consumption. These containers were generously provided by our collaborator, but cannot be disclosed publicly due to the imposed proprietary constraints.

Second, we evaluated the effectiveness of our search algorithm in balancing the load in a heterogeneous environment considering varied resource limitations compared to existing techniques. Therefore, we considered two representative real case studies in which ECUs had different resource capabilities.

The experiments consist of 50 containers allocated across 3 ECUs, representing low CPU/high memory resources and high CPU/high memory resources, respectively. In our experiments, we maintain the number of ECUs at three, aligning with the automotive industry standard for modern software-defined vehicles. There has been a significant trend towards consolidating the number of ECUs into three, reflecting industry practice [132, 131]. This is further confirmed by our industrial partner, who emphasizes their goal of optimizing the number of ECUs in the car to three. We evaluated our approach with 50 containers, as it represents the average operational load in our partner's current SDVs.

As shown in Table 5.1, in the first use case (UC), ECU1 possesses a memory capacity that is four times greater than that of ECU2 and ECU3. Conversely, ECU2 and ECU3 have CPU resources that are four times superior to those of ECU1. Whereas in UC2, ECU1 possesses both memory and CPU capacity that is four times greater than that of ECU2 and ECU3. We aim to assess our methodology by taking into account the various limitations of resources within the vehicle and how it can efficiently allocate the containers to ensure an equitable distribution of load among the ECUs. CPU and memory usage metrics are vital for evaluating the performance of individual nodes. they provide critical insights into how effectively each node is operating and how well it handles its workload,

they help in identifying resource utilization patterns, detecting potential performance issues, and making informed decisions for optimizing software performance. Therefore, we computed the % of CPU and memory usage per ECU for each search algorithm used to evaluate the effectiveness of our search algorithm in balancing the load in a Heterogeneous environment compared to the existing techniques.

Table 5.1: Heterogeneous environment.

Use Case	ECU ID	CPU	Memory(GB)
UC1	1	1	4
	2	4	1
	3	4	1
UC2	1	4	4
	2	1	1
	3	1	1

Finally, since the execution time is critical when applying evolutionary algorithms, we evaluated the scalability of our method to existing techniques by reporting for each search algorithm the execution time for various numbers of containers (from 10 to 100) and three ECUs.

The performance of the evolutionary algorithms NSAII, NSAIII, and IBEA is significantly influenced by their parameter settings. Therefore, for each algorithm, we perform a set of experiments where we vary certain parameters such as the population sizes [103] and the maximum number of generations. We execute each algorithm 30 times with various configuration settings and we apply the trial and error method in order to select the fittest setting. Noting that if the results are similar for a given combination of parameters, the execution time was considered. For the evolutionary operators, we adopted Integer SBX crossover, Integer Polynomial mutation, and binary tournament for selection of the individuals [105] because they have been designed to work with integer solutions.

Therefore, the list of selected parameters used to answer the stated research questions is described in Table 5.2.

Table 5.2: Parameter settings.

Parameter	**NSGA-III**	**NSGA-II**	**IBEA**
Population Size	200	200	100
Maximum Number of Generations	2500	2500	2500
Crossover Probability	0.95	0.9	0.9
Mutation Probability	0.05	0.01	0.01
Crossover Operator	Integer SBX		
Mutation Operator	Integer Polynomial		

Result & Discussion. Table 5.3 and table 5.4 summarize the results of mean values and standard deviations in terms of HV, IGD, and IC indicators over 30 independent execution runs. They depicts the results of considering all the objectives for the evolutionary algorithms. The values of the objectives were standardized between 0 and 1 and set to be minimized; the sequence of the objectives has no bearing on the outcomes. All the results were statistically significant on the 30 independent simulations using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

Table 5.3: HV, IGD, and IC mean values with Our Approach and IBEA. Results are based on 30 independent simulation runs using the Wilcoxon rank sum test with 99% confidence level (α<1%). (Highest values are in bold)

#C	**Our Approach**			**IBEA**		
	IGD	**HV**	**IC**	**IGD**	**HV**	**IC**
25	**0.020 ± 0.000**	**0.506 ± 0.000**	**0.666 ± 0.000**	0.025 ± 0.000	0.000 ± 0.010	0.333 ± 0.010
50	**0.021 ± 0.001**	**0.144 ± 0.001**	**0.644 ± 0.001**	0.023 ± 0.007	0.138 ± 0.007	0.612 ± 0.007
75	**0.000 ± 0.000**	**0.146 ± 0.000**	**1.000 ± 0.000**	0.004 ± 0.004	0.144 ± 0.004	0.978 ± 0.004
100	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.000 ± 0.000	0.080 ± 0.000	0.997 ± 0.000

Among the search algorithms, NSGA-III exhibits the lowest IGD value, indicating superior average performance. Additionally, NSGA-III consistently produces outstanding

Table 5.4: HV, IGD, and IC mean values with Random Search (RS) and NSGAII. Results are based on 30 independent simulation runs using the Wilcoxon rank sum test with 99% confidence level (α<1%).

#C	RS			NSGAII		
	IGD	HV	IC	IGD	HV	IC
25	0.025 ± 0.000	0.000 ± 0.000	0.333 ± 0.000	0.025 ± 0.000	0.000 ± 0.000	0.333 ± 0.000
50	0.054 ± 0.020	0.108 ± 0.020	0.355 ± 0.020	0.023 ± 0.007	0.138 ± 0.007	0.612 ± 0.007
75	0.103 ± 0.024	0.135 ± 0.024	0.806 ± 0.024	0.004 ± 0.004	0.144 ± 0.004	0.978 ± 0.004
100	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000

results in terms of HV and IC values, demonstrating that its solutions are much closer to the ideal set as compared to those provided by other approaches. Furthermore, we observed that the solutions of all search algorithms are distributed across the objective space to some extent. Specifically, RS and NSGA-II exhibit the weakest performance. On a smaller scale, with up to 25 containers, NSGA-III provides better IGD with 0.02 compared to 0.025 for NSGA-II, RS, and IBEA. (the smaller the IGD values, the better the results are). NSGA-III also gives the best results in terms of HV and IC with 0.506 and 0.666, respectively. In contrast, the three algorithms, including IBEA, equally provide 0.0 and 0.333 for the HV and IC, respectively.

On a medium scale, specifically between 50 and 75 containers, NSGA-III outperforms NSGA-II, IBEA, and RS in terms of IGD with a maximum of 0.021 compared to a maximum of 0.023, 0.103, and 0.023 for NSGA-II, IBEA, and RS, respectively. Similarly, NSGA-III surpasses NSGA-II, IBEA, and RS in terms of HV with a minimum of 0.144 compared to a minimum of 0.138, 0.108, and 0.138 for NSGA-II, IBEA, and RS, respectively.

Furthermore, in terms of IC, NSGA-III outperforms NSGA-II, IBEA, and RS, with a minimum value of 0.000. In comparison, NSGA-II, IBEA, and RS show minimum values of 0.004, 0.02, and 0.004, respectively.

On a large scale, NSGA-III outperforms all the other algorithms in terms of IGD, HV, and IC by providing as results 0.00, 0.083, and 1.00 respectively. These results align with prior research in other areas. For instance, Random search can be found to be inadequate for handling multi-objective optimization problems with conflicting objectives [137], while NSGA-II and IBEA do not perform well with more than 2 or 3 objectives [138, 54]. Therefore, their performance is not guaranteed and can be quite inconsistent. Compared to other contemporary many-objective algorithms, NSGA-III is explicitly designed to address multi-objective optimization problems and can provide a set of Pareto optimal solutions, offering a trade-off between up to 15 conflicting objectives. This unique capability could contribute to the observed superior performance of NSGA-III.

Figure 5.5 and Figure 5.6 show the percentage of resource usage per node for the first use case, UC1. The obtained results illustrate that the load is not distributed evenly across the nodes. Regarding memory utilization (Figure 5.5), ECU1 has a very low memory usage compared to the other ECUs for all techniques (Spread, RS, Monobjective, NSGAII, and IBEA). However, using our approach, we can determine that all nodes have extremely similar memory utilization (55.76%, 66.21%, and 69.66%).

Also, Figure 5.6 shows that the CPU usage is not balanced between the three nodes. When Spread and RS are applied, the CPU utilization of ECU2 and ECU3 is lower than ECU1. For example, the Spread consumption is 9.8%, 2.7%, and 2.0% for ECU1, ECU2, and ECU3, respectively.

The findings demonstrate that the Spread algorithm allocates containers evenly across nodes without considering the varying resource capacities of each node, while the Random Search algorithm, as its name suggests, is characterized by randomness.

The approach fails to consider the potential influence of learning or guidance derived from the search history or the problem's structure. Nevertheless, a notable balance in CPU use is observed for the NSGA-III, IBEA, and Mono-objective Algorithm when

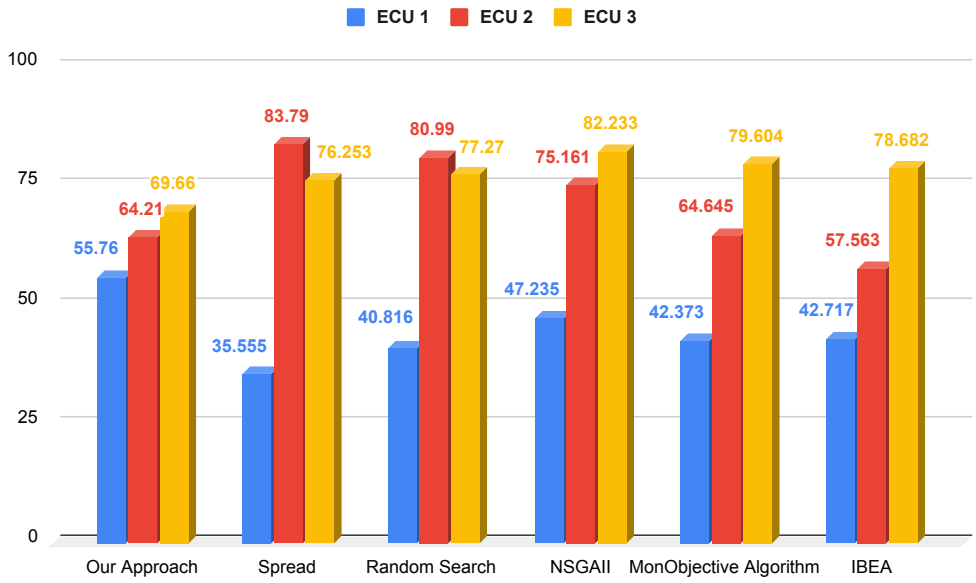

Figure 5.5: The percentage (%) of memory usage per node (UC1).

comparing ECU1 to ECU2. However, it is evident that the workload distribution is not well balanced between ECU1, ECU2, and the third node.

Using our methodology reveals an equitable distribution of CPU load across the nodes, with Node 1 accounting for 6.4%, Node 2 for 5.7%, and the third node for 5.844%. Our approach takes into account the diverse resource capacities of the ECUs and determines the optimal allocation of containers while adhering to the resource limitations of each ECU and achieving a balanced load distribution.

Figure 5.7 and Figure 5.8 show the resource usage per node reported for the second use case, UC2. We can observe from the results that the CPU and memory usage are low for ECU1 compared to nodes 2 and 3 for all algorithms. Spread and Random Search, for example, exhibit CPU utilization of 5.13% and 7.4% for ECU1, compared to 18.27% and 48.670% for ECU2, while ECU3 attained 29.867% and 16.273% respectively. In terms of memory usage (Figure 5.7), the Mono-objective method and NSGA-II exhibit memory consumption of 36.994% and 24.437 respectively, for ECU1, compared to 64.458% and

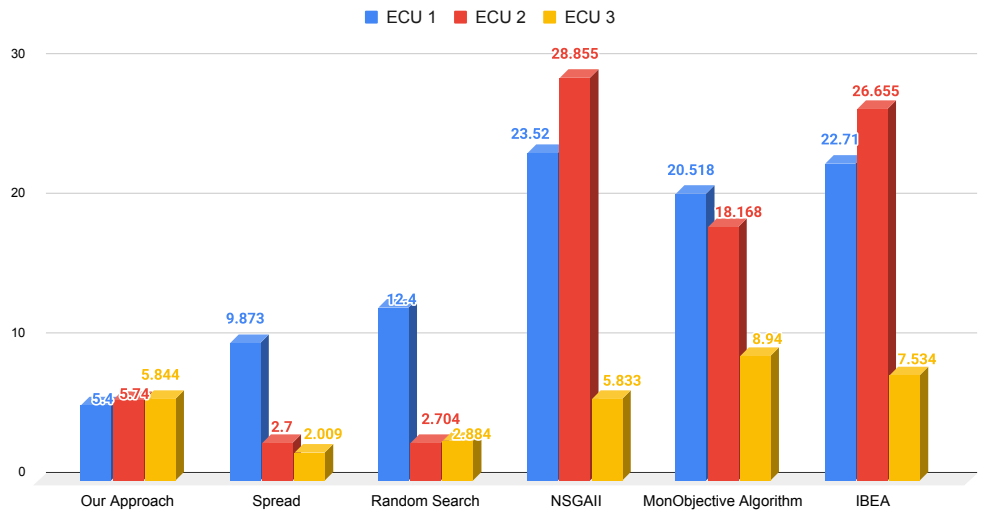

Figure 5.6: The percentage (%) of CPU usage per node (UC1).

57.492% for ECU2 and 74.686% and 63.684% for ECU3. In contrast, we can observe that for both CPU and Memory usage using our approach, the load between the three nodes is well-balanced compared to the rest of the algorithms. For instance, the CPU usage for the three nodes is 12.844%, 15.704%, and 14.844% using our algorithm.

The above findings from both UC1 and UC2 result from our approach's ability to handle the conflicting objectives we have in terms of balancing the load of CPU and Memory usage between the cluster. By making use of information about the problem, such as the different resources for each ECU, our technique yields the best results in balancing the load between the ECUs while respecting each ECU's resource constraints.

In order to test the scalability of our approach in terms of successfully handling different numbers of containers, far exceeding the average operational load (up to 50 containers) in our partner's current SDVs, we calculate the scheduling time in seconds for managing up to 100 containers of varying computational and resource demands—from large-scale containers like those managing engine and brakes to medium-scale (e.g., heaters and maps), and low-scale applications (e.g., lights and setup). These results are illustrated in

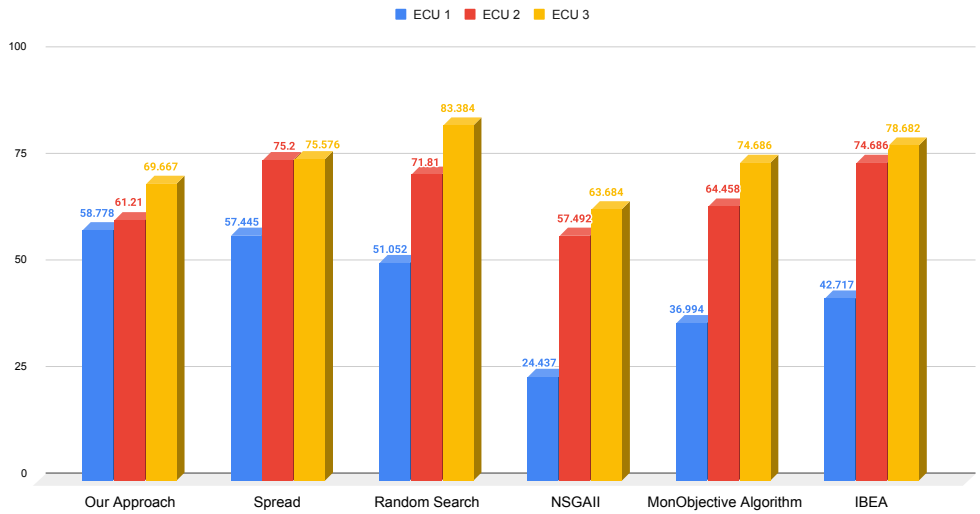

Figure 5.7: The percentage (%) of memory usage per node (UC2).

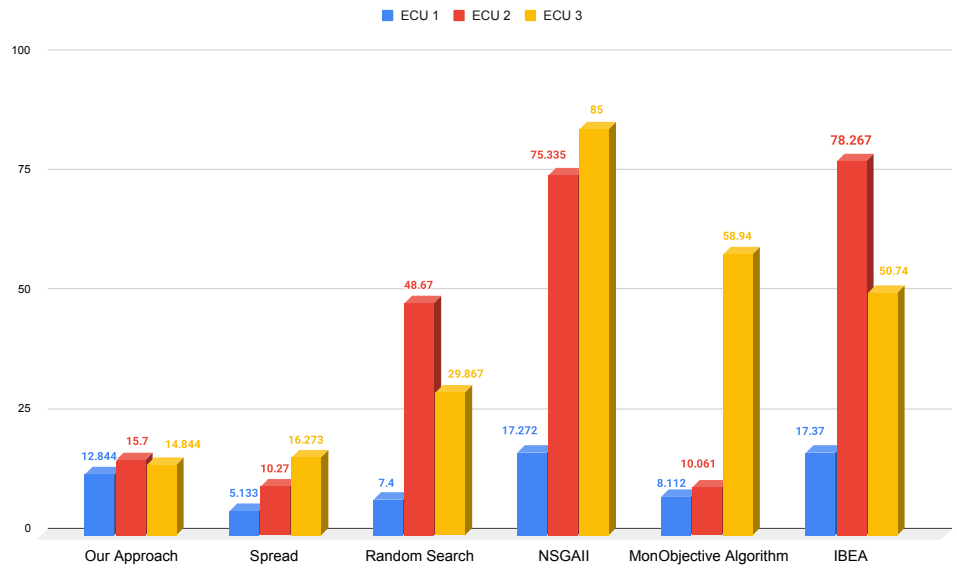

Figure 5.8: The percentage (%) of CPU usage per node (UC2).

Figure 5.9. Our NSGA-III algorithm is the fastest among the five algorithms, where the execution time is, on average, 39.5 seconds compared to 41.6, 49.5, 47.2, and 45.5 for RS, IBEA, NSGA-II, and Mono-Objective, respectively. The computational effort required by

NSGA-II and IBEA algorithms to find the best solution might explain the slowness thereof. Furthermore, due to the limited distribution of solutions in the Pareto front, when considering more than three objectives, NSGA-II may take longer to locate relevant solutions than many objective algorithms. The average difference in the time required to run the higher and lower number of containers (from 10 to 100 containers), is approximately 66 seconds, whereas the difference for RS, IBEA, NSGA-II, and Mono-Objective is, respectively, 67, 69, 74, and 71 seconds. We conclude that our approach scales the best in terms of the number of containers running in the cluster. Studying the scalability of our approach in terms of different numbers and sizes of containers highlights the practical applicability of our proposed container management system in the automotive industry.

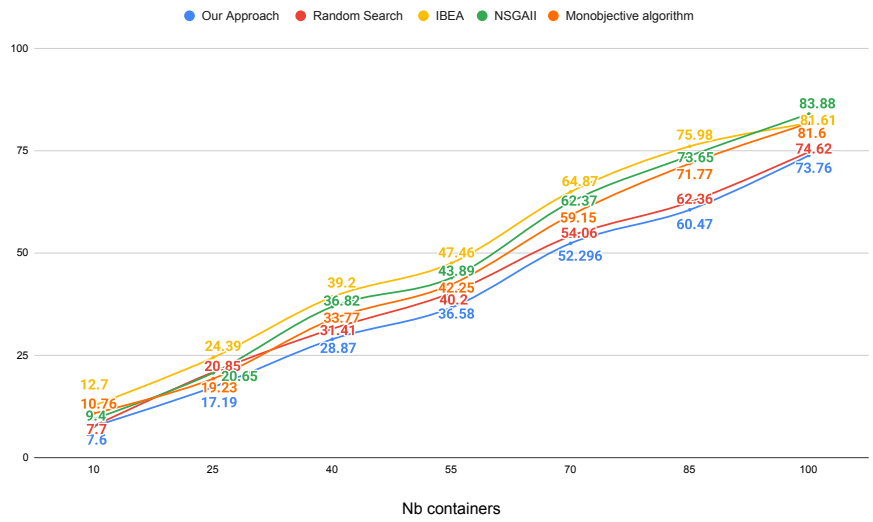

Figure 5.9: Scheduling time (in seconds) per number of containers. For a more readable version, you can find it on the website [4].

