Towards an Enhanced Dependency Graph

by

Seyed Mehran Meidani

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Dependency graphs are at the heart of software analytics tasks like change impact analysis, test selection, and maintenance analysis. Despite their importance, current approaches to extract and analyze dependency graphs overlook configuration settings and code-adjacent artifacts in large software systems. These shortcomings directly affect the results of the aforementioned analytics tasks. Indeed, changing a software application with many build-time configuration settings may introduce unexpected side effects. For example, a change intended to be specific to a platform (e.g., Windows) or product configuration (e.g., community edition) might impact other platforms or configurations. Moreover, a change intended to apply to a set of platforms or configurations may be unintentionally limited to a subset of platforms. In addition to build-time configuration settings, software projects require a broad range of expertise to develop. For example, to produce a video game, engineers, like software developers, and artists, like 3D designers, must iterate on the same project simultaneously. In such projects, a change to the work products of any of the teams can impact the work of other teams. As a result, any analytics tasks should consider intra- and inter-dependencies among artifacts produced by different teams. For instance, the focus of the quality assurance team for a change local to a team differs from one that impacts others.

Indeed, understanding the exposure of changes is an important risk mitigation step in change-based development approaches. In this thesis, we first present DiPiDi, a new approach to assess the exposure of source code changes under different build-time configuration settings by statically analyzing build specifications. To evaluate our approach, we produce a prototype implementation of DiPiDi for the CMake build system. We measure the effectiveness and efficiency of developers when performing five tasks in which they must identify the deliverable(s) and conditions under which a source code change will propagate. We assign participants into three groups: without explicit tool support, supported by existing impact analysis tools, and supported by DiPiDi. While our study does not have the statistical power to make generalized quantitative claims, we manually analyze the full distribution of our study's results and show that DiPiDi results in a net benefit for its users. Through our experimental evaluation, we show that DiPiDi is associated with a 36 percentage point improvement in F_1-score on average when identifying impacted deliverables and an average reduction of 0.62 units of distance when ranking impacted patches. Furthermore, DiPiDi results in a 42% average task time reduction for our participants when compared to a competing impact analysis approach. DiPiDi's improvements to both effectiveness and efficiency are especially prevalent in complex programs with many compile-time configurations.

Next, to extract and analyze cross-disciplinary dependencies, we propose a multidisciplinary dependency graph. We instantiate our idea by developing tools that extract dependencies and construct the graph at a multinational video game organization with more than 18,000 employees. Our analysis of the historical data from a recently launched video game project demonstrates that 41% of the studied source code changes impact other teams' artifacts, highlighting the importance of analyzing inter-artifact dependencies. We also observe that 66% of the studied changes do not modify the graph, suggesting that prior graph versions are often accurate for analytics tasks (*e.g.,* impact analysis); however, rapid incremental approaches are needed to update the graph and ensure its usefulness for all types of changes.

The enhanced dependency graph presented in this thesis can be leveraged to develop a new generation of risk assessment, build failure prediction, and code review prioritization tools.

Acknowledgements

This thesis would not have been possible without the help, support and guidance of many people. First and foremost, I am extremely grateful to my supervisor, Dr. Shane McIntosh, for his support and help during my study at his lab. His immense knowledge