While Containers Replicas in Docker are utilized to ensure that we mitigate any potential issues stemming from slight latency in the scheduling process, we conducted additional experiments to reinforce our confidence in their reliability. Our experiments

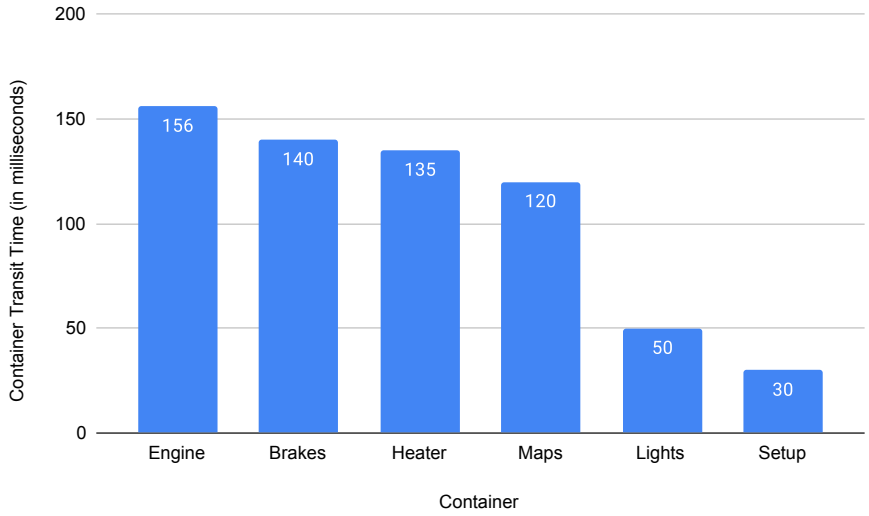

Figure 5.10: Container Transfer Time (in milliseconds) Across Nodes for Different Container Sizes: High (e.g., Engine, Brakes), Average (e.g., Heater, Maps), and Small (e.g., Lights, Setup)

presented in Figure 5.9 focused on measuring the average time in milliseconds (ms) required to move a container from one node to another within our SDV for Different Container Sizes: High (e.g., Engine, Brakes), Average (e.g., Heater, Maps), and Small (e.g., Lights, Setup), even with the presence of Containers Replicas. It demonstrates that the average migration time corresponding to the duration for which Containers Replicas are active varies between 156 ms and 30 ms which is negligible. This ensures that the container transfer time does not compromise the functionality or safety of real-time applications.

🔑 **Key findings:** The answer to **RQ1** reveals that, on average, our NSGA-III-based approach outperforms various search algorithms, including RS, IBEA, Spread, Mono-Objective GA, and NSGA-II in terms of both the quality and spread of the identified scheduling solutions in the Pareto front. Our algorithm successfully balances CPU load and memory consumption within the cluster. Moreover, it demonstrates significantly higher efficiency and scalability, proven to handle a large number of containers.

RQ2: To what extent does our container scheduling technique, based on NSGA-III, address real-world case studies related to SDVs?

To answer RQ2, ensure the sufficient coverage of sources of data, and enhance the reliability of the findings, the case studies were built with inputs from our Automotive Industrial partner. The process involved consultations and gathering feedback on the hardware architecture, software architecture, the parameters involved, and their values. The design underwent iterative improvements, necessitating the removal, addition, and modification of the different involved aspects. The draft case studies underwent peer review by automotive engineers at Ford who were not involved in the project to ensure their quality. The design of the case study involves four main stages: (i) the hardware architecture (the ECUs), their resource properties, and constraints; (ii) the software architecture, which includes the properties of the software containers, including the software they are running, their sizes, dependencies, placement constraints, and average resource consumption values; (iii) the formulation of the scheduling problem, which includes the objectives that require optimization; and (iv) the real-time constraints and requirements in the vehicle, and how to respect them. To check for the significance of the findings, a set of experiments was conducted, where the problem instances were generated using the properties identified in the following case studies.

We conducted experiments using a heterogeneous cluster made up of 3 ECUs with varying resource limitations. We used up to 100 containers based on different real-world scenarios in the context of smart SDVs, as defined in collaboration with researchers and engineers from our industrial partner. The containers are representative of real-world applications, differing in terms of CPU and memory consumption, as well as the software they deploy. In our study, we compared the default Docker Swarm scheduler, *"Spread"*, against our proposed approach. We utilized ApacheBench [94] as a stress testing tool and

cAdvisor [96] as a monitoring tool to report the status and the performance of each working node, as well as the CPU and memory usage of the cluster. To evaluate the load-balancing capabilities of our scheduling approach, we compared the resource consumption by Spread for each ECU. Several demos of the following scenarios are available on our website [4].

Scenario (1): Dynamic Load and Resource balancing. This use case focuses on analyzing a homogeneous environment, where all ECUs in the cluster have identical CPU and memory capacity. The power-saving mode has been disabled, and the distribution of the load among the nodes in the cluster is not equitable. As depicted in Figure 5.11, before executing our scheduler, the containers are distributed unevenly across the nodes in the cluster. The manager node has 30 running containers, while worker1 and worker2 have 12 and 10 containers respectively. Consequently, the load on the manager node is higher than the load on the other two nodes. The utilization of resources is outlined in Table 5.5. The manager node has only 18.6% of CPU and 14.07% of memory remaining, while the other ECUs have more than enough unused resources. Specifically, worker1 has 50.4% of CPU and 33.27% of memory remaining, and worker2 has 48.6% of CPU and 35.96% of memory remaining.

We run our scheduler to examine its distribution of containers across nodes to save CPU, memory, and power in the manager while also achieving a balanced load among the three ECUs. We compared the resource consumption and the container distribution after the execution to evaluate the efficiency of our scheduler. In this scenario, no constraints were applied; the scheduler only took into account the resource limits of each node, as well as the real-time CPU and memory consumption of each container.

The outcomes of the scheduler implementation are presented in Table 5.6. After reallocating the containers, we observed a significantly more equitable distribution of the load among the nodes compared to the previous state shown in Figure 5.11. The distribution of containers is as follows: the manager is allocated 17 containers, worker1 is allocated 17

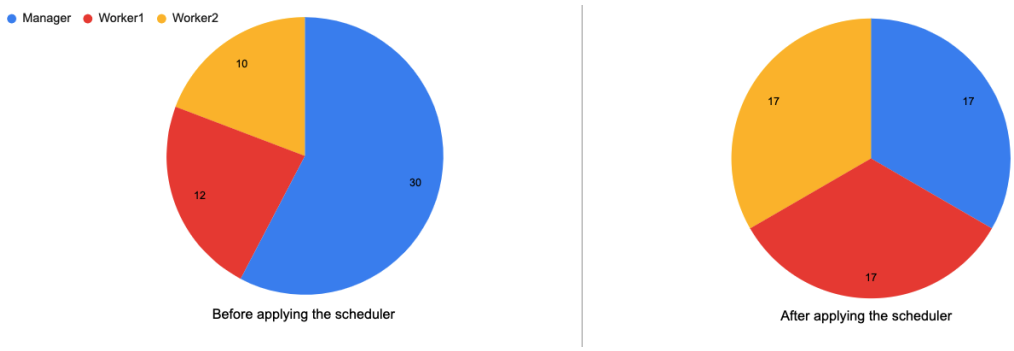

Figure 5.11: Scenario (1): Distribution of containers Per three ECUS before and after applying our scheduler.

Table 5.5: Resource consumption before rescheduling.

Node Name	Status	% Remaining CPU	% Remaining Memory
Manager	Leader	18.6%	14.07%
Worker1	Worker	50.4%	33.27%
Worker2	Worker	48.6%	35.96%

containers, and worker2 is allocated 16 containers. Additionally, both nodes have a similar percentage of remaining CPU and memory. Specifically, 15.56%, 17.67%, and 18.87% remaining CPU and 25.17%, 23.57%, and 21.39% remaining memory for manager worker1 and worker2, respectively.

Table 5.6: Resource consumption after rescheduling.

Node Name	Status	% Remaining CPU	% Remaining Memory
Manager	Leader	15.56%	25.17%
Worker1	Worker	17.76%	21.39%
Worker2	Worker	18.87%	23.57%

Key findings: Our scheduler efficiently distributes the resource consumption and load across the ECUs by considering the resources utilized by each container in the cluster, together with the available memory and CPU on each ECU.

Scenario (2): Fail Operational Use Case. This use case scenario is illustrated in Figure 5.3 and detailed in section 5.1. The leader ECU (The manager) is swapped in response to an upgrade request, a failure (e.g., bug, security attack, resource consumption over the limit, etc.), and has to be completely shut down. To address these situations, the targeted ECU should be isolated, and then replaced with a promoted ECU/node as the leader. The containers of the isolated ECU should be allocated to another designated ECU, taking into account all constraints, including container priorities, placement constraints, available CPU, memory resources, and energy per ECU. Once the isolated ECU returns to its normal behavior, the manager of the containers can rejoin the cluster and re-balance the distribution of the containers accordingly.

Before the operational failure, all three ECUs were active, containers were distributed equally (17 containers per node) and the memory and CPU consumption were balanced between the nodes as shown in Table 5.7. Figure 5.12 shows the running containers on the manager ECU before its isolation. Containers in red boxes have higher priorities compared to the remaining containers (e.g. Containers related to the Engine, brakes, ITS, etc..)

When the manager goes down, our scheduler is notified and promptly selects the next elected leader. The results are reported in Table 5.8. The manager node becomes unreachable, and worker 2 becomes the new leader. It's important to note that our scheduler utilizes the election feature provided by Docker Swarm. This last is designed to automate the re-election process, requiring no manual intervention under normal circumstances, where the nodes vote for a new leader using the RAFT algorithm [139].

We represent the new distribution of the containers in Figure 5.13. During the temporary disablement of the manager ECU, no containers are assigned to it. Notably, we see that high-priority containers with placement constraints are preserved in the final distribution with respect to their placement constraints. Containers with high priority are

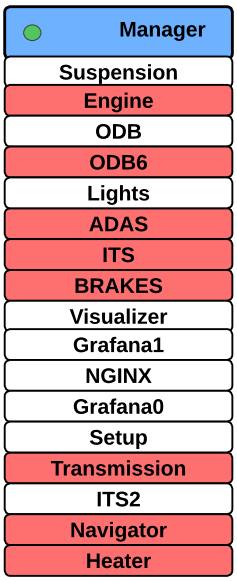

Figure 5.12: Containers allocated on the Manager ECU before rescheduling.

Table 5.7: Resource consumption before rescheduling.

Node Name	Status	% Remaining CPU	% Remaining Memory
Manager	Leader	93.06%	23.07%
Worker1	Worker	91.19%	22.98%
Worker2	Worker	94.39%	22.62%

marked in red in Figure 5.13, including those previously assigned to the previous leader (manager). For instance, the container ***Engine*** was initially assigned to the manager with high priority and could be reassigned to either the manager or worker2, given their proximity to the engine system. Following the manager's shutdown, this container continues running and is migrated to worker2. This observation confirms that both priority and placement constraints are adhered to. Only a few containers remain operating, as shown in Figure 5.13, since the resource capacities did not enable all containers in the active nodes to run. The scheduler shuts off several low-priority containers. This strategic decision retains 25 containers beyond the initial 50. As a result, resource constraints are respected, and the load between the two remaining nodes is indeed balanced, as indicated in Table 5.8. The

resources utilized by worker1 and worker2 are now similar and balanced, with worker1 consuming 75.77% of the resources compared to 80.62% for worker2, and 6.2% versus 7.3% in terms of memory and CPU resources, respectively.

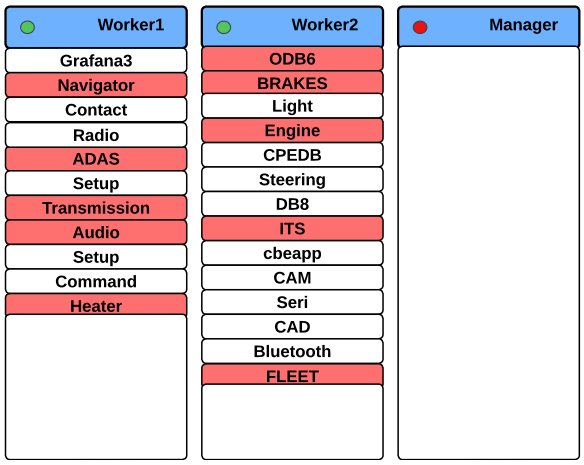

Figure 5.13: New distribution of containers.

Table 5.8: New resources consumption per node.

Node Name	Status	% Remaining CPU	% Remaining Memory
Manager	unreachable	N/A	N/A
Worker1	Worker	93.8%	24.23%
Worker2	Leader	92.4%	19.38%

Key findings: Our scheduler is able to shift containers to active nodes in the event of an operational failure or software upgrade in certain nodes while preserving the priorities and placement constraints, as well as the resource limits of the activated nodes in the cluster. After returning to normal, the scheduler returns the isolated ECU to the cluster.

Scenario (3): Power Conservation. This use case scenario, illustrated in Figure 5.2 and detailed in section 5.1, presents a real-world scenario where a vehicle in normal mode is unable to reach the driver's destination due to the available battery power is insufficient.

In this situation, the car needs to switch into a power conservation mode, and the promoted ECUs are placed in Low Power Mode (LPM) by lowering their power consumption boundaries. The same scenario applies when full memory or CPU is about to be achieved due to the large number of software applications running by the driver in the car. If the driver agrees to move the car to Power Conservation mode, the scheduler will select certain ECUs to be placed in LPM (low power mode), therefore reducing their power consumption restrictions. Additionally, the scheduler will initiate the transfer of critical applications to the designated ECUs and shut down non-critical programs. We assigned Worker3 to a Low Power Mode (LPM) and specified container placement restrictions as seen in Table 5.9. We execute the scheduler and assess its capacity to allocate the containers when a certain node is in low power mode (LPM) while adhering to the power restrictions and placement requirements.

In response, the car switches into a power conservation mode, and the promoted ECUs are placed in Low Power Mode (LPM) by lowering their power consumption boundaries. A similar scenario arises when full memory or CPU capacity is about to be reached due to the large number of software applications running in the car. If the driver agrees to move the car to Power Conservation mode, the scheduler selects specific ECUs to be placed in LPM, thereby reducing their power consumption restrictions. Additionally, the scheduler initiates the transfer of critical applications to designated ECUs and shuts down non-critical programs. Worker3 is assigned to Low Power Mode (LPM), and container placement restrictions are specified, as seen in Table 5.9. We execute the scheduler and assess its ability to allocate containers when a certain node is in low power mode (LPM), while adhering to power restrictions and placement requirements.

Table 5.9: Placement constraints for some containers in the cluster.

Containers	Manager	Worker1	Worker2
Engine	X	X	
Breaks		X	X
Audio			X
Maps	X	X	
FM		X	X
ADAS	X	X	
Heater		X	X
AC	X	X	
Contact		X	X
Setup	X		

The outcomes of this use case are displayed in Figure 5.14. Our scheduler maintained 34 out of the 50 active containers. The high-priority containers were preserved according to their specified placement constraints (highlighted in red). It is evident that worker3 received fewer containers due to its power limitations.

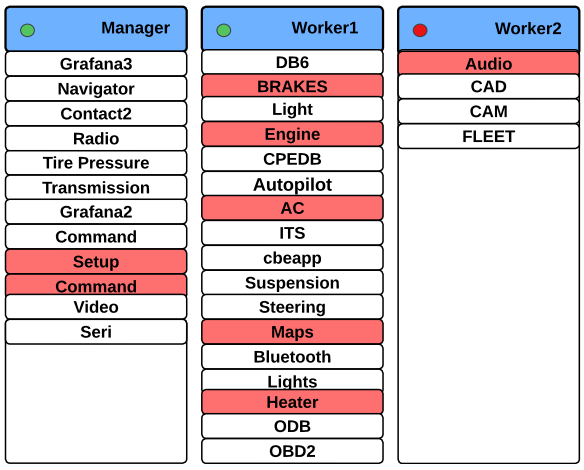

Figure 5.14: Containers distribution after rescheduling.

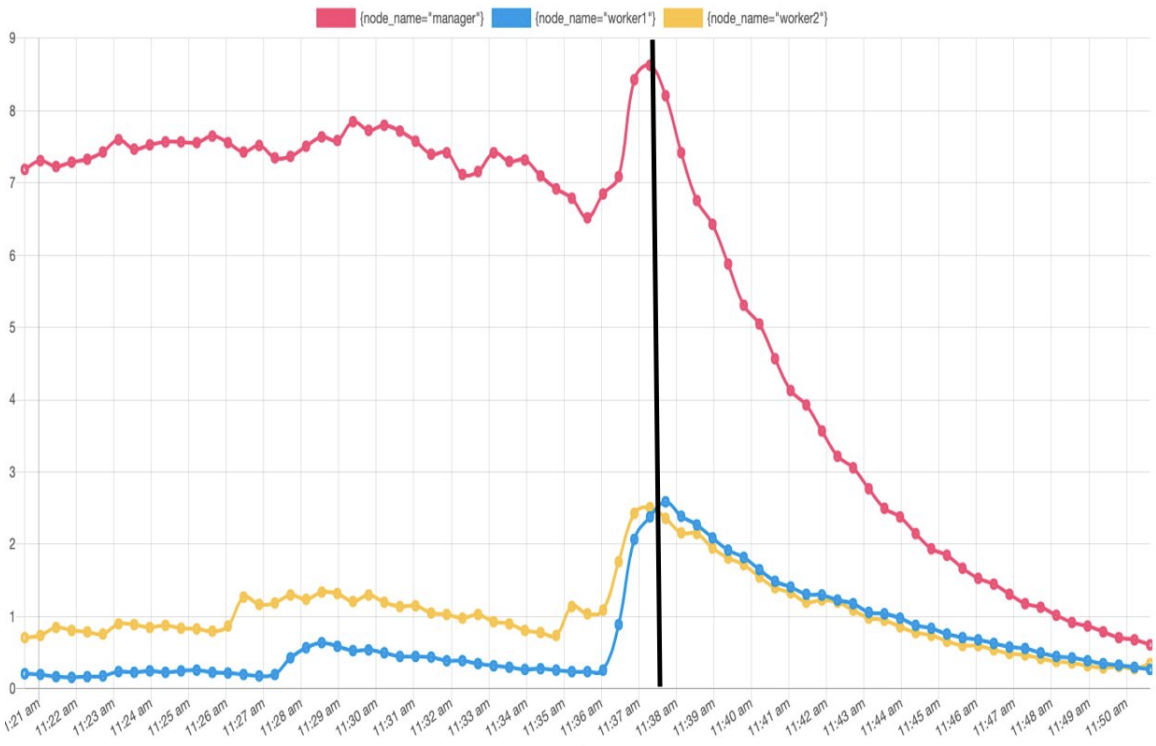

Figure 5.15: % Of CPU consumption before and after rescheduling for the 3 ECUs.

Figure 5.15 and Figure 5.16 demonstrate a notable decrease in resource utilization for both ECUs following the activation of the power conservation mode. The black line serves as a visual representation of the resource consumption before and after rescheduling.

Key findings: Once in LPM mode, our scheduler is able to manage the distribution of the container while respecting the SDV challenges such as the ECUs' power limits and the containers placements constraints.

5.4 Threats to Validity

Conclusion Validity. We addressed conclusion threats to validity by performing 30+ independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 99% confidence level (α <1%). However, the parameter tuning of the different optimization algorithms employed in our

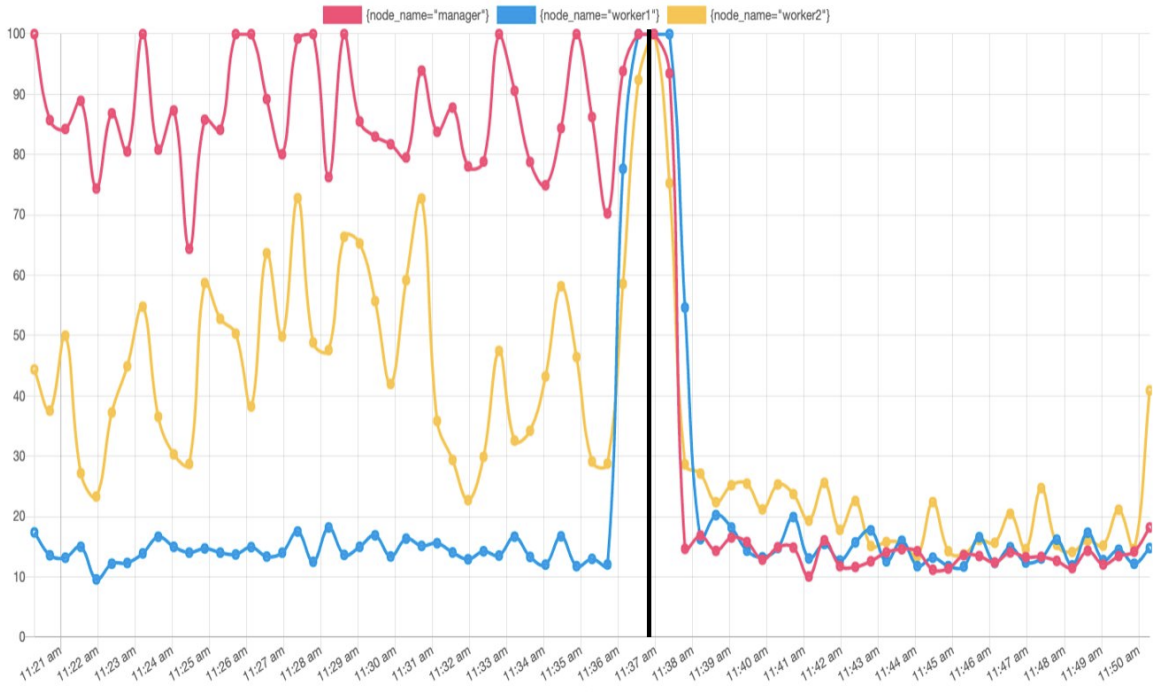

Figure 5.16: % Of Memory consumption before and after rescheduling for the 3 ECUs.

experiments poses an additional internal threat. The values for these parameters, determined through trial-and-error, align with common practices in the search-based software engineering community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [107] for our approach so that parameters are updated during the execution in order to provide the best possible performance. Finally, We did not compare our approach with other existing scheduling approaches in the literature for two key reasons. Firstly, these schedulers frequently lack the capability to grant access to their artifacts. Furthermore, they lack the necessary tools and expertise to dynamically tackle the precise automotive goals and limitations that our study is centered around. Therefore, we chose NSGA-II, NSGA-III, and IBEA for their successful adoption in other software engineering areas [54, 53].

In addition, the weights used for the objectives are selected manually and further experiments are required to study the impact of the variation of these weights on the quality of the results.

Construct Validity. To evaluate the results of our approach, we selected solutions at the knee point when we compared our approach with a fully automated scheduling approach (Docker Swarm) and the mono-objective search, but users may select a different scheduling solution based on their preferences to give different weights to the objectives when selecting the best solution. To mitigate this threat, we have to use the quality indicators of the Pareto fronts when comparing the different search algorithms.

External Validity. This concerns the generalizability of the findings. In this study, we performed our experiments on the docker swarm. However, we cannot assert that our results can be generalized to other schedulers such as Mesos by Apache [26], or Kubernetes [27]. Also, since the study was conducted in collaboration with Ford Motor Company, the results may be specific to their systems and may not generalize to other vehicle architectures or different ECUs. Additionally, the use of real-world scenarios provided by a single industry partner might not cover all possible scenarios in the automotive domain. To mitigate this threat, the process of building the use cases involved a series of consultations and refinements to ensure that the objectives were tailored to a wide array of use cases within the vehicle's ecosystem, drawing from a rich tapestry of automotive references and scheduling-related works in embedded devices. [121, 89, 90, 91, 93, 128, 68, 129, 130].

5.5 Conclusion

In this study, we propose an intelligent solution for dynamic workload balancing of docker containers across a cluster of multiple ECUs in software-defined vehicles. We employ a range of objective algorithms to optimize multiple conflicting goals including minimizing the number of nodes, intra-communications, and the usage of Memory, CPU,

and power ECU resources. Additionally, we address challenges inherent in automotive systems, such as stringent real-time requirements, energy constraints, the need for failure tolerance, security against attacks, etc. Our experiments, conducted in collaboration with Ford, on a pure Raspberry PI cluster of software-defined cars with 3 ECUs, provide evidence that our approach based on NSGA-III is able to balance the workload between the different ECUs while adapting to the many conflicting objectives, constraints, and challenges in the car. As part of our future work, we plan to generalize our results with a larger number of software containers as well as broader scenarios and configurations. We also plan to extend our approach by proposing intelligent off-board workload management on IoT devices using containers. The continued collaboration with industry partners and the sharing of research outcomes will be crucial for the innovation and adoption of these advanced container management solutions. The methodology and techniques developed in our project have broad applicability and could be effectively extended to other domains. Furthermore, extending the principles of our approach to other industries beset by similar constraints could catalyze a new wave of innovation in container management across various technological frontiers.

CHAPTER SIX

MIND THE GAP: THE DISCONNECT BETWEEN REFACTORING CRITERIA USED IN INDUSTRY AND REFACTORING RECOMMENDATION TOOLS

6.0 Introduction

Successful software lives in a perpetual state of change. Development teams continuously evolve their software in response to environmental changes like new requirements, technology changes, and updates to their library and framework dependencies. Some changes are purely functional, but many benefit from, if not require, some refactoring to be successful. Common business drivers for larger changes include the intent to reuse capabilities, migrate to new platforms like the cloud, and containerize software for DevOps. Less specific, but nonetheless important goals like improving software quality, readability, and extensibility also motivate refactoring activities. Whether explicit or implicit, a combination of these motivations drives software evolution exercises, particularly for large-scale refactorings [60].

Refactoring is restructuring software to improve its quality without altering its external behavior [28]. Whatever the motivation, developers must eventually make hard decisions about *how* to refactor code – that is, what specific refactoring changes to make and to what code [140]. Refactoring recommendation tools, a class of tools that provide its users a list of suggested refactorings (e.g., Move Method or Extract Interface) to apply, aim to assist developers with this process. Refactoring recommendation tools often focus on improvements to many issues across many files to guide a dedicated improvement effort. The specific criteria that developers apply when making such refactoring decisions and subsequent code changes has not been well studied. As such, software engineers lack insight into the relative importance of different criteria or the degree to which available refactoring recommendation tools address the criteria that influence developer decisions the most.

Developer studies consistently identify that while developers and organizations want the benefits of refactoring, they are hindered by barriers like the available tool support, fear of introducing errors, and the difficulty of the problem [140] [141]. Recent studies also confirm that developers lack refactoring support for higher-level tasks, such as feature removal or platform upgrades [142]. Such exercises are common in industry and the effort involved in them (sometimes measured in staff years of effort) demonstrates the need for successful refactoring automation to reduce costs [60] [143] [144].

Refactoring tools generally fall into three categories – tools that identify where refactoring is needed (e.g., static analyzers like SonarQube), tools that implement low-level code refactorings based on user direction (e.g., IDEs and specialized tools like ReSharper) [59], and tools that recommend refactorings for a specific goal such as to resolve code smells (e.g., JDeodorant [32]) or improve general code quality metrics [145] [74][146]. This third class, refactoring recommendation tools, serves an important purpose by providing robust decision making support to developers making refactoring decisions throughout a codebase. A developer implementing a refactoring recommendation provided by a tool is dependent on how well the tool incorporates the criteria driving the need for refactoring. However, these criteria are not commonly known.

Achieving greater alignment of developer criteria with recommended refactorings has the potential to improve industry adoption of this class of tools. However, crisp identification of the wide range of criteria that align developer intent with relevant refactoring recommendations is a non-trivial task. Generic criteria, like focusing on system-wide reduction of maintainability metrics or removal of code smells, dominate refactoring recommendation research. While making software more maintainable is a valid motivation for refactoring efforts, it is not sufficiently precise, as there are many different types of changes that can improve general notions of maintainability. Moreover, previous studies have established that large-scale refactoring efforts are driven by goals beyond a

single quality goal such as maintainability, which highlights the importance of incorporating different criteria in refactoring recommendations tools [60] [143] [144].

Consider an example from the safety-critical domain, one that requires balancing safety and performance qualities with a goal of minimizing use of shared resources. Architectures in this domain can be broadly viewed as compositions of multiple partitioned applications. It is not uncommon for developers of such systems to refactor to group similar functionality or safety criticality levels together, partitioning higher criticality elements from lower criticality elements to reduce assurance and certification costs [147]. This is one important reason to refactor that is not driven by maintainability. When it comes to making detailed decisions about how to refactor, developers need to get specific – what portions of code to change or not change, what design principles to follow, and what metrics to optimize. These questions inform and constrain the criteria that developers value.

In this study, we focus on this understudied problem of understanding the criteria that drive developer decisions about *how to refactor* their code (i.e., what specific refactorings to apply). This is an important complement to existing research on the question of *whether refactoring is needed*, which emphasizes the trade-offs around the return on investment of the effort. The goal in studying refactoring criteria is to understand the breadth of criteria that industry developers routinely apply, their relative importance, and the extent to which refactoring recommendation tools incorporate these criteria in their recommendation algorithms. Improved (and shared) understanding of these criteria is likely to accelerate tool adoption by improving their alignment with criteria favored by developers.

From October 2022 to June 2023, we extracted criteria from practitioner interviews, studied their significance with an industry survey, and studied their realization by analyzing refactoring recommendation tools. Our contributions are as follows:

- We identify 13 criteria that influence practitioner refactoring decisions, with four criteria focusing on the refactoring process and nine focusing on the qualities of the code.
- We report survey results from 142 developers.
- We identify 26 state of the art refactoring recommendation tools and assess how they support the criteria we identify.
- We share the raw survey data, survey analysis, and tool analysis in a replication package [148].

Our findings illuminate a significant gap between developers' needs and what current refactoring recommendation tools support: while all 13 refactoring criteria matter to developers, none of the refactoring recommendation tools we identified were cited as being used by survey respondents. Furthermore, only a handful of tools provide support for nine of the 13 criteria. There are significant lessons to be learned from our results for research and tool development of refactoring recommendation tools as we share in this work.

6.1 Methodology

We applied the multi-method approach summarized in Figure 6.1, combining industry interviews, a broad industry survey, and a concurrently conducted analysis of existing tools to address the following questions.

- RQ1: What criteria do developers use when deciding how to refactor code?
- RQ2: How do developers perceive the significance of different criteria?
- RQ3: Which criteria are incorporated in current refactoring recommendation tools and how does this set compare with those developers consider most significant?

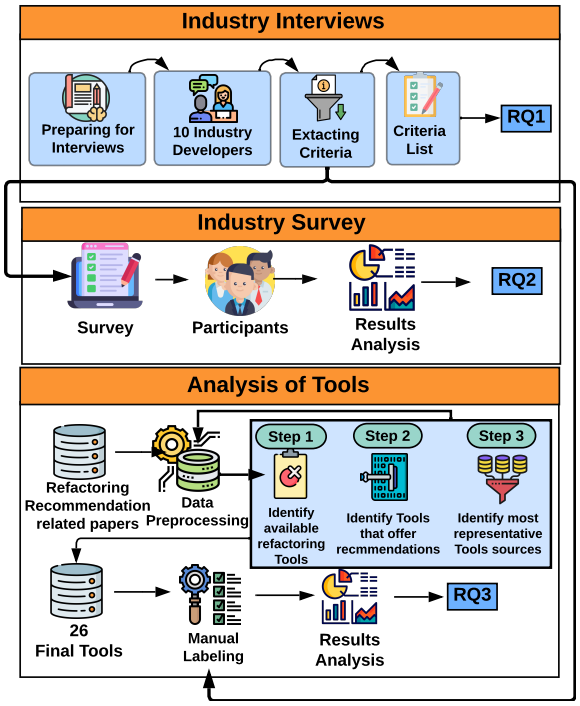

Figure 6.1: Overview of multi-method approach.

6.1.1 Industry Interviews

To discover an initial enumeration of what refactoring criteria practitioners apply when deciding how to refactor their code, we interviewed ten developers, all of whom had industry experience performing refactoring. We asked the interviewees to explain their refactoring experiences and to reflect on why they made the decisions they ultimately implemented. We stopped after ten interviews, at which point we observed saturation in responses. We recruited the interviewees from our network. Table 6.1 summarizes the interviewee demographics.

Each interview lasted approximately one hour and was attended by at least two authors. With interviewee permission, we recorded the interviews for our analysis purposes only. The interviews followed a simple structure, with the majority of time allocated to asking interviewees to recount past refactoring experiences and what criteria drove their

Table 6.1: Interviewee Demographics.

ID	Development experience	Years of experience
A	IT, Financial systems, Web apps	30
B	IT, Business systems, Web apps	25
C	IT, Safety critical systems, Embedded & transportation systems	21
D	Safety critical systems, Embedded & transportation systems	19
E	Software development tools, Mobile applications	32
F	Software development tools/Bots	7
G	IT	21
H	AI applications, Web apps	12
I	Web apps, Network applications	18
J	Web apps	13

decision making. The discussion topics of the interviews included what refactoring meant to the interviewees and specific examples of their refactoring experiences. For each refactoring experience, interviewees shared what they were trying to accomplish, what tools they used, what criteria drove their decisions, and what challenges they faced.

Three co-authors reviewed all interviews and independently extracted concepts from each interview as candidate refactoring criteria. We conducted a categorization and consolidation exercise to distill the concepts. Given the varied expressions of several criteria from different interviewees, we aggregated the commonly considered criteria and refined the criteria labels iteratively. For example, we associated Duplicated Code with Code Size, Resource Scaling with Cost/time, and considered Modular Code and Clean Code as aspects of the Design Principles criterion. In addition, we refined the scope of certain criteria as we uncovered related variations of it in the interviews. For instance, we generalized Module Dependencies to Dependencies to include dependencies among classes within modules in addition to dependencies among modules and refined our definition accordingly. We excluded discussions on deciding whether (as opposed to how) to refactor and comments that lacked specificity (e.g., "code should be maintainable").

This exercise resulted in a set of 13 refactoring criteria described in detail in Section 6.2. These criteria were defined in a codebook that was used to label survey and tool data, connecting developer views of criteria importance with tool support for the same criteria. Our replication package includes the codebook and a table enumerating examples of how different criteria were mentioned by interviewees or in the sources we analyzed for tools analysis [148].

6.1.2 Industry Survey

We designed a survey with two broad goals – validating the criteria distilled from the interviews (including identification of missing criteria) and learning the significance of the 13 refactoring criteria from a broader population of industry developers. Following survey design best practices, we conducted a series of iterative pilot surveys with a representative population and refined the questions until reaching saturation [149]. This process resulted in 13 questions, of which eight were multiple-choice and five were open-ended responses [148]. These questions addressed respondents'

- *background:* years of experience, type of organization, and role in their organization (three questions).
- *refactoring experience:* their definitions of refactoring, if they consider specified scenarios to be examples of refactoring, how often and what programming languages they refactor, what tools they use to refactor, and how often different activities trigger refactoring (six questions).
- *views on refactoring criteria:* descriptions of what refactoring criteria they use, how significant they perceive each of our 13 criteria to be, and explanations of why they consider specific criteria to be significant (four questions)

The survey design employed two approaches to ensure collection of reliable data. First, prior to revealing the 13 criteria in the survey, we used an open-ended question to ask what criteria respondents consider when choosing specific refactorings. Second, in a later question, we provided descriptions of the 13 criteria and asked respondents to rate each criterion on a four-value scale from Not Significant to Very Significant. We used the first question to get an unbiased view and to discover any refactoring criteria that had not emerged from the interviews. We used the second question to obtain a level comparison among the 13 criteria.

Three co-authors labeled the responses to the open-ended question about criteria using the codebook [148] and conducted categorization and consolidation exercises as needed. The goal of this labeling exercise was to assess how the mentioned criteria fit into one of the 13 criteria we identified and whether any missing criteria were mentioned.

We distributed the survey to members of the software engineering community via email (dlists: 8,200), LinkedIn (subscribers: 21,401), Twitter (followers: 9,102), industry colleagues that included our interviewees and their networks, and company internal technical interest groups through our collaborations. The online survey was open between March 13 and June 12, 2023, and yielded 142 responses, 106 of which are 100% complete responses. Of the complete responses, the respondents report

- 56% have more than ten years of experience and 23% have 3-10 years of experience.
- 75% work as software engineers, system engineers, or software architects.
- 76% have spent most of their career in industry or contractor organizations.

We summarize the analysis of these responses in Section 6.3.

6.1.3 Analysis of Tools

To address our last research question and identify which of the 13 criteria are supported by current tools, we analyzed software refactoring recommendation tools. A Google search for refactoring recommendation tools resulted in a variety of tools that identify where refactoring is needed, recommend specific refactorings, and implement refactorings. While all play a part in the refactoring process, most research and commercial tools support only a subset of these activities. In focusing on tools that make specific refactoring recommendations, we excluded the following categories of tools

- Tools that primarily identify where refactoring is needed (e.g., static analyzers like SonarQube). While such tools may offer generic advice (e.g., reduce coupling between classes), we did not consider such advice to be specific refactoring recommendations.
- Tools that only implement refactorings based on user direction (e.g., IDEs like IntelliJ, Eclipse, VisualStudio).

To identify the tools that met our study criteria, our selection process used the following three steps.

Step 1: Identify potential refactoring recommendation tools. To identify tools that recommend refactorings, we conducted a targeted search in Google Scholar using keywords such as *Refactoring recommendation tool*, *Search-based refactoring*, and *Rule-based refactoring*. The search yielded a combination of academic publications and websites, including websites for commercial tools. For the academic publications, we employed both backward and forward snowballing techniques. Backward snowballing involved examining the references cited within the selected publications and forward snowballing entailed analyzing publications that cited the publications within our initial

pool. This comprehensive snowballing approach ensured that we explored both earlier and more recent works. After removing duplicates, this search process resulted in 104 resources. This entire list is included in our replication package [148]. The authors then reviewed each of the sources and documentation of the tools where relevant to ensure that a tool's description included the basis (e.g., criteria incorporated and specific metrics) for its recommendations. 13 sources were eliminated in this step.

Step 2: Check whether tools offer specific refactoring recommendations. Not all of the sources collected in Step 1 described a concrete refactoring recommendation tool. To qualify as a refactoring recommendation tool, each source had to satisfy the following conditions:

1. The tool must provide recommended changes (as opposed to only identifying where refactoring is needed).

2. The recommendations must be for refactoring code (as opposed to other artifacts such as architecture models).

3. The recommendations must be unambiguous (as opposed to requiring developers to interpret vague suggestions).

Let's review some examples of our selection process. Dijkman et al's study [150] presents a technique that is used to identify the locations of refactoring opportunities based on three similarity measures. The authors, however, do not advocate any particular refactoring recommendations that should be applied, therefore this work was disqualified under condition (1). The work of Alkhazi et al [151] does provide refactoring recommendations, but was disqualified under condition (2) as it focuses on refactoring models instead of code. Many industry tools, such as SonarQube and CodeGuru, have a place in the overall refactoring process as they identify refactoring opportunities and provide

abstract recommendations (e.g., "refactor the class into smaller ones which focus on well-defined topics"). However, these recommendations are open to interpretation and, as such, were disqualified under condition (3). All commercial tools were disqualified from this study by one or more of these conditions.

Step 3: Identify the most representative source(s) to evaluate the tools. In several cases, steps 1 and 2 resulted in multiple sources that described the same tool. As our primary objective is to assess each tool comprehensively, we consolidated these sources where feasible. During this process, we identified the source that provided the most comprehensive insights into each tool's implementation decisions, specifically focusing on the criteria that influence its refactoring recommendations. An example of this effect is JDeodorant [32]. The initial search yielded seven sources that explained different perspectives of the tool [152][153][154] [32, 155, 156, 157], each of which provides a subset of the criteria the tool employs. For our analysis, we extracted criteria from multiple sources but cite [32] as this paper incorporated the most relevant details for identifying criteria JDeodorant supports.

This rigorous selection process yielded 26 distinct tools for analysis out of 104 sources. To improve reproduciblity of this study, the replication package identifies which source and tool were filtered in each step of our selection process. Using the same codebook and labelers from our survey labeling, we labeled each tool in terms of what criteria each includes in making its refactoring recommendations. For each tool, the labelers reviewed published material for each tool, commonly basing labels on fitness functions and metrics on which rules are based to generate recommendations. This exercise resulted in a list of refactoring recommendation tools and an enumeration of which criteria are included in their refactoring recommendations that we present in detail in Section 6.4.

Table 6.2: Criteria for that focus on the refactoring process.

Criteria	Description
Change Size	Favors making fewer refactoring changes or smaller, incremental refactoring changes
Cost/time	Favors refactoring changes that require less cost/time to perform
Propagation of Changes	Favors refactoring changes that affect fewer code elements and impact fewer teams (e.g., preserving interface contracts)
Unchangeable Code	Favors refactoring changes that avoid specific code (e.g., avoiding changes that require expensive certification)

6.2 RQ1: Refactoring Criteria

To answer **RQ1**, we interviewed ten industry developers and performed the categorization and consolidation exercise described in Section 6.1. Tables 6.2 and 6.3 summarize the codebook of the 13 criteria resulting from this analysis. The first list includes four criteria that focus on the refactoring process itself, including criteria like the Cost/time required to implement recommended refactorings. The second list includes nine criteria that focus on the qualities of the code being refactored, including criteria like Code Complexity and Readability.

Undeniably, there are relations among these criteria as the code properties can influence multiple qualities. For example, code that is less complex (Code Complexity criteria) is also generally more readable (Readability criteria). To avoid double counting, we adopted an interpretation in which each quality (e.g., complex logic or highly entangled code) would map exclusively to the most specific Criterion (e.g., Code Complexity or Dependencies) rather than to multiple criteria (e.g., also mapping to Readability and Design Principles). As such, the interpretation of a general criterion like Readability is qualities relating to Readability that are not captured by other criteria. See Section 6.5.2 for more discussion on this issue.

Table 6.3: Criteria that focus on the qualities of refactored code.

Criteria	**Description**
Code Complexity	Favors code with less complex units (e.g., classes or methods with acceptable cyclomatic complexity)
Code Size	favors a smaller code base (e.g., removing dead code or lifting common code to a superclass)
Dependencies	Favors code with fewer dependencies among units (e.g., introduction of abstraction layers)
Design Principles	Favors code following design principles not addressed by other criteria (e.g., design patterns or SOLID principles)
Ease of Testing	Favors code that is easier to test (e.g., easier to insert mocking to improve unit testing)
Extensibility	Favors code that is easier to extend over time (e.g., using interfaces or abstract classes)
Performance	Favors changes that use less computing resources (e.g., use of processing, bandwidth, or storage)
Readability	Favors code that is easier for any developer to understand (e.g., use of meaningful names)
Well-defined APIs	Favors code with well-defined interfaces that expose only what is necessary (e.g., private-by-default)

The distribution of refactoring criteria across interviewees indicates common use of many of the 13 criteria. Eight of 13 criteria were mentioned by all interviewees. All criteria (with the sole exception of Ease of Testing) were used by more than half of the interviewees.

The caveat to these numbers is the importance of context. Most interviewees brought up the role of context in driving the match between specific criteria and the problem being addressed through refactoring. In summary, developers draw from a broad range of refactoring criteria in their experiences and each criterion we identified appears to be commonly applied in practice.

Key findings:

Finding 1: When deciding how to refactor code, developers employ a variety of criteria. We identified 13 distinct refactoring criteria. All interviewees use at least eight of the 13 criteria.

Finding 2: Eight of the 13 the refactoring criteria are used by the majority of the developers. Ease of Testing was the only criterion mentioned by fewer than half of the interviewees.

Finding 3: All interviewees emphasized the importance of context in selecting which criteria to apply for a given refactoring exercise.

6.3 RQ2: Significance of Criteria

To answer **RQ2**, we issued the survey described in Section 6.1, which directly asked respondents to rate the significance of the 13 refactoring criteria identified through practitioner interviews. The survey results suggest that *all* of the 13 criteria matter to the developers responding to our survey. This finding suggests that we were able to identify criteria that are relevant to developers. However, there are important nuances.

Prior to revealing the 13 refactoring criteria, the survey used an open-response question to ask "What criteria do you consider when choosing specific changes to make as you refactor your code?" Figure 6.2 shows the results of labeling these responses, dividing them into three categories of criteria. The top two bands reflect the criteria for the refactoring process and the criteria for code quality, corresponding to the descriptions in

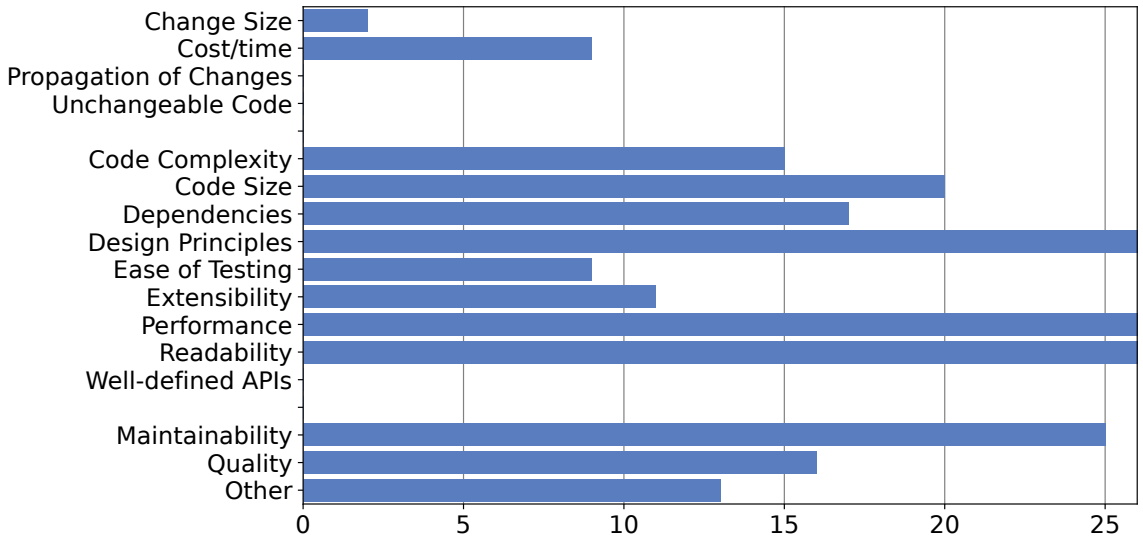

Figure 6.2: Number of survey respondents (n=115) mentioning each refactoring criterion in open response comments.

Tables 6.2 and 6.3. The third band contains mentions that did not fit into our initial 13 criteria.

Maintainability was mentioned by many respondents, however, it is a very broad notion that is already supported by many of our criteria (e.g., Code Complexity, Dependencies, Design Principles, and Readability). We regardless created a new label for Maintainability due to high number of mentions (n=25) rather than arbitrarily assume a more precise meaning or map the idea to a superset of criteria that it could imply. *Quality* was likewise mentioned by many (n=16), but is also such a broad notion that it could plausibly map to many of our criteria. As with Maintainability, we created a new label to specifically capture references to quality. *Other* includes refactoring criteria that do not overlap with the 13 criteria based on the interviews, Maintainability or Quality categories. The most commonly mentioned criteria included in Other were Architecture Conformance (n=5) and Security (n=3). Reliability, Reconfigurability, and Traceability were each mentioned once.

The number of spontaneous mentions through open-response answers can be considered a soft proxy for the importance of criteria. From these open responses, the criteria for code quality (the middle band in Figure 6.2) are mentioned much more commonly than the criteria for the refactoring process (the top band). Mentions of broad categories like Maintainability and Quality, both of which align with criteria for code quality, were also mentioned more commonly than criteria for the refactoring process.

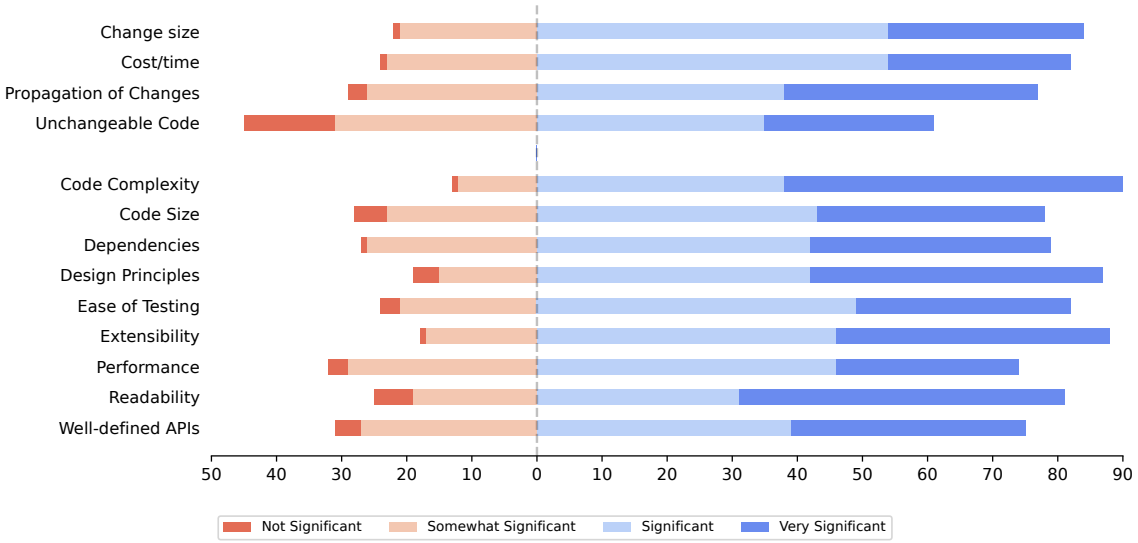

Figure 6.3: Survey responses on the significance of refactoring criteria, showing counts of respondents (n=106) by category.

Figure 6.3 shows the results of asking respondents to explicitly identify the significance of the 13 refactoring criteria along the Not Significant to Very Significant scale. Unlike the open response answers, respondents indicated much greater parity between the significance of criteria for the refactoring process and criteria for code quality after being exposed to all criteria.

The Unchangeable Code criterion is the least significant in these results, which is unsurprising as it is a concern more commonly encountered in regulated domains in which

code changes must undergo expensive approval processes. Even so, only 13% of respondents indicated that this criterion was Not Significant. Excluding this outlier, even the least significant criterion (Performance) was deemed Significant or Very Significant by 70% of respondents and only 1% to 6% of respondents indicated that other criteria were Not Significant.

So, all criteria matter, but surely not all the time? While the survey did not ask this directly, multiple open responses matched interviewee comments that context of a refactoring exercise dictates which criteria are most significant. The following survey comments are illustrative of this point:

- *I approach each refactor with a specific goal dependent on the target project and its deficiencies. There's no "one reason fits all" rule for refactoring.*
- *Really depends but improving a fundamental characteristic (readability, cohesion, coupling, modularity, speed, . . .) while minimizing the risk of changing behavior is common.*
- *Depends on the need/motivation of the task. Often it is in service of upgrading to newer versions of underlying languages or libraries, but it may be for readability, maintainability, performance, etc.*

Key findings:

Finding 4: All 13 refactoring criteria matter to survey respondents. The perception of a criterion as Significant or Very Significant ranges from 58% for Unchangeable Code to 88% for Code Complexity and averages 75.5%.

Finding 5: Architecture Conformance and Security are candidates to add to the refactoring criteria in Table 6.3.

Finding 6: Respondents commonly mentioned imprecisely defined concepts like Maintainability and Quality that could be plausibly mapped to multiple refactoring criteria.

6.4 RQ3: Criteria Support in Tools

To answer **RQ3**, we analyzed which refactoring criteria are supported by current refactoring recommendation tools. We used the steps described in Section 6.1 to arrive at the collection of 26 tools shown in Table 6.4. Of note, our survey included a question asking respondents to list tools that they use to assist in refactoring. No respondents mentioned using any tools that meet our criteria for tools that generate specific refactorings. There are many potential explanations, such as lack of awareness of available tools, lack of tools for specific programming languages, and perception that the tools do not make suitable recommendations.

Table 6.4 shows the results of identifying which of the 26 tools incorporate which criteria through studying the sources and mapping to the criteria using our codebook and the same labelers, along with a count of tools supporting each criterion. The Dependencies criterion is the most widely supported, being supported by 65% of the tools we identified. The next most commonly supported criteria are Design Principles and Code Complexity, with each being supported in some way by 50-60% of tools. No tools support the Propagation of Changes, Unchangeable Code, or Performance criteria. Further, few tools

Table 6.4: Refactoring recommendation tools and the refactoring criteria that each supports.

No.	Tool ID	Change Size	Cost/time	Propagation of Changes	Unchangeable Code	Code Complexity	Code Size	Dependencies	Design Principles	Ease of Testing	Extensibility	Performance	Readability	Well-defined APIs
T1	Griffith et al. [73] - Truefactor					X		X	X				X	
T2	Morales et al. [158] - TARF					X			X	X				
T3	Tairas et al. [77] - A-CMA					X		X	X					
T4	Vimaladevi et al. [159]					X		X			X			
T5	Han et al. [160]	X						X	X					
T6	Ivers et al. [161]		X				X	X						
T7	Abid et al. [75] - Refbot					X		X	X		X			X
T8	Di Penta et al. [162] - Evolution Doctor						X	X						
T9	Tsantalis et al. [163] - JDeodorant					X	X	X	X		X			
T10	Ouni et al. [164]	X				X		X	X					
T11	Nyamawe et al. [165]					X		X	X					
T12	Cui et al. [166] - RMove							X						
T13	Lin et al. [167] - Refactoring Navigator							X	X					
T14	Moghadam et al. [168]													
T15	Terra et al. [169] - Archfix								X					
T16	Terra et al. [170] - JMove								X					
T17	Ujihara et al. [171] - c-JRefRec							X	X					
T18	Steidl et al. [172]					X	X							
T19	Yoshida et al. [173]						X							
T20	Kessentini et al. [174] - DINAR	X				X		X	X		X			
T21	Vidal et al. [175] - Bandago					X								
T22	Niu et al. [176]					X	X	X			X			
T23	Kurbatova et al. [177]													
T24	Bavota et al. [178]							X	X					
T25	Rahman et al. [179] - MMRUC3							X	X					
T26	Szoke et al. [180] - FaultBuster					X	X						X	
Total Count Per Criteria		**3**	**1**	**0**	**0**	**13**	**8**	**17**	**14**	**1**	**5**	**0**	**2**	**1**

provide support for any of the criteria that focus on the refactoring process. Finally, no tool supports more than five of the original 13 refactoring criteria.

A developer who is concerned with multiple refactoring criteria falls into one of these categories of use (assuming programming language support, which is an entirely different topic), shown in descending order of expected utility:

- All of the criteria they care about are supported by a single tool.
- All of their criteria are supported, but not by the same tool. Developers can use multiple tools and manually integrate the recommendations, but there is little evidence to suggest that this will be easy or productive.
- Some of their criteria are supported by tools, but others are not. Developers can use a tool to get good but incomplete advice, relying on their own judgment to refine or decline recommendations based on missing criteria.
- No tools support their criteria, and developers will have to come up with their own refactoring recommendations.

Key findings:

Finding 7: No survey respondents reported using refactoring recommendation tools that met our criteria for tools that generate specific refactoring recommendations.

Finding 8: Nine criteria are supported by five or fewer of the 26 recommendation tools. Only four tools support any of the criteria that focus on the refactoring process.

Finding 9: No refactoring recommendation tools support more than five of the 13 refactoring criteria.

6.5 Closing Tool Gaps

The ten interviews, our survey, and the analysis of tools mapping to the survey results collectively demonstrate that there is a significant disconnect between the criteria

that developers value and the criteria that refactoring recommendation tools employ. In this section, we discuss implications of our findings and present recommendations focusing on gaps in the breadth of criteria supported by tools, the transparency of criteria supported by tools, and the ability to capture context that should inform refactoring recommendations. As the collection of recommendation tools available today fall short of developer needs and most of the existing tools are currently in research stage, we address these recommendations to the researcher and tool vendor communities who can best close these gaps.

6.5.1 Breadth of Criteria

The majority of survey participants rated all four criteria for the refactoring process (58% to 78%) and all nine criteria for the quality of refactored code (70% to 88%) as Significant or Very Significant (Figure 6.3). However, Table 6.4 shows that many of these criteria are supported by few tools that recommend specific refactorings. None of these tools support Propagation of Changes, Unchangeable Code, or Performance. Cost/time and Well-defined APIs are each supported by only one tool. Overall 0% to 65% of the tools address the nine criteria for the quality of refactored code (with Dependencies being the most commonly supported criterion) and 0% to 11% of the tools provide some support for the four criteria for the refactoring process (with Change Size being the most commonly supported criterion at only three tools). Figure 6.4 contrasts these perspectives, clearly showing the gap between what recommendation tools support and what developers want.

The following survey quotes illustrate developer views of three of the criteria exhibiting particularly large gaps

- *Cost/time is a consideration in every single project I have worked on* [only one tool supports Cost/time]

- *Making fewer and smaller changes (Change size) makes it easy to test and understand the impact of what you are doing.* [only three tools support Change Size]

- *Propagation of changes is important, because if what you do affects others, then you are likely to encounter pushback as the change becomes larger due to its impact. This must be considered carefully when deciding what to refactor.* [no tools support Propagation of Changes]

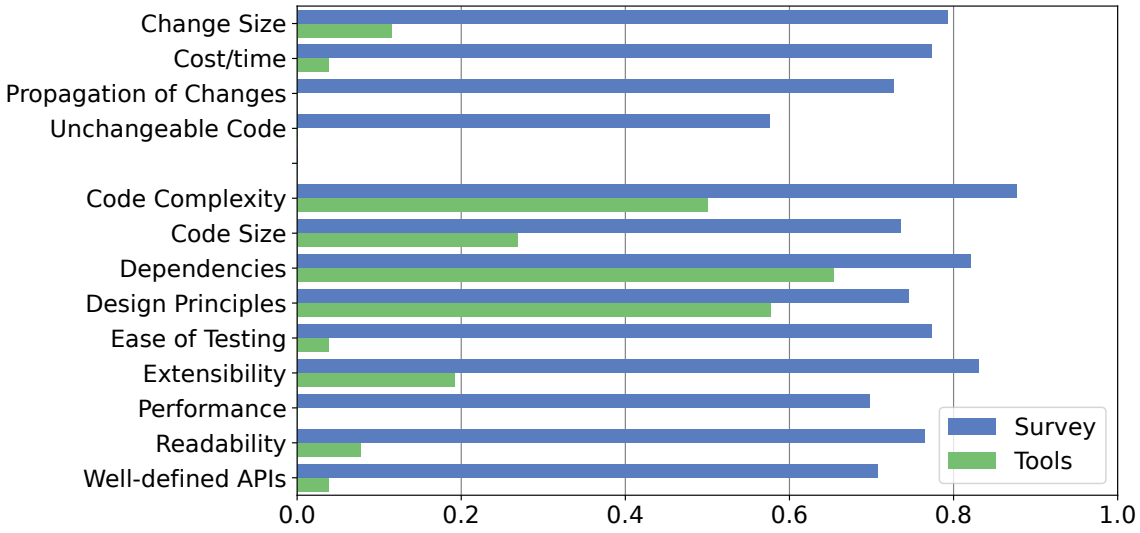

Figure 6.4: For each criterion, portion of survey respondents (n=106) identifying it as Significant/Very Significant against portion of refactoring recommendation tools (n=26) supporting it.

As our survey reveals, developers care about multiple criteria that are often not supported in the same tool (as illustrated in Section 6.4). Though no respondents reported using refactoring recommendation tools that generate specific recommendations (Finding 7), the propensity for tools to support only a subset of criteria leaves potential users with complex decisions, ranging from having have to use multiple incompatible tools, relying on lengthy and error prone manual approaches, or not addressing the criteria at all. New or

improved recommendation tools that support a broader range of refactoring criteria could remove this barrier to adoption, taking on what is currently a user burden to reconcile competing or incomplete refactorings.

Recommendation 1: Create more flexible tools and algorithms that support a broader range of refactoring criteria in a single tool.

Recommendation 2: Enrich the representations that tools use to support more diverse criteria. For example, annotations that identify code that is subject to expensive certification processes would enable Unchangeable Code and annotations that identify externally facing APIs would enable Propagation of Changes.

Recommendation 3: Create cost models to estimate the work required to implement and test refactoring recommendations, enabling Cost/time trade-offs in recommendation generation.

6.5.2 Transparency of Criteria

While Table 6.4 shows that many recommendation tools address several of the more common criteria (e.g., Code Complexity, Dependencies, and Design Principles), careful study shows that they do so in different ways. These subtle differences in how tools address the same criterion create lack of transparency and consistency in how each criterion is treated, complicating decision making for developers. Should they treat each tool as comparable or become experts in the underlying metrics and approaches in order to select an appropriate tool? For example, each of our noted tools treat the Dependencies criterion in one of the following categories.

1. *Optimize a direct measure of Dependencies.* Ivers et al use a coupling measure along modularization boundaries as an objective in multi-objective optimization [161].

2. *Optimize an aggregate measure that includes measures of Dependencies.* Abid et al and Kessentini et al use QMOOD measures, among which are aggregates that include a count of different classes that a class relates to (DCC) [75][174]. Lin et al use an aggregate that includes coupling between objects (CBO) [167]. Ujihara et al use an aggregate measure that includes the number of dependencies (reads, writes, calls), the number of classes depending on a class, and the number of classes whose members are used by a class [171]. Each of these aggregate measures includes measures of something other than Dependencies.

3. *Minimize a count of code smells.* Griffith et al, Niu et al, and Ouni et al all aim to reduce counts of code smells, though they address varying code smells [73] [176] [164]. Code smells like feature envy have a clear link to Dependencies and resolution should improve dependency measures in a code base as a by product. However, success is not measured based on dependency metrics. It is possible to argue that by reducing code smells, these tools address more important (or at least more focused) problems than approaches that opportunistically improve Dependencies across a code base.

4. *Use Dependencies to parameterize refactoring recommendations.* Bavota et al, Nyamawe et al, Rahman et al, and Tsantalis et al use techniques like Jaccard similarity (based on Dependencies, among other factors) to parameterize recommendations (e.g., determining which class to move a method to or which members to include in an extract class refactoring) [178][165][179][163].

Dependencies is an illustrative example, as across these approaches there are clear differences in the phenomena being measured (e.g., whether a metric is based only on method calls and field references or whether it also includes inheritance). This variation reflects a lack of community consensus on the best metrics for a largely consistent concept.

Design Principles provides a different kind of example. This is a broad criterion, aggregating distinct concepts such as design patterns, code smells, and principles like DRY and SOLID. Tools supporting this criterion generally support different aspects of Design Principles, which implies a need for developers to understand this more granular view to select appropriate tools.

Recommendation 4: Clearly describe the criteria used by tools in making recommendations (transparency in terms easy for developers to consume).

Recommendation 5: Determine which metrics best align with developers' preferences and refactoring outcomes. Use these as the basis for recommendations (transparency in repeatable assessment of criteria and recommendations).

Recommendation 6: Assess and fill gaps in criteria support (e.g., unaddressed code smells).

6.5.3 Capturing Context

While survey results show that all criteria are important, interview and survey comments both indicated that context plays a role in determining which subset of criteria matter for a specific refactoring exercise. Some criteria may have more consistent prominence (e.g., Cost/time is mentioned as almost always important by some) and others may be more dependent on code being refactored (e.g., Unchangeable Code applying more often to code that undergoes expensive certification practices). In other cases, the initial qualities of the code being refactored (e.g., how readable or complex code already is) might be part of the context that determines what kinds of improvements are most useful and what criteria should drive recommendations.

Tools can incorporate context in different ways, such as

- Ask users to provide explicit context (e.g., select which criteria and/or portions of code should be improved).

- Provide alternate suggestions that reflect different trade-offs among criteria and allow users to choose.
- Elicit user feedback on recommendations and use this to generate new recommendations.

These mechanisms integrate user preferred criteria at different points in the refactoring process, such as before (a priori) or after (a posteriori) generating recommendations [72]. There are trade-offs among these mechanisms that involve the attention and expertise required of users, the degree to which criteria improve recommendations, and the compatibility with the algorithms used to generate recommendations (e.g., the performance of some algorithms degrades quickly with the number of objectives). Further research is needed to find a balance among these mechanisms that fits with developer workflows and yields more satisfactory results, from both quality and time commitment perspectives.

Recommendation 7: Allow users to express which criteria they value (a priori) and hence should drive recommendation generation.

Recommendation 8: Allow users to set thresholds for measures that are "good enough" and need not be further optimized, allowing tools to focus recommendations on refactorings that provide greater value.

Recommendation 9: Provide transparency regarding the expected benefits of recommendations to improve a posteriori review (e.g., details of smells removed or degree of improvement in any measures).

Recommendation 10: Develop techniques to use developer feedback on recommendations to refine criteria that drive subsequent recommendation generation (e.g., relative importance of criteria or missing important criteria).

6.6 Threats to Validity

Our threats to validity include the following:

Internal Validity. Our interviews, analysis of survey responses, and tool analysis represent a potential threat to internal validity. To mitigate this threat and to ensure the reliability of our qualitative findings, we implemented and consistently adhered to established guidelines and best practices for conducting qualitative research, including comprehensive data use, constant comparison, use of tables, and refinement of codes through adjudication and investigator triangulation. We used the same three labelers and codebook for survey responses and tool analysis. Tool labeling is limited by what is exposed in literature. A positive label does not imply fully developed support and there are many ranges of functionality in each tool from naive to sophisticated.

External Validity. Our findings are based on the data we collected from ten interviews, 142 survey respondents, and 26 tool sources over a specific period of time. New research and tools may affect findings and recommendations. For example, one investigation that was published after our study period, EM-Assist, focuses on generating refactoring recommendations by combining large language models (LLMs) and static analysis [181]. The scope of this study is on improving recall of correct Extract Method recommendations to refactor Long Methods, demonstrating 6.4x recall rate improvements over its peers. While our tool data does not include EM-Assist, our findings are still valid as EM-Assist addresses a single code smell (Long Method) that is addressed by other tools in our study. We do not make a generalizability claim, but position our findings as outcomes supported by our data and existing research literature.

Conclusion Validity. We interviewed ten industry practitioners, stopping when we observed convergence on criteria mentioned and no new criteria emerging. We distributed our survey to a broad audience to collect the most relevant data for our goals. To ensure that we asked the right questions and to avoid introducing our own biases into the wording and

selection of questions, we conducted a series of iterative pilots to identify and address shortcomings in the survey design. Furthermore, we included several open-ended questions to allow participants to share their views and experiences. Tools identification is limited by what can be accessed through publicly available resources.

6.7 Conclusion

All of the 13 refactoring criteria originating from developer interviews are important to practitioners, at least contextually. However, the 26 refactoring recommendation tools that we identified as providing specific recommendations are all research tools and favor a small subset of these criteria (only three criteria are addressed by at least one-third of tools and nine criteria are supported by five or fewer tools). While some of these tools have some industry penetration (e.g., JDeodorant), none were identified as being used by our survey respondents. One explanation is that developers simply are not aware of these tools, but awareness alone would not solve the wide gap between the criteria that developers value and the criteria that these tools support. The gap is too large to ignore.

In this study, we provide evidence of this gap, focusing on the criteria that developers care about when making refactoring decisions and how existing refactoring recommendation tools support these criteria. A significant portion of the responsibility for closing this gap rests on the shoulders of researchers and tool vendors. Our findings and recommendations suggest focusing on incorporating more of the 13 criteria we identified from interviews, investigating the addition of criteria (e.g. Architecture Conformance and Security) revealed by our survey, normalizing the diversity within how the criteria are evaluated, and incorporating features to elicit and use more of the context as fruitful starting points to generating better refactoring recommendations.

CHAPTER SEVEN

BUILD CODE NEEDS MAINTENANCE TOO: A STUDY ON REFACTORING AND TECHNICAL DEBT IN BUILD SYSTEMS

7.0 Introduction

Build systems are responsible for transforming source code into executable programs by coordinating the execution of various tools, ranging from compilers to code analyzers [182, 183, 184]. Build systems, such as Maven [18], Ant [19] and Gradle [20] are commonly employed in the development of large software projects to automate the process of compiling, packaging, and testing software products. However, with this wide range of capabilities and flexibility, a lot of complexity can emerge in build code. Indeed, prior research [21, 22, 23, 24] has demonstrated how configuring build systems can frequently lead to challenges in their maintenance and result in delays in software development projects. Seo et al. [23] demonstrated that up to 37% of builds conducted at Google experience failure, while Kumfert et al. [24] further estimate that build maintenance imposes a 12% overhead on the development process, distracting developers from their main tasks. This highlights the importance of a robust, clean, and well-maintained build system to facilitate seamless development tasks, thereby preventing them from becoming arduous and time-consuming, consequently affecting the overall efficiency of the software.

Refactoring, defined as the restructuring of existing code to improve its quality without altering its outward behavior [13], is frequently proposed as a method to achieve this objective. However, Refactoring build systems, despite their importance in software development, remains an area with limited understanding [42]. To the best of our knowledge, no empirically validated taxonomy of build refactorings currently exists. Moreover, no analysis has been conducted to connect build refactorings to the technical debts they may mitigate. Technical debt (TD) is a metaphor that describes the lower-quality

code, which represents a trade-off between the short-term benefits of rapid delivery and the long-term value of software [185].

Currently, practitioners have limited access to specific guidance on how to apply build-related refactorings and organize their build code. Knowing the types of refactorings and TDs that may occur within build systems can shape developer guidance when it comes to maintaining their existing build files and can support the development of tools that can automatically detect and recommend refactoring opportunities. To highlight the relevance of refactorings in build files, we provide the example in Listing 7.1, which demonstrates a Don't Repeat Yourself (DRY) refactoring applied to a Gradle build file, to address Code Duplication technical debt. Here, 'Connect' and 'Insert' tasks are streamlined using a loop, which reduces repetitive task definitions, thus minimizing redundancy by avoiding repetitive setups for each task, making the code more maintainable.

```
['Connect', 'Insert'].each { taskName ->
task "$taskName" (type: JavaExec) {environment 'username',..}
task('Connect', type: JavaExec) {environment 'username', ..}
task('Insert', type: JavaExec) {environment 'username', ..}
```

Listing 7.1: Commit 560850*d* in *biginsight-examples* Project: An Example of **DRY** Refactoring Type in Gradle.

In this study, we address the knowledge gap on build refactoring types by conducting an empirical study on build refactorings in open-source projects. Our analysis includes building a taxonomy of build-related refactorings, an investigation into which TDs these refactorings address, and the creation of BuildRefMiner to automatically detect build refactorings in past commits. In this context, we address the following research questions:

RQ1: Which refactoring types are developers applying to build code? This RQ aims to build a taxonomy of build refactorings. We were able to develop a taxonomy of 24 build refactoring types classified into 6 main categories. 8 refactoring changes are

Build-specific such as Dependency Organization and Synchronizing Shared Build Properties refactorings.

RQ2: How are build refactorings linked to technical debt? This RQ aims to uncover which TDs can motivate developers to implement build refactoring operations. In total, we were able to extract 5 TDs linked to refactoring categories. For example, DRY addresses the TD Code Duplication & Redundancy.

RQ3: What is the effectiveness of our tool BuildRefMiner in identifying build refactorings? To provide guidance to future research, we developed BuildRefMiner, to automatically detects build refactorings in the commit history of a project. We were able to achieve an F-1 score of 0.76 in detecting build refactorings.

The main contributions of this work are:

1. The first dataset on refactorings in Build systems. The dataset can be further utilized in future research on developing tools and techniques for detecting and recommending refactoring opportunities.

2. The first quantitative and qualitative study on refactoring changes in multiple build systems: Maven, Ant, and Gradle. We propose a rich taxonomy of a total of 24 refactorings divided into 6 main build refactoring categories. Furthermore, we link 20 of these categories with TDs they address via 85 commit messages and 60 developer responses.

3. BuildRefMiner, which we develop for automatic build-refactoring identification, enabling researchers to detect refactoring patterns within build systems more efficiently.

7.1 Methodology

In this section, we describe our research methodology. Figure 7.1 provides an overview of the process, which is composed of three primary phases: Data Preparation, Quantitative Analysis, and Implementation of BuildMiner.

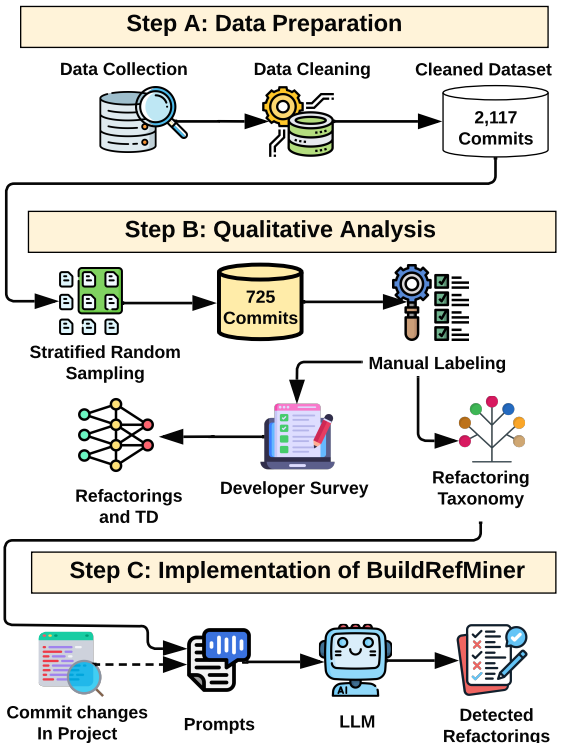

Figure 7.1: Approach Overview

7.1.1 Data Preparation

7.1.1.1 Data Collection In this step, our goal is to have a gold set of commit changes in which the developers explicitly reported the refactoring activity.

To achieve this, we set out to retrieve commits that involve refactorings and are associated with Gradle, Maven, and Ant. We utilized BigQuery [186], an extensive collection of open-source GitHub projects, via the SQL query presented in Listing 7.2. We

utilized a keyword-based mechanism to filter out all entries that do not contain the keyword 'refactor*' in the commit message. We use * to capture extensions like refactors, refactoring etc. The keyword-based approach has been widely used in previous studies related to identifying refactoring changes [187, 188, 189], as it allows us to prune the search space to only consider code changes whose documentation matches a specific intention. The choice of 'refactor*', was specifically made to reduce the occurrence of false positives [190, 188, 191]. Furthermore, we mention variations of file names corresponding to Maven, Ant, and Gradle in our query. Overall, these steps helps ensure that the extracted refactorings are more likely to be build-related. Through this initial filtering, we collected a total of 5k commits.

```
SELECT * FROM "bigquery-public-data.github_repos.commits"
WHERE (message LIKE '%refactor%') AND (message LIKE '%pom.xml%'
OR message LIKE '%build.xml%' OR message LIKE '%build.gradle%');
```

Listing 7.2: Extracting Refactoring-related Commits

7.1.1.2 Data Cleaning In this step, we omit commits pertaining to repositories derived from other repositories, such as forked repositories, to prevent any redundancies and resulting potential biases in our study. In addition, we exclude projects that no longer exist, as they may be de-listed from GitHub but still exist in BigQuery. This left a total of 2453 commits, 691 correspond to Gradle, 1450 correspond to Maven and 312 correspond to Ant. Given the substantial size of the data and the considerable effort and time required for its manual examination, we applied a stratified random selection process to select a diverse and representative sample of build-related commits, following the recommendations of Levin et al. [192]. The stratification criteria was the build systems to which the commits correspond. For each build system, we selected a sample with a confidence level of 95% and a Margin of Error of 5%, which was determined to 248 are for Gradle, 173 for Ant, and 304 for Maven,

for a total of 725 commits. These commits originated from 609 projects, which had an average 86K and median 9.7K lines of codes, an average 160 and median 1 forks, and finally an average 606 and median 5 stars.

These projects used a varied set of 30 programming languages. For example: 487 projects used Java, 51 used Kotlin, 19 used PHP, among other programming languages, as outlined in the appendix [193].

7.1.2 Qualitative Analysis methodology

7.1.2.1 Commit Analysis Methodology

Given that this study represents the initial investigation into the development of code-specific refactorings for build systems, it was imperative to conduct a manual examination of commits to identify the different refactorings as well as any relationship they may have with TD. We designed this process to minimize bias, by abiding by the recommendations outlined by Usman et al. concerning the construction of taxonomies [194].

First, during the planning phase, three co-authors agreed on the area of focus being build refactoring. The main aim as the identification of the different types of refactoring that developers use in build scripts and to ascertain whether these were utilized to address technical debt. Finally the classification framework as a tree. Although a build refactoring taxonomy had been proposed in the literature by Simpson et al. [61], it was not utilized in this process to avoid bias, as it was based on subjective opinions, lacked empirical validation, and was technology-specific to an old version of Ant.

Second, the identification and extraction phase was performed over two rounds. For the first round, two of the co-authors with refactoring and build scripts experience separately observed all the build commits in order to identify any refactoring changes and explicit mentions of TD. They utilized code modifications; code comments; commit title, message, comments; and build-systems-related online resources as information sources. The two

co-authors maintained a shared record of the names and high-level descriptions of the refactoring changes in order to avoid any redundant categorization. For the second round, the third co-author with previous build scripts experience was asked to rejoin this process. This was to minimize the bias, as this author did not participate in the categorization of refactoring changes in the previous round. These three co-authors then categorized all 725 build commits separately, extracting any refactoring operations and mentions of TD. These rounds were performed over one month to avoid labeling fatigue.

After the two labeling rounds concluded, Fleiss' Kappa was calculated at 0.76, signaling high agreement [195]. Then, three rounds of consensus meetings were carried out to resolve any disagreements.

Third, for the design and construction phase, a card-sorting process [196, 197] was performed by the three authors. This process grouped refactoring changes of similar characteristics into 24 categories. Subsequently, they refined this classification by determining 6 high-level main categories, for an easier generalizability and usability, giving it the format of a tree. Furthermore, we provide specific examples along with the definitions of the categories in order to avoid definition bias. This taxonomy is presented and detailed in subsection 7.2.1.

Finally, for the Validation phase, a fourth co-author who possesses extensive expertise in the field of build systems was able to verify the relevance of the different categories. Notably, the authors found that this process alone was insufficient to establish a link between build refactorings and technical debt. We address this in subsubsection 7.1.2.2.

7.1.2.2 Developer Opinion Collection Concerning the uncovering of links between the identified refactoring categories and TDs, a main challenge was the lack of documentation, along with ambiguous descriptions, which made it difficult to identify the prevalence of TD in the context of build refactoring. Hence, we opted for a conservative approach, where we

only considered TD as being addressed by the refactorings if it was explicitly mentioned in the commit message or code comments. However, this approach may have led to the underestimation of the actual number of instances of TD repayment via build refactoring. To compensate for this shortcoming, and in line with previous studies [198], we performed a cold-calling-based survey. We emailed and send direct-messages to the commit authors to enquire about the different refactoring instances and whether they were linked to a TD. This survey was open-ended, and only took the form of one question: The motivation behind the applied refactoring changes. This was done to minimize the burden on the developers. This survey was conducted over a period of one month.

In total, we identified 85 commit messages/descriptions that clearly describe the motivation behind applying the refactoring changes. Furthermore, we received survey responses from 60 developers, out of 250 originally contacted, achieving a 24% response rate, in line with the average response rate of 15-30%, for software engineering surveys[199, 200, 201]. The survey responses provided us with additional insights into the refactoring changes and their relationship with technical-debt. For example, commit [7] contains an *Extract Module* refactoring , which was accompanied with the ambiguous commit message '*build.gradle refactoring*'. The developer clarified that this refactoring was used to address the modularity, and provided us with the following explanation: *build.gradle was too long (500 lines) and divided the functions...for better modularity*—thus clarifying the link between the different refactorings in this commit and TD. The results of this process and this survey are detailed in subsection 7.2.2.

7.1.2.3 Implementation of BuildRefMiner As mentioned in subsection 7.1.2, the difficulty and time-consumption of manually identifying instances of build refactoring highlights the need for automated, build-specific refactoring discovery. Hence, we design BuildRefMiner, a tool that can automatically analyze commits that affect build files to extract any refactoring

Prompt

```
Task: Given the commit changes below that are applied on build system build scripts which
will be delimited with delimiter characters, identify any occurrences of the listed refactoring
types.
Provide the results strictly in JSON format with the following keys: RefactoringType and
Details.
If there are multiple refactorings, return them as a list of JSON objects, where each object
contains the following: - "RefactoringType": The type of refactoring detected. - "Details": A
Description with further information about the change.
Here are two examples of the format I expect: ....
If no refactorings are detected, return the message: "No refactorings found."
This is a list of 24 refactoring types in build files:
...
commit
```

Figure 7.2: BuildRefMiner Prompt

operations that may have occurred. Currently, it utilizes GPT-4o, but it can be configured to use other LLMs. It utilizes LLMs due to their adaptability and proclivity with source-code analysis [202, 203, 204, 205]. We utilized Prompt-Engineering practices [206, 207] while building this tool, specifically Zero-shot and One-Shot prompting.

The prompt, a snippet of which is shown in Figure 7.2, is composed of a system instruction, a list of the names and definitions of the build refactorings we discovered, inserted at the location of the text highlighted in green. As part of the One-Shot variant of the prompt, a code snippet example demonstrating each refactoring type is given along with each definition. Then, we introduce the new commit changes for analysis, under the commit variable highlighted in yellow. We developed three distinct one-shot learning prompts, each tailored with examples specific to the build systems Gradle, Maven, and Ant for each refactoring type, where the build system and delimiter variables are changed depending on the build system being analyzed.

We evaluated the performance of both prompting approaches, and we detail the results of this evaluation in subsection 7.2.3. While it is possible to use static analysis techniques to implement this tool, the reliance on LLMs has allowed us to quickly utilize,

evaluate, and improve BuildRefMiner. Furthermore, this implementation is more flexible and easier to extend for other build systems, unlike a static-analysis tool that would need to be customized to support every specific build tool. We provide BuildRefMiner as an accompaniment to our taxonomy and as a proof of concept concerning the relevance and importance of build refactorings and to facilitate future work. The implementation of BuildRefMiner with the usage of static analysis and its comparison with the LLM implementation can be the subject of interesting future work.

7.2 Study Results

7.2.1 RQ1:Which refactoring types are developers applying to build code?

To uncover and categorize the Build refactoring types that are present within build scripts, we followed the methodology discussed in subsection 7.1.2 and carried out a manual analysis of 725 commits. We discovered that 32% of them were false positives, as they did not contain any discernible refactorings. The primary cause of the inclusion of these false positives was the use which sometimes involved the addition or modification of existing functionality [208, 209]. This left a total of 403 true-positive Build-refactoring-related commits.

Figure 7.3 represents the taxonomy that we have generated based on a parent-child hierarchy. These refactoring types have been organized into 6 main categories for ease of classification and analysis. The tuple under each refactoring type indicates the number of refactoring changes we discovered from Gradle, Ant, and Maven respectively. Each percentage value represents the proportion of refactorings within a specific category out of the total of identified refactorings of the same main category. The classification into 6 main categories was based on the scope of their impact on the build code. These ranged from broad, project-level impacts ('Code Clean Up') to more specific levels, including Module ('Module Hierarchy Organization'), Method/Task ('Subroutine Organization'),

Dependencies ('Dependency Organization'), Shared Properties ('Synchronizing Shared Build Properties'), and Local Variables ('Variables Organization'). The different subcategories offer greater specificity as they represent the different kinds of refactoring identified. In total, we were able to develop a taxonomy of 24 build refactoring types. In the rest of this section, we discuss each of the Build-related Refactoring categories as well as their sub-categories by giving detailed definitions. In addition, we provide code examples to mitigate the potential bias inherent in definitions. It is notable that these definitions may be extended for application in other code artifacts.

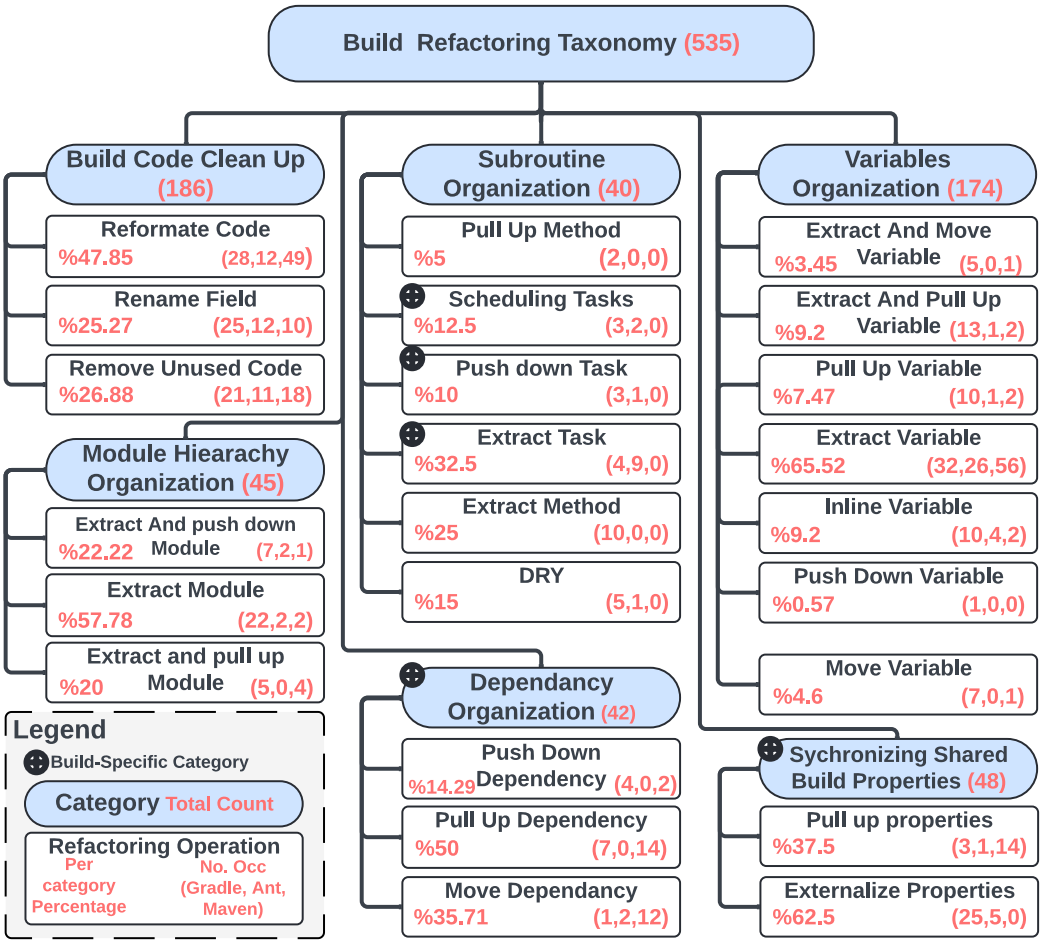

Figure 7.3: Build-related refactorings taxonomy

7.2.1.1 **Build Code Clean Up** This particular category encompasses alterations that serve to improve the cleanliness, readability, and comprehensibility of the build script. The detected refactoring operations represented the most prevalent refactorings for the three build systems, accounting for 34.77% of all refactorings. Three subcategories were observed:

1.1 Reformat Code (RC) This refactoring makes up 47.8% (89/186) of the Build Code Clean Up category. It involves aligning code with specific style conventions, adjusting indentation, reorganizing code blocks, and ensuring consistent coding style for clarity. Listing 3 shows an example in a Gradle script, switching to the plugins DSL for readability and modern best practices.

```
plugins {
    id 'jacoco'}
    id 'groovy'}
apply plugin: 'jacoco'
apply plugin: 'groovy'
```

Listing 7.3: Commit *f*833*d*19 in *android-gradle-jenkins-plugin* Project: An example of Reformat Code Refactoring Type in Gradle.

1.2 Remove Unused Code (RUC) This removes redundant or unused build code, accounting for 26.88% (50/186) of the Build Code Clean Up category.

1.3 Rename Field (RF): This refactoring, 25.27% (47/186) of the total, involves renaming build components (e.g., files, tasks, methods) to improve code clarity.

7.2.1.2 **Module Hierarchy Organization** This particular category pertains to the process of dividing the build script into separate modules (build files), with each build file being assigned a specific role or functionality. These modules are then arranged hierarchically, taking into consideration their dependencies and relationships. The refactoring operations

that were identified collectively accounted for 8.41% of the total refactorings. The study identified three subcategories:

2.1 Extract Module (EM) represents 57.78% (26/45) of the refactorings in this category, which involves extracting and relocating functionality or responsibilities into a new build file within the same hierarchy. Notably, this refactoring type is more common in Gradle (22 occurrences) than in Maven and Ant (2 occurrences each), making it one of the most prevalent refactorings found in Gradle. Commit [210] illustrates an example of Extract Module. In this commit, developers extracted a segment of the parent 'build.gradle' script detailing multiple functionalities and relocated it to a new build file within the same hierarchy, 'publish.gradle'. The initial build file utilizes apply from 'publish.gradle' to invoke the extracted responsibilities.

2.2 Extract And Pull Up Module (EPUM) represents 20% (9/45) of the refactorings, it entails moving functionality or responsibilities from a more specialized module or class (child) to a newly created more generalized or parent module.
Commit [211] illustrates an example of Extract and Pull Up Module in Maven. This change extracts a segment of a build code that delineates Maven properties from "lib/pom.xml and relocate them to a superior newly created Maven file "pom.xml".

2.3 Extract And Push Down Module (EPDM) represents 22.22% (10/45) of the refactorings, it shifts functionality or responsibilities from a general or parent module to a more specific one newly created (child).

7.2.1.3 **Subroutine Organization** This category covers operations modifying subroutines, specifically tasks or methods, primarily in Gradle and, to a lesser extent, Ant. It is noteable that Maven lacks comparable concepts. In Gradle, tasks represent actions, and methods configure them. Ant has predefined XML tasks but no methods. Therefore, refactoring related to methods applies solely to Gradle, while task-related refactoring includes both Ant

and Gradle but excludes Maven. This category accounts for 7.48% of all identified refactorings. In total, we discovered two refactorings that focus on methods, and four refactoring types that are focused on tasks.

3.1 Extract Method (EM) This type represents 25% (10/40) of the Subroutine Organization category. It entails the identification of a cohesive code fragment and its subsequent relocation into a newly created method or function. Subsequently, the original code is substituted with an invocation of the recently generated method. Commit [212] provides an example of the Extract Method refactoring applied to a Gradle script. In this instance, the methods *createChecksum()* and *createExecutable()* were extracted from the *makeExecutables* task and invoked within the same task (Lines 34-41), improving code extensibility and reusability.

3.2 Pull Up Method (PUM) accounts for 5% (2/40) of the total refactorings in this category. It involves moving an existing method to the relevant Parent build file. An example of this refactoring is illustrated in commit [213] where *uploadArchives* method has been pulled up from a subfile *concourse-cli/build.gradle* to the parent file *build.gradle*.

3.3 Extract Task (ET) accounts for 32.5% (13/40) of the total refactorings in this category. It involves identifying a cohesive code fragment, relocating it into a new task, and replacing the original code with a call to the newly created task. Listing 7.4 is an example of Extract Task refactoring in Gradle. It defines a new task, resolveDependencies, which is set as a prerequisite for the publish task. Commit [214] serves as an illustration of the Extract Task refactoring type within the Ant build system. In this instance, eleven defined tasks (such as *start*, *start-secure*, and *start-batch*) exhibited identical behaviors, leading to code duplication. To eliminate this redundancy, a new task, *exec-scipio-jar*, was introduced (line 1033) to consolidate and encapsulate the shared behavior of these tasks.

```
1 publish.dependsOn {
2 task resolveDependencies{doFirst {
3    project.publishing {....}}}
4 publish.dependsOn(resolveDependencies)
```

Listing 7.4: Commit 4*f*03800 in *droidmate* Project: An example of Extract Task Refactoring Type in Gradle.

3.4 Push Down Task (PDT) represents 10% (4/40) of the refactorings. It entails moving a task defined in the root project's build file to the build file of one or more sub-projects.

3.5 Scheduling Tasks (ST): This refactoring type represents 12.5% (5/40) of the total refactorings in this category. It ensures a specific order of execution of tasks, or the execution multiple tasks simultaneously. In Ant, sorting tasks can be done using *depends* attribute that specifies the order of execution by indicating which tasks need to be completed before the current task begins. Listing 7.5 shows an example of a Scheduling Tasks refactoring in Ant. The changes were in place to ensure that *init* task was executed only after the *clean* task (Line 2).

```
1 <target name="init"
2 <target name="init" depends="clean">
3    <mkdir dir="${bin}"/>.....
4 </target>
```

Listing 7.5: Commit 717417*f* in *aDTN-platform* Project: An example of Scheduling Tasks Refactoring Type in Ant.

3.6 Don't Repeat Yourself (DRY):

This refactoring type is related to the process of consolidating the common behavior of multiple tasks into a single location method to void redundancy. Listing 7.1 shows an example of the *DRY* refactoring type in Gradle. Initially, tasks such as Connect and Insert were explicitly configured, each possessing its own set of parameters such as environment

variables (Line 3-4). In the revised iteration, the identical configurations are dynamically applied to tasks by iterating through a list that includes the task names Connect and Insert (Line 1). As a consequence, the script becomes more succinct by eliminating repetitive configurations and replacing them with a generalized loop.

7.2.1.4 **Dependency Organization** This particular category is a build-specific category, concerned with the reorganization of dependencies between build files, accounting for 7.85% of the overall count of identified refactoring changes. It plays a crucial role in determining the order and manner in which build dependencies are built and integrated. The study conducted identified three distinct types:

4.1 Move Dependency (MD) It refers to the transfer of dependency between two build files within the same hierarchy. It accounts for 35.71% (15/42) of the overall count of identified refactoring changes in this category, with 12 occurrences for Maven, compared to 1 and 2 occurrences each for Gradle and Ant respectively.

4.2 Pull Up Dependency (PUD) It involves moving a dependency from a sub-build file to its parent Build file; It accounts for 50% (21/42) of the overall count of identified refactoring changes in this category. Notably, this refactoring is more frequent in Maven, with 14 occurrences, compared to 7 and 0 occurrences each for Gradle and Ant respectively. Commit [215] exemplifies this type of refactoring by consolidating various dependencies, including the Spring Boot Gradle plugin and the Kotlin Gradle plugin, that were previously declared across multiple sub-build files. These dependencies were eliminated from the individual sub-build files and migrated to the parent build file *build.gradle*, specifically within lines 1-38, to enhance maintainability and reduce redundancy.

4.3 Push Down Dependency (PDD) It accounts for 14.29% (6/42) of the overall count of identified refactoring changes in this category. It entails moving a dependency from a parent file to a child or specific build file.

7.2.1.5 **Synchronizing Shared Build Properties** This category is another discovered Build-specific refactoring category and represents 8.97% of the overall count of the identified refactoring types. It refers to the process of ensuring that multiple parts or modules of a build system use a consistent set of properties. These properties can be configurations, versions, paths, or any piece of data that affects how the build process operates. When projects grow in complexity and include multiple components or modules, it's essential to maintain a single source of truth for shared properties to avoid inconsistencies. Two distinct types were observed during the study:

5.1 Externalize Properties (EP): Accounts 37.5% (18/48). This type of refactoring focuses on the centralized and harmonized handling of build configurations by extracting environment-specific configurations, credentials, and settings from the build script and application code. This is typically achieved by utilizing external properties or separate configuration files, such as *.properties* files. This type of refactoring is particularly common in Gradle, with 25 instances, compared to 5 and 0 instances for Ant and Maven, respectively. Listing 7.6 shows an example of Externalize Properties. The initial implementation involved hardcoding the version value directly into the script. The revised modifications improve version management efficiency by introducing an external *build.properties* file (Line 2) that encapsulates essential configurations. This approach decouples version management from the build script, allowing users to update configurations easily by modifying the properties file rather than directly altering the script itself. Decoupling the components of the build system not only improves the ease of reading, but also enhances the ability to maintain it. This allows developers to make changes to configurations without needing to delve into the fundamental build logic.

```
version = "1.7.10-0.1"
ext.configFile = file "build.properties"
configFile.withReader {def prop = new Properties()
  project.ext.config = new ConfigSlurper().parse prop}
version = config.version
```

Listing 7.6: Commit 611*c*4*b*3 in *YetAnotherBackupMod* Project: An Example of Externalize Properties Refactoring in Gradle.

5.2 Pull Up Properties (PUP) Accounts 62.5% (30/48): pertains to the process of consolidating frequently used configurations and properties within a build file into a centralized location or method within the root build file. This refactoring type is most common in Gradle Maven with 25 occurrences compared to 3 and 1 occurrences for Gradle and Ant respectively. A specimen of this refactoring type is illustrated in Commit [216]. The changes involve migrating configuration elements from a child Maven build file *core/pom.xml* (Line 24-29) to the root *pom.xml* (Line 56-60). By centralizing the source control management metadata in the root file, the project benefits from improved coherence and ease of updates, as any changes to these properties need to be made in only one location.

7.2.1.6 **Variables Organization** This category constituted the second largest refactoring category after the Code Clean Up Category, accounting for 32.52% of the total count of identified refactoring types. It is primarily associated with the organization of variables within a single build file or across the hierarchy of build files. All the identified subcategories of variable refactoring are more frequent in Gradle compared to Ant and Maven. They are as follows: **Extract Variable(EV)** accounting for (65.52%), **Inline Variable (IV)** (9.2%), **Move Variable (MV)** (4.6%), **Pull Up Variable (PUV)** (7.47%), **Push Down Variable (PDV)** (0.57%), **Extract And Move Variable (EMV)** (3.45%), and **Extract And Pull Up Variable (EPUV)** (9.20%).

Out of these refactoring categories, we would like to highlight those that are specifically related to Build systems (Build-specific) that do not have equivalents in other artifacts (e.g., source code). Among the 24 refactorings, we identified 8 as Build-specific types. These include the Tasks-related refactorings, under the *Subroutine Organization* main category (Scheduling Tasks, Push Down Task, and Extract Task), and all the refactoring belonging to the *Dependency Organization* and *Synchronizing Shared Build Properties* main categories. We observed that developers have to specifically adapt to the context of Build files when applying these refactoring changes. This distinction arises from the nature of their primary concerns, which revolve around the foundational elements of build systems: tasks, dependencies, and properties.

Comparing our Taxonomy with Simpson et al. [61], the categories we defined are much more technology agnostic, in comparison to some of their technology-specific categories, such as *Move element to Antlib* and *Replace Exec with Apply*. Furthermore, Simpsons' taxonomy does not include some categories we discovered, such as DRY and Code Cleanup. While some categories are indeed similar between the two taxonomies, such as Simpson's Introduce Property File and our Externalize Properties Categories, our taxonomy is organized into empirically validated 6 Main categories that are dependent on scope, and we use naming conventions that are more inline with Refactoring in other domains, which makes our taxonomy easier to understand and utilize.

Finding 1

We created a build refactoring taxonomy composed of 24 refactorings types, that fall under 6 main refactoring categories. 8 of the 24 refactorings are specific to build systems.

7.2.2 RQ2: How are build refactorings linked to technical debt?

Our analysis of 85 commit messages and 60 developer responses we that 20 out of the 24 identified refactoring types are associated with technical debt (TD) reduction, accounting for 94.01% of the total refactoring instances. However, five refactoring types—Move Dependency, Extract and Move Variable, Move Variable, Push Down Task, and Push Down Variable—were not classified as related to TD repayment. The rationale behind the application of these five refactoring types remains unclear, suggesting the need for further investigation into their role in TD repayment.

7.2.2.1 Technical Debt Categories We extracted 5 technical debt categories related to *Extensibility & Maintainability* , *Modularity*, *Clarity & Readability*, *Code Duplication & Redundancy*, and *Security*. We organized the different refactoring types based on their rationale, e.g., technical debts they address as shown in Table 7.1. The percentage associated with each technical debt represents the cumulative number of refactorings undertaken to address that particular TD out of the total number of refactorings identified.

Clarity & Readability is a TD that receives considerable attention during the process of refactoring build files by 39.63% of the total refactorings. It was linked with the following refactoring types: *Inline Variable*, *Rename Field*, *Reformat Code* and *Remove Unused Code*. These refactorings were used when code is not clear, and hard to read requiring huge efforts to understand it. For example, Rename field was used to avoid using irrelevant names for build artifacts and to ensure consistency in naming conventions, like the example commit message *"In order for this to become more comprehensive ... the package name needed to be changed"* from the commit [217].

Reformat Code and Inline Variable were used to ensure better readability and understandability as shown in the developer response in Figure 7.4 to commit [5].

Developer Response

```
The reason was rather to unify code stale and improve readability.
```

Figure 7.4: Developer Response on Commit [5]

Extensibility & Maintainability was among the most addressed TDs in build files, and was mainly associated with: *Extract Variable*, *Extract And Move Variable*, *Extract Method*, *Extract Task*, and *Move Task*, repaid by 30.66% of the total refactorings.

As software grows and evolves, adding new components or improving existing ones often requires modifying or duplicating elements in the build system, which can lead to challenges in both extensibility and maintainability. Refactoring techniques like Extract Variable, Extract Task, and Extract Method aim to increase the system's abstraction levels and reusability, making future modifications easier and reducing the risk of widespread bugs. This is illustrated by a example commit message *"Task 'retroweaver' moved to the prepare target so it can be reused in several places"* for the commit [218].

Also, externalizing properties ensures that changes can be made with minimal disruption, enhancing both the flexibility to adapt the system and the ease with which future changes and maintainability efforts as in the example of a developer response shown in Figure 7.5

Developer Response

```
Moving properties on its own file gives us the ability to force a dependency(s) version to be
consistent ...
```

Figure 7.5: Developer Response of Commit [6]

Modularity This technical debt is addressed by 14.95% of the total refactorings, such as Extract Module, Extract and pull up Module, Extract and Push Down Module, Push Down Task, Pull Up Method, Pull Up Variable, Extract and Pull Up variable and Push Down Dependency. These refactoring techniques break down complex build files into smaller, modular units, ensuring that each component is situated at the most appropriate level of the build system hierarchy, making management easier, clearer, and reducing errors during updates. This is illustrated by commit messages [7]: *"refactor build.xml for centralized usage for all plugins."*, and developers answers such as the ones shown in Figure 7.6 and Figure 7.7. Extract Module, Extract and pull up Module, Extract and Push Down Module, Push Down Task, Pull Up Method, Pull Up Variable, Extract and Pull Up variable and Push Down Dependency. These refactoring techniques break down complex build files into smaller, modular units, ensuring that each component is situated at the most appropriate level of the build system hierarchy, promoting simplicity, clarity, making it easier to manage, and less complex, and reducing the likelihood of errors when components are changed. This is illustrated by developers answers shown in Figure 7.6 and Figure 7.7, and the commit message *"refactor build.xml for centralized usage for all plugins."* for commit [7]

Developer Response

```
...  I have decided that build.gradle was too long (500 lines) and divided the functions in it
according to responsibilities, for better modularity ...
```

Figure 7.6: Developer Response for Commit [7]

Developer Response

```
Split dependencies in optional modules (at this time gumetree was growing to much and we started
to make it more decoupled)
```

Figure 7.7: Developer Response for Commit [8]

Code Duplication Duplicated code and redundant elements make a system more difficult to maintain, prone to errors, and less efficient. This is addressed by 4.20% of the total refactorings Refactoring techniques such as Pull Up Properties, Pull Up Method, Pull Up Dependency, Extract and Pull Up Variable, and Pull Up Variable focusing on consolidating shared code fragments across various classes into higher-level modules. By pulling up shared elements, these techniques eliminate duplication, making the code more manageable, and reducing the likelihood of inconsistencies. This is illustrated by a commit message *"Reduced duplicate POM XML content"* for commit [219], where the refactorings Pull Up Properties and Pull Up Dependency where applied. Also, an example of a developer response for commit [6] shown in Figure 7.8, where Pull Up Variable and Pull Up Dependency were applied.

Developer Response

```
This was purely to dedup code, so gradle changes could be made in a single place, instead of
multiple files, for each sub project.
```

Figure 7.8: Developer Response of Commit [6]

Also, implementing the DRY principle is key in reducing code redundancy, and improving clarity and maintainability while reducing the risk of future bugs. An example of

DRY change in commit [220] where the developer states in the commit message *"Refactor of productFlavors So now we aren't duplicating a bunch of buildConfigField values"*.

Security: This technical debt was mainly addressed by *Externalize Properties* refactoring (1.12%) . Build scripts may expose sensitive information or have vulnerabilities that could be exploited. Refactoring techniques such as "Externalize Properties" help address these security risks by reducing potential breaches through improved access control and securing configurations, safeguarding the system from attacks. The goal of this refactoring was also described in the commit message *"Fixes security consideration issues"* for the commit [221], where sensitive build properties were externalized to a separate file that is then added to .gitignore.

Technical Debt	**Percentage**	**Refactorings**
Clarity & Readability	39.63%	RF, IV, RUC, RC
Extensibility & Maintainability	30.66%	EV, EP, EM, EV
Modularity	14.95%	EM, EPUM, PUV, EPDM, PDD
Code Duplication	14.20%	PUP, PUM, PUD, EPUV, PUV, DRY
Security	1.12%	EP

Table 7.1: build refactoringTechnical Debt Categories

Finding 2

Among the 24 refactoring types identified, 20 are linked to reducing technical debt. We identify which of these refactoring types addresses which of the five specific types of technical debt.

7.2.3 RQ3: What is the effectiveness of BuildRefMiner in identifying build refactorings?

After the manual labeling of refactoring commits undertaken in subsection 7.1.2, we created a gold set of labeled commits and corresponding refactorings within them. We use this labeled set to evaluate the performance of BuildRefMiner. We calculate the precision (PR), recall (RE), and F1-score (F1) of BuildRefMiner across the different refactoring types. The results of this evaluation are shown in Table 7.2. It's important to note that the refactoring code examples used within the One Shot variant of the prompt were removed from this evaluation set when evaluating the One-Shot variant. This was done to avoid information leakage and ensure the accuracy of our evaluation.

Using Zero-Shot prompting, BuildRefMiner yielded mixed results across the 24 refactoring types, achieving an overall precision of 0.67, recall of 0.72, and F1-score of 0.66. Notably, two refactorings such as Extract and Move Variable and Push Down Variable achieved an F1-score of 0.95 and 1 respectively, indicating a strong alignment of BuildRefMiner with the empirical definitions in this category. In four refactoring types—Extract and Pull Up Module, Extract Module, Push Down Dependency, and Move Dependency—BuildRefMiner maintained a high degree of performance in its classification, with an F1-score from 0.7 to 0.84. However, a notable deviation was observed for certain types, including Push Down Module, Extract Method, Move Variable, and Pull Up Properties, which presented lower F-1 scores, between 0.25 and 0.57, suggesting that the Zero-Shot model struggles with nuances in detecting these refactorings without prior context.

A significant improvement in BuildRefMiner performance across several refactoring types can be seen when using One Shot Prompting. The overall precision, recall, and F1-score go up to 0.79, 0.78, and 0.75 respectively. Going over the performance of BuildRefMiner over the different refactoring types, BuildRefMiner demonstrated enhanced

performance in detecting 20 out of the 24 refactorings overall. A notable example is the dramatic increase in F1-score from 0.25 to a 1 for the Extract and Push Down Module refactoring. This substantial improvement highlights the effectiveness of providing contextual examples, enabling the model to better interpret and accurately detect code changes without generating false positives or missing instances. Additionally, in two other refactoring categories—Extract Module and Extract and Move Variable—BuildRefMiner maintained high precision (1 and 0.96) alongside near-perfect recall, achieving 0.95 in both cases.

However, despite the tool achieving significant recall improvements for Rename Field, Remove Unused Code, Scheduling Tasks, Reformat Code, and Move Variable, ranging from 0.81 to 1, the precision for these categories was noticeably lower, falling between 0.55 and 0.72. This indicates that, although the tool successfully identified every true instance of these refactorings, it also misclassified some unrelated changes as refactorings, resulting in moderate precision. Thorough manual validation, it became apparent that some of these refactorings were often intertwined with other refactoring actions, which contributed to the false positives observed. BuildRefMiner tended to identify them as distinct refactorings even when they were embedded within broader refactoring activities, thus inflating the number of false positives. For example, Move Variable was frequently accompanied by actual refactorings such as Move Method, Move Task, or Extract Module. Remove Unused Code frequently overlaps with refactorings that involve moving code to another class or module (e.g Extract Module, Move Method etc..).

To demonstrate the usefulness of BuildRefMiner on projects in the wild, we share the result of running it on the gradle-modifying commit [9] from `Google/Nomulus`. This particular commit involved multiple modifications across different build files at various levels of the project hierarchy, making it hard to parse manually. BuildRefMiner categorized

Refactoring Types	Zero-Shot			One-Shot		
	PR	**RE**	**F1**	**PR**	**RE**	**F1**
Rename Field	0.54	0.71	0.61	0.72	0.82	0.76
Remove Unused Code	0.53	0.77	0.63	0.56	0.81	0.66
Scheduling Tasks	0.48	0.84	0.60	0.55	1.00	0.70
Reformat Code	0.53	0.80	0.64	0.55	0.83	0.66
Extract and Pull Up Module	1.00	0.75	0.81	1.00	0.78	0.88
Extract and Push Down Module	0.17	0.50	0.25	1.00	1.00	1.00
Extract Module	0.79	0.95	0.84	0.96	0.95	0.93
Extract Method	0.33	0.57	0.42	0.75	0.6	0.55
Pull Up Method	1.00	0.50	0.67	1.00	1.00	1.00
Extract Task	0.50	0.63	0.55	0.58	0.63	0.60
Push Down Task	0.75	0.67	0.70	0.75	0.67	0.7
DRY	0.75	0.60	0.67	0.88	0.80	0.84
Push Down Dependency	0.84	0.84	0.84	0.84	0.84	0.84
Pull Up Dependency	0.82	0.53	0.64	0.95	0.53	0.67
Move Dependency	0.78	0.81	0.70	0.78	0.81	0.70
Pull Up Properties	0.52	0.63	0.57	0.65	0.67	0.66
Externalize Properties	0.73	0.57	0.64	0.81	0.65	0.72
Extract Variable	0.74	0.74	0.74	0.77	0.78	0.77
Inline Variable	0.60	0.55	0.51	0.83	0.58	0.61
Move Variable	0.50	0.50	0.50	0.67	1.00	0.80
Pull Up Variable	0.78	0.47	0.58	0.62	0.53	0.57
Push Down Variable	1.00	1.00	1.00	1.00	1.00	1.00
Extract And Move Variable	1.00	0.90	0.95	1.00	0.95	0.96
Extract And Pull Up Variable	0.50	0.62	0.49	0.83	0.68	0.68
All Refs.	**0.67**	**0.72**	**0.66**	**0.79**	**0.78**	**0.76**

Table 7.2: BuildRefMiner Performance across the refactoring types with Zero-Shot and One-Shot Prompting.

the changes as shown in Figure 7.9, of which we manually verified the veracity, thus giving a glimpse into the usefulness and time-efficiency of our tool for future research.

- Extract Module
  Details: Code changes in 'gradle/build.gradle' have been extracted to 'appengine_war.gradle', simplifying the main build file and centralizing common configurations.
- Extract and Pull Up Module
  Details: In 'gradle/buildSrc/build.gradle', configurations such as buildscript dependencies, plugins, and checkstyle configurations have been extracted and referenced using 'apply from: '../java_common.gradle''.

Figure 7.9: BuildRefMiner output on Commit [9]

Finding 3

We tested two BuildRefMiner variants, finding that the One-Shot prompt variant achieved Precision, Recall, and F-1 scores of 0.79, 0.78, and 0.76, respectively, outperforming the Zero-Shot variant, which scored 0.67, 0.72, and 0.66.

7.3 Threats To Validity

Internal Validity: These threats may stem from errors in the categorization of refactoring changes in build code, which could introduce bias due to limited supporting documentation. To reduce this risk, a panel of three experts independently evaluated selected commits, where they documented refactorings and justifications with caution, only when highly confident. Results showed strong agreement across identification and classification.

External Validity: Random sampling may have excluded some large commits, potentially missing certain refactorings. However, we applied a stratification strategy to obtain a 95%-confidence representative sample, and the commits selected span a large variety of systems, giving credence to our results.

7.4 Study Implications

For Researchers: We help familiarize researchers with build refactoring, by providing them with an empirically-grounded and definition-supported Taxonomy composed of 24 Build refactorings, and we highlight the 5 TDs some of these refactorings address. Furthermore, we believe that the 2117 Build Refactoring commits we collected, BuildRefMiner which we developed, and the findings we reach can provide a groundwork to help guide future research into the field of Build refactoring.

For Practitioners: Due to the lack of empirically-grounded existing taxonomies on Build refactoring, practitioners are unaware or misinformed about this practice. We believe the taxonomy we provided, along with the descriptions of the different TDs addressed by the refactorings within this taxonomy, can help guide practitioners in performing Build refactoring procedures.

7.5 Conclusion

In this study, we performed an empirical analysis on 725 commits that were related to build code refactoring from 609 open-source projects. Through manual analysis, we classified the 24 build-related refactorings we discovered into 6 main categories, and we distinguish 8 of which as build-specific. We also linked some of them to 5 TD categories with the help of developer feedback. Finally, we developed BuildRefMiner to help guide future research into this field. To the best of our knowledge, this is the first empirical study of refactorings and technical debt in Build Systems.

CHAPTER EIGHT

CONCLUSION AND FUTURE WORK

The features and improvements that were delivered in this dissertation and the results that were achieved are summarized in this chapter. In addition, we suggest possible enhancements to the proposed contributions and discuss them.

8.1 Conclusion

This thesis explores various facets of software optimization to address the growing complexity, performance degradation, and maintenance challenges that modern software systems face. Our research contributions are multi-dimensional and address both the theoretical and practical aspects of software engineering, spanning from core code refactoring to advanced container management in diverse environments.

In **Chapter I, II and III** we define the research context and challenges, the contributions of this dissertation, required background, and state-of-the-art and related works to our work.

In chapters IV, and V, we have developed a multi-objective optimization strategy for managing software containers, ensuring efficient resource allocation and scalability in cloud and edge environments. Our approach leverages many-objective search techniques to dynamically balance workloads, particularly focusing on the efficient deployment in both traditional cloud infrastructures and resource-constrained environments such as smart vehicles.

In Chapter V, we have identified the gap between industry practices and existing refactoring recommendation tools. By conducting a comprehensive study of refactoring commits across multiple open-source projects, we have proposed context-aware refactoring recommendations that significantly enhance the relevance and effectiveness of these tools.

This contributes to bridging the gap between theoretical research and practical application, ultimately improving code maintainability and developer productivity.

Chapter VI extends beyond traditional refactoring by addressing the optimization of the entire software development lifecycle, including continuous integration and deployment (CI/CD) processes. we have conducted an empirical study on the refactoring of software build systems, an often-overlooked area in software maintenance. Our findings highlight the importance of maintaining build scripts to mitigate technical debt, and we have proposed the development of automated refactoring detection tools using large language models to further streamline this process.

In summary, our contributions provide a comprehensive framework for software optimization that addresses the multifaceted challenges faced by modern software systems. By integrating advanced optimization techniques, context-aware recommendations, and empirical validations, our work, which is published in different top journals and conferences as detailed in section 1.5, lays the groundwork for future research and practical implementations that will enhance the overall efficiency and performance of software systems.

8.2 Future Work

Building on the substantial contributions presented in this dissertation, several avenues for future research and development emerge, each with the potential to further advance the field of software quality optimization.

Future research will focus on the development of dynamic and context-aware refactoring tools. These tools will incorporate real-time insights to enable automated, flexible decision-making, ensuring they adapt to various development contexts and align closely with developers' needs. By addressing limitations in current tools, this work aims to

bridge the gap between theoretical recommendations and practical applications in refactoring.

Another avenue for exploration is the application of resource-aware strategies to new domains. Building on the many-objective optimization techniques designed for containers, this approach can be extended to areas such as high-performance computing or IoT ecosystems, where efficient resource allocation is critical. These efforts will enhance the versatility and impact of the proposed methodologies.

Enhanced analysis of build systems will also be a priority. Expanding the capabilities of BuildRefMiner to handle more complex scenarios and integrating advanced AI techniques will further improve its performance. Additionally, we will also establish quality metrics for build systems code and create a recommendation tool leveraging LLMs and machine learning to provide personalized refactoring recommendations.

The study of refactoring in emerging technologies, particularly in machine learning (ML) and cyber-physical systems (CPS), represents another critical focus. Investigating the unique challenges of these domains, such as technical debt and performance optimization, will extend the applicability of refactoring principles to cutting-edge areas of software engineering.

Beyond technical advancements, spreading refactoring literacy among developers and students is vital to fostering sustainable software engineering practices. As a first step, we developed a gamified learning platform that introduces refactoring concepts through engaging, interactive scenarios. This tool leverages game mechanics to help learners identify, understand, and apply common refactoring patterns in a fun, low-stress environment. Early results have shown promise in increasing both motivation and retention among participants.

We envision extending this gamified platform to cover refactoring concepts in build systems, a domain often overlooked in educational tools. Future versions will incorporate more refactoring types, hands-on coding challenges, real-world examples, and interactive

code simulations. The ultimate goal is to create a comprehensive, modular educational toolkit that demystifies refactoring and equips both novice and experienced developers with the skills necessary to improve software quality across the full lifecycle—from source code to build automation.

Through these continued efforts, we aim to not only push the boundaries of software optimization research but also cultivate a broader culture of code quality awareness in academia and industry alike.

Bibliography

[1] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs.," *Communications of the ACM*, vol. 36, no. 11, pp. 81–95, 1993.

[2] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering*, vol. 1, pp. 1–22, 1995.

[3] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Management science*, vol. 44, no. 4, pp. 433–450, 1998.

[4] "Additional material." `https://sites.google.com/view/containerscheduling/`. Accessed on 03/17/2022.

[5] "mockito version bump · koral–/android-gradle-jenkins-plugin@f833d19."

[6] "Gradle refactor issue49 (51) · airbnb/epoxy@49cd795."

[7] "build.gradle refactoring · ati-ozgur/kdd99reproducibleexperiments@fd166ab."

[8] "Refactored and upgraded dependencies · caiusb/gumtree@bf85ae7."

[9] "Refactor gradle project setup · google/nomulus."

[10] R. Salameen, H. Takruri-Rizk, and G. Cooper, "A preliminary study of software maintenance in e-commerce companies in jordan," *Journal of Multidisciplinary Engineering Science and Technology*, vol. 2, no. 9, pp. 2374–2381, 2015.

[11] G. Bhatia, A. Choudhary, and V. Gupta, "The road to docker: A survey.," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 8, 2017.

[12] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[13] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[14] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, 2012.

[15] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th international conference on software engineering*, pp. 488–498, 2016.

[16] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 21–30, IEEE, 2016.

[17] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*, pp. 404–428, Springer, 2006.

[18] B. Varanasi, *Introducing Maven: A Build Tool for Today's Java Developers*. Apress, 2019.

[19] B. Matzke, *Ant: The Java Build Tool in Practice*. Charles River Media, Inc., 2003.

[20] A. L. Davis and A. L. Davis, "Gradle," *Learning Groovy 3: Java-Based Dynamic Scripting*, pp. 105–114, 2019.

[21] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of java build systems," *Empirical Software Engineering*, vol. 17, pp. 578–608, 2012.

[22] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 41–50, IEEE, 2014.

[23] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 724–734, 2014.

[24] G. Kumfert and T. Epperly, "Software in the doe: The hidden overhead of"the build"," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2002.

[25] "what is docker swarm ?," 2021.

[26] "what is mesos by apache?," 2021.

[27] "what is kubernetes ?," 2021.

[28] M. Fowler, *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 ed., 1999.

[29] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–4, IEEE, 2016.

[30] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[31] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[32] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of JDeodorant: Lessons learned from the hunt for smells," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pp. 4–14, IEEE Computer Society, 2018.

[33] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 241–250, IEEE, 2014.

[34] C. Macho, S. McIntosh, and M. Pinzger, "Extracting build changes with builddiff," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 368–378, IEEE, 2017.

[35] Portworx, "2017 annual container adoption survey: Huge growth in containers," 2020. https://portworx.com/2017-container-adoption-survey/.

[36] R. Senington, B. Pataki, and X. V. Wang, "Using docker for factory system software management: Experience report," *procedia CIRp*, vol. 72, pp. 659–664, 2018.

[37] "Docker scheduling strategy." https://semaphoreci.com/community/tutorials/scheduling-services-on-a-docker-swarm-mode-cluster. Accessed on 26/04/2021.

[38] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, vol. 16, pp. 113–135, 2018.

[39] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, no. 23, pp. 7741–7752, 2018.

[40] T. Menouer, O. Manad, C. Cérin, and P. Darmon, "Power efficiency containers scheduling approach based on machine learning technique for cloud computing environment," *International Symposium on Pervasive Systems, Algorithms and Networks*, pp. 193—-206, 2019.

[41] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th international conference on software engineering (ICSE)*, pp. 222–232, IEEE, 2012.

[42] M. Nejati, M. Alfadel, and S. McIntosh, "Code review of build system specifications: Prevalence, purposes, patterns, and perceptions," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1213–1224, IEEE, 2023.

[43] M. Mohamed, M. Romdhani, and K. Ghédira, "Classification of model refactoring approaches," *Journal of Object Technology*, vol. 8, no. 6, pp. 121–126, 2009.

[44] J. Ivers, A. Ghammam, K. Gaaloul, I. Ozkaya, M. Kessentini, and W. Aljedaani, "Mind the gap: The disconnect between refactoring criteria used in industry and refactoring recommendation tools," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 138–150, IEEE, 2024.

[45] A. Ghammam, R. Khalsi, M. Kessentini, and F. Hassan, "Efficient management of containers for software defined vehicles," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–36, 2024.

[46] W. Aljedaani, A. Ghammam, M. W. Mkaouer, and M. Kessentini, "From boring to boarding: Transforming refactoring education with game-based learning," in *Proceedings of the ACM/IEEE 8th International Workshop on Games and Software Engineering*, pp. 20–27, 2024.

[47] A. Ghammam, T. Ferreira, W. Aljedaani, M. Kessentini, and A. Husain, "Dynamic software containers workload balancing via many-objective search," *IEEE Transactions on Services Computing*, 2023.

[48] Docker. `https://docker.com`, 2020. Accessed on 18/05/2021.

[49] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, pp. 323–333, 2017.

[50] Docker Compose, 2020. `https://github.com/docker/compose`.

[51] DockerHub, 2020. `https://hub.docker.com`.

[52] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[53] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, Aug. 2002.

[54] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *International conference on parallel problem solving from nature*, pp. 832–842, Springer, 2004.

[55] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.

[56] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, pp. 577–601, Aug 2014.

[57] H. Jain and K. Deb, "An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach," *IEEE Transactions on Evolutionary Computation*, vol. 18, pp. 602–622, Aug 2014.

[58] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *SIGSOFT Softw. Eng. Notes*, vol. 38, p. 51–54, Aug. 2013.

[59] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[60] J. Ivers, R. L. Nord, I. Ozkaya, C. Seifried, C. S. Timperley, and M. Kessentini, "Industry experiences with large-scale refactoring," ESEC/FSE 2022, (New York, NY, USA), p. 1544–1554, Association for Computing Machinery, 2022.

[61] S. Duncan, “The thoughtworks® anthology, essays on software technology and innovation,” *Software Quality Professional*, vol. 11, no. 2, p. 53, 2009.

[62] B. Muschko, *Gradle in Action*. Simon and Schuster, Feb 2014.

[63] G. B. T. 2023, “`https://gradle.org/`,” Online; accessed on 22 September 2023.

[64] C. MacNeill, “Apache ant - welcome.”

[65] B. Porter, “Maven – welcome to apache maven.”

[66] K. Kaewkasi, C.and Chuenmuneewong, “Improvement of container scheduling for docker using ant colony optimization,” *9th International Conference on Knowledge and Smart Technology*, no. 7886112, pp. 254–259, 2017.

[67] G. Fan, L. Chen, H. Yu, and W. Qi, “Multi-objective optimization of container-based microservice scheduling in edge computing,” *Computer Science and Information Systems*, vol. 18, no. 1, pp. 23–42, 2021.

[68] A. Aleti, “Designing automotive embedded systems with adaptive genetic algorithms,” *Automated Software Engineering*, vol. 22, pp. 199–240, 2015.

[69] F. Chen, X. Zhou, and C. Shi, “The container scheduling method based on the min-min in edge computing,” *ACM International Conference Proceeding Series*, pp. 83—-90, 2019.

[70] F. Zhang, Z. Tang, J. Lou, and W. Jia, “Online joint scheduling of delay-sensitive and computation-oriented tasks in edge computing in edge computing,” *15th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, 2019.

[71] T. J. Dea, *Mining, Understanding and Integrating User Preferences in Software Refactoring Using Computational Search, Machine Learning, and Dimensionality Reduction*. PhD thesis, 2017.

[72] T. N. Ferreira, S. R. Vergilio, and J. T. de Souza, "Incorporating user preferences in search-based software engineering: A systematic mapping study," *Information and Software Technology*, vol. 90, pp. 55–69, 2017.

[73] I. Griffith, S. Wahl, and C. Izurieta, "TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility," in *Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering (CAINE)*, p. 1, 2011.

[74] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[75] C. Abid, M. Kessentini, V. Alizadeh, M. Dhaouadi, and R. Kazman, "How does refactoring impact security when improving quality? a security-aware refactoring approach," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 864–878, 2020.

[76] E. Koc, N. Ersoy, A. Andac, Z. S. Camlidere, I. Cereci, and H. Kilic, "An empirical study about search-based refactoring using alternative multiple and population-based search techniques," in *Computer and Information Sciences II* (E. Gelenbe, R. Lent, and G. Sakellari, eds.), (London), pp. 59–66, Springer London, 2012.

[77] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Information and Software Technology*, vol. 54, no. 12, pp. 1297–1307, 2012.

[78] M. Mohan, D. Greer, and P. McMullan, "Technical debt reduction using search based automated refactoring," *Journal of Systems and Software*, vol. 120, pp. 183–194, 2016.

[79] R. Hardt and E. V. Munson, "Ant build maintenance with formiga," in *2013 1st International Workshop on Release Engineering (RELENG)*, pp. 13–16, IEEE, 2013.

[80] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, pp. 1–10, 2014.

[81] T. Xiao, D. Wang, S. McIntosh, H. Hata, R. G. Kula, T. Ishio, and K. Matsumoto, "Characterizing and mitigating self-admitted technical debt in build systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4214–4228, 2021.

[82] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, pp. 113–135, 2017.

[83] L. Li, J. Chen, and W. Yan, "A particle swarm optimization-based container scheduling algorithm of docker platform," in *Proceedings of the 4th International Conference on Communication and Information Processing*, pp. 12–17, 2018.

[84] D. Zhang, B.-H. Yan, Z. Feng, C. Zhang, and Y.-X. Wang, "Container oriented job scheduling using linear programming model," in *Proceedings of the 3rd International Conference on Information Management (ICIM '17)*, (Chengdu, China), pp. 174–180, IEEE, 2017.

[85] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–45, 2015.

[86] D. H. Phan and J. Suzuki, "R2-ibea: R2 indicator based evolutionary algorithm for multiobjective optimization," in *2013 IEEE Congress on Evolutionary Computation*, pp. 1836–1845, IEEE, 2013.

[87] Anwar Ghammam, Thiago Ferreira, Wajdi Aljedaani, Marouane Kessentini, and Ali Husain, 2023. `https://anwarghammam.github.io/ContainersSchedulingWebSite/`.

[88] M. Sureshkumar and P. Rajesh, "Optimizing the docker container usage based on load scheduling," *2nd International Conference on Computing and Communications Technologies, ICCCT 2017; Chennai,*, no. 7972269, pp. 165–168, 2017.

[89] S. Meisenbacher, K. Schwenk, J. Galenzowski, S. Waczowicz, R. Mikut, and V. Hagenmeyer, "A lightweight user interface for smart charging of electric vehicles: A real-world application," in *2021 9th International Conference on Smart Grid and Clean Energy Technologies (ICSGCE)*, pp. 57–61, 2021.

[90] R. Morabito, R. Petrolo, V. Loscrì, N. Mitton, G. Ruggeri, and A. Molinaro, "Lightweight virtualization as enabling technology for future smart cars," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1238–1245, 2017.

[91] S. Dabbene, C. Lehmann, C. Campolo, A. Molinaro, and F. H. P. Fitzek, "A mec-assisted vehicle platooning control through docker containers," in *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*, pp. 1–6, 2020.

[92] T. Mikkonen, C. Pautasso, K. Systä, and A. Taivalsaari, "Cargo-cult containerization: A critical view of containers in modern software development," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 93–98, IEEE, 2022.

[93] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *2017 9th international conference on knowledge and smart technology (KST)*, pp. 254–259, IEEE, 2017.

[94] ApacheBench, 2020. `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[95] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.

[96] cAdvisor, 2020. `https://github.com/google/cadvisor`.

[97] E. Zitzler, M. Laumanns, and S. Bleuler, "A tutorial on evolutionary multiobjective optimization," *Metaheuristics for multiobjective optimisation*, pp. 3–37, 2004.

[98] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[99] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," tech. rep., Citeseer, 1998.

[100] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, "Not going to take this anymore: multi-objective overtime planning for software engineering projects," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 462–471, IEEE, 2013.

[101] H. Meunier, E.-G. Talbi, and P. Reininger, "A multiobjective genetic algorithm for radio network optimization," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, vol. 1, pp. 317–324, IEEE, 2000.

[102] R. A. Matnei Filho and S. R. Vergilio, "A mutation and multi-objective test data generation approach for feature testing of software product lines," in *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES'15)*, (Belo Horizonte, Brazil), pp. 21–30, IEEE Computer Society, Sept. 2015.

[103] A. Strickler, J. A. Prado Lima, S. R. Vergilio, and A. Pozo, "Deriving products for variability test of feature models with a hyper-heuristic approach," *Applied Soft Computing*, vol. 49, pp. 1232–1242, Dec. 2016.

[104] T. N. Ferreira, J. N. Kuk, A. Pozo, and S. R. Vergilio, "Product selection based on upper confidence bound MOEA/D-DRA for testing software product lines," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '16)*, (Vancouver, Canada), pp. 4135–4142, IEEE, July 2016.

[105] K. Deb, "Multi-objective optimisation using evolutionary algorithms: an introduction," in *Multi-objective evolutionary optimisation for product design and manufacturing*, pp. 3–34, Springer, 2011.

[106] B. Rosner, R. J. Glynn, and M.-L. Ting Lee, "Incorporation of clustering effects for the wilcoxon rank sum test: a large-sample approach," *Biometrics*, vol. 59, no. 4, pp. 1089–1098, 2003.

[107] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1–10, IEEE, 2011.

[108] N. Ayres, L. Deka, and D. Paluszczyszyn, "Continuous automotive software updates through container image layers," *Electronics*, vol. 10, no. 6, p. 739, 2021.

[109] D. Lennick, A. Azim, and R. Liscano, "Container-based internet-of-things architecture pattern: Kill switch," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 74–78, IEEE, 2020.

[110] P. Varaiya, "Smart cars on smart roads: problems of control," *IEEE Transactions on Automatic Control*, vol. 38, no. 2, pp. 195–207, 1993.

[111] F. Arena, G. Pau, and A. Severino, "An overview on the current status and future perspectives of smart cars," *Infrastructures*, vol. 5, no. 7, 2020.

[112] I. Yaqoob, L. U. Khan, S. M. A. Kazmi, M. Imran, N. Guizani, and C. S. Hong, "Autonomous driving cars in smart cities: Recent advances, requirements, and challenges," *IEEE Network*, vol. 34, no. 1, pp. 174–181, 2020.

[113] D. Yang, K. Jiang, D. Zhao, C. Yu, Z. Cao, S. Xie, Z. Xiao, X. Jiao, S. Wang, and K. Zhang, "Intelligent and connected vehicles: Current status and future perspectives," *Science China Technological Sciences*, vol. 61, no. 10, pp. 1446–1471, 2018.

[114] M. Cebe, E. Erdin, K. Akkaya, H. Aksu, and S. Uluagac, "Block4forensic: An integrated lightweight blockchain framework for forensics applications of connected vehicles," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 50–57, 2018.

[115] B. Xu, X. J. Ban, Y. Bian, W. Li, J. Wang, S. E. Li, and K. Li, "Cooperative method of traffic signal optimization and speed control of connected vehicles at isolated intersections," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 4, pp. 1390–1403, 2018.

[116] Z. Mahmood, *Connected Vehicles in the Internet of Things*. Springer, 2020.

[117] R. Okuda, Y. Kajiwara, and K. Terashima, "A survey of technical trend of adas and autonomous driving," in *Proceedings of Technical Program - 2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pp. 1–4, 2014.

[118] K. Brookhuis, D. Waard, and W. Janssen, "Behavioural impacts of advanced driver assistance systems-an overview," *European Journal of Transport and Infrastructure Research*, vol. 1, 01 2001.

[119] A. Jarašūniene, "Research into intelligent transport systems (its) technologies and efficiency," *Transport*, vol. 22, no. 2, pp. 61–67, 2007.

[120] R. Buyya, "Market-oriented cloud computing: Vision, hype, and reality of delivering computing as the 5th utility," in *2009 Fourth ChinaGrid Annual Conference*, pp. xii–xv, IEEE, 2009.

[121] W. Ribbens, *Understanding automotive electronics: an engineering perspective*. Butterworth-heinemann, 2017.

[122] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016.

[123] G. Sayfan, *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[124] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, vol. 11, pp. 22–22, 2011.

[125] R. Rana, M. Staron, N. Mellegård, C. Berger, J. Hansson, M. Nilsson, and F. Törner, "Evaluation of standard reliability growth models in the context of automotive software systems," in *Product-Focused Software Process Improvement: 14th*

International Conference, PROFES 2013, Paphos, Cyprus, June 12-14, 2013. Proceedings 14, pp. 324–329, Springer, 2013.

[126] S. Duri, M. Gruteser, X. Liu, P. Moskowitz, R. Perez, M. Singh, and J.-M. Tang, "Framework for security and privacy in automotive telematics," in *Proceedings of the 2nd international workshop on Mobile commerce*, pp. 25–32, 2002.

[127] node-exporter, 2020. `https://prometheus.io/docs/guides/node-exporter/`.

[128] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 278–287, IEEE, 2019.

[129] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty, "Security challenges in automotive hardware/software architecture design," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 458–463, IEEE, 2013.

[130] H. Mun, K. Han, and D. H. Lee, "Ensuring safety and security in can-based automotive embedded systems: A combination of design optimization and secure communication," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7078–7091, 2020.

[131] H. Vdovic, J. Babic, and V. Podobnik, "Automotive software in connected and autonomous electric vehicles: A review," *IEEE Access*, vol. 7, pp. 166365–166379, 2019.

[132] C. Buckl, A. Camek, G. Kainz, C. Simon, L. Mercep, H. Stähle, and A. Knoll, "The software car: Building ict architectures for future electric vehicles," in *2012 IEEE International Electric Vehicle Conference*, pp. 1–8, IEEE, 2012.

[133] T. Menouer, "Kcss: Kubernetes container scheduling strategy," *The Journal of Supercomputing*, vol. 77, no. 5, pp. 4267–4293, 2021.

[134] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.

[135] V. Alizadeh, H. Fehri, and M. Kessentini, "Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 181–192, IEEE, 2019.

[136] "Docker scheduling strategy." `https://semaphoreci.com/community/tutorials/scheduling-services-on-a-docker-swarm-mode-cluster`. Accessed on 26/04/2021.

[137] Z. B. Zabinsky *et al.*, "Random search algorithms," *Department of Industrial and Systems Engineering, University of Washington, USA*, 2009.

[138] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *2013 35Th international conference on software engineering (ICSE)*, pp. 492–501, IEEE, 2013.

[139] N. Naik, "Applying computational intelligence for enhancing the dependability of multi-cloud systems using docker swarm," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–7, IEEE, 2016.

[140] T. Haendler. and J. Frysak., "Deconstructing the refactoring process from a problem-solving and decision-making perspective," in *Proceedings of the 13th International Conference on Software Technologies - ICSOFT*, pp. 363–372, INSTICC, SciTePress, 2018.

[141] J. Brant and F. Steimann, "Refactoring tools are trustworthy enough and trust must be earned," *IEEE Software*, vol. 32, no. 6, pp. 80–83, 2015.

[142] M. Paixão, A. Uchôa, A. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in *Proceedings of the 17th International Conference on Mining Software Repositories*, p. 125–136, 2020.

[143] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[144] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[145] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, 2016.

[146] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, (New York, NY, USA), p. 49–58, Association for Computing Machinery, 2012.

[147] R. L. Nord, I. Ozkaya, R. S. Sangwan, and R. J. Koontz, "Architectural dependency analysis to understand rework costs for safety-critical systems," ICSE Companion 2014, (New York, NY, USA), p. 185–194, ACM, 2014.

[148] "Additional materials." `https://helloreviewer.github.io/ICSME24/#`.

[149] M. Ciolkowski, O. Laitenberger, S. Vegas, and S. Biffl, *Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering*, pp. 104–128. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[150] R. Dijkman, B. Gfeller, J. Küster, and H. Völzer, "Identifying refactoring opportunities in process model repositories," *Information and Software Technology*, vol. 53, no. 9, pp. 937–948, 2011.

[151] B. Alkhazi, T. Ruas, M. Kessentini, M. Wimmer, and W. I. Grosky, "Automated refactoring of ATL model transformations: A search-based approach," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[152] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *2007 ieee international conference on software maintenance*, pp. 519–520, IEEE, 2007.

[153] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1037–1039, 2011.

[154] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "JDeodorant: clone refactoring," in *Proceedings of the 38th international conference on software engineering companion*, pp. 613–616, 2016.

[155] N. Tsantalis, "JDeodorant: Clone refactoring support beyond ides," *IEEE Software Blog*, 2016.

[156] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, no. 3, pp. 391–404, 2010.

[157] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 119–128, IEEE, 2009.

[158] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, "Finding the best compromise between design quality and testing effort during refactoring," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1, pp. 24–35, IEEE, 2016.

[159] M. Vimaladevi and G. Zayaraz, "Stability aware software refactoring using hybrid search based techniques," in *2017 International Conference on Technical Advancements in Computers and Communications (ICTACC)*, pp. 32–35, IEEE, 2017.

[160] A.-R. Han, D.-H. Bae, and S. Cha, "An efficient approach to identify multiple and independent move method refactoring candidates," *Information and Software Technology*, vol. 59, pp. 53–66, 2015.

[161] J. Ivers, C. Seifried, and I. Ozkaya, "Untangling the knot: Enabling architecture evolution with search-based refactoring," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pp. 101–111, IEEE, 2022.

[162] M. Di Penta, "Evolution doctor: a framework to control software system evolution," in *Ninth European Conference on Software Maintenance and Reengineering*, pp. 280–283, Ieee, 2005.

[163] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 60–70, IEEE, 2017.

[164] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, pp. 1–53, 2016.

[165] A. S. Nyamawe, H. Liu, Z. Niu, W. Wang, and N. Niu, "Recommending refactoring solutions based on traceability and code metrics," *IEEE Access*, vol. 6, pp. 49460–49475, 2018.

[166] D. Cui, S. Wang, Y. Luo, X. Li, J. Dai, L. Wang, and Q. Li, "RMove: Recommending move method refactoring opportunities using structural and semantic representations of code," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 281–292, IEEE, 2022.

[167] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 535–546, 2016.

[168] I. H. Moghadam and M. Ó. Cinnéide, "Automated refactoring using design differencing," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 43–52, IEEE, 2012.

[169] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, vol. 45, no. 3, pp. 315–342, 2015.

[170] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "Jmove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19–36, 2018.

[171] N. Ujihara, A. Ouni, T. Ishio, and K. Inoue, "c-JRefRec: Change-based identification of move method refactoring opportunities," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 482–486, IEEE, 2017.

[172] D. Steidl and S. Eder, "Prioritizing maintainability defects based on refactoring recommendations," in *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 168–176, 2014.

[173] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, "Proactive clone recommendation system for extract method refactoring," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 67–70, IEEE, 2019.

[174] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: two industrial case studies," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1303–1310, 2017.

[175] S. Vidal, I. Berra, S. Zulliani, C. Marcos, and J. A. D. Pace, "Assessing the refactoring of brain methods," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 1, pp. 1–43, 2018.

[176] N. Niu, T. Bhowmik, H. Liu, and Z. Niu, "Traceability-enabled refactoring for managing just-in-time requirements," in *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pp. 133–142, IEEE, 2014.

[177] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of move method refactoring using path-based representation of code," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 315–322, 2020.

[178] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y.-G. Guéhéneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *2010 IEEE International Conference on Software Maintenance*, pp. 1–5, IEEE, 2010.

[179] M. M. Rahman, R. R. Riyadh, S. M. Khaled, A. Satter, and M. R. Rahman, "Mmruc3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design," *Software: Practice and Experience*, vol. 48, no. 9, pp. 1560–1587, 2018.

[180] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 253–258, IEEE, 2015.

[181] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig, "Together we go further: LLMs and IDE static analysis for extract method refactoring," 2024.

[182] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 38–47, 2017.

[183] S. McIntosh, B. Adams, and A. E. Hassan, "The Evolution of ANT Build Systems," in *Proc. of the Working Conference on Mining Software Repositories (MSR)*, pp. 42–51, 2010.

[184] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An Empirical Study of Build Maintenance Effort," in *Proc. of the International Conference on Software Engineering (ICSE)*, pp. 141–150, 2011.

[185] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," 2016.

[186] S. Fernandes and J. Bernardino, "What is bigquery?," in *Proceedings of the 19th International Database Engineering & Applications Symposium*, pp. 202–203, 2015.

[187] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 348–357, IEEE, 2021.

[188] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. D. Newman, and A. Ouni, "Behind the scenes: On the relationship between developer experience and refactoring," *Journal of Software: Evolution and Process*, vol. 36, no. 1, p. e2395, 2024.

[189] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. D. Newman, and A. Ouni, "An exploratory study on refactoring documentation in issues handling," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 107–111, 2022.

[190] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, pp. 1192–1223, 2016.

[191] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Systems with Applications*, vol. 167, p. 114176, 2021.

[192] S. Levin and A. Yehudai, "Towards software analytics: Modeling maintenance activities," *arXiv preprint arXiv:1903.04909*, 2019.

[193] "Additional material." `https://sites.google.com/view/build-code-refactoring/home`. Accessed on 03/17/2022.

[194] M. Usman, R. Britto, J. Börstler, and E. Mendes, "Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method," *Information and Software Technology*, vol. 85, pp. 43–59, 2017.

[195] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews," *Sociological Methods & Research*, vol. 42, p. 294–320, Aug. 2013.

[196] S. Fincher and J. Tenenberg, "Making sense of card sorting data," *Expert Systems*, vol. 22, no. 3, pp. 89–93, 2005.

[197] D. Spencer and T. Warfel, "Card sorting: a definitive guide," *Boxes and arrows*, vol. 2, no. 2004, pp. 1–23, 2004.

[198] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, "An empirical study of refactorings and technical debt in machine learning systems," in *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*, pp. 238–250, IEEE, 2021.

[199] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE software*, vol. 12, no. 4, pp. 52–62, 1995.

[200] C. Pacheco and I. Garcia, "Stakeholder identification methods in software requirements: empirical findings derived from a systematic review," in *2008 The Third International Conference on Software Engineering Advances*, pp. 472–477, IEEE, 2008.

[201] M. Ciolkowski, O. Laitenberger, S. Vegas, and S. Biffl, *Practical experiences in the design and conduct of surveys in empirical software engineering*. Springer, 2003.

[202] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[203] R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level coding using llms and planning," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 675–698, 2024.

[204] Z. Li, C. Wang, P. Ma, C. Liu, S. Wang, D. Wu, C. Gao, and Y. Liu, "On extracting specialized code abilities from large language models: A feasibility study," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[205] R. Patil and V. Gudivada, "A review of current trends, techniques, and challenges in large language models (llms)," *Applied Sciences*, vol. 14, no. 5, p. 2074, 2024.

[206] L. Giray, "Prompt engineering with chatgpt: a guide for academic writers," *Annals of biomedical engineering*, vol. 51, no. 12, pp. 2629–2633, 2023.

[207] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.

[208] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[209] J. Bauer, "Make the featurefactory an option get the parser's objects from the... stanfordnlp/corenlp@ 616d524," *Stanford NLP*, 2014.

[210] "Gradle refactoring. · zoltu/stubkafkabroker@489dac4."

[211] "Refactored build.gradle · pixplicity/gene-rate@1800eae."

[212] "refactored the maven pom.xml layout to include modules for parent · thrasher/f8api."

[213] "Gradle build refactoring · hcuffy/concourse@13555a8."

[214] "Refs 1514 build.xml."

[215] “Refactoring gradle kotlin configuration to root multiproject build.gr. . . · atrujillofalcon/spring-guides-kotlin@8732237.”

[216] “changed pom.xml – refactoring · treasure-data/td-logger-java@4e9590c.”

[217] “Refactor sample module · farbodsz/usefulviews@70e7b00.”

[218] “Refactorized build.xml.”

[219] “Refactored examples to use a parent pom. reduced duplicate pom xml co. . . · matthewmccullough/encryption-jvm-bootcamp@11840e3.”

[220] “build.gradle.”

[221] “refactor.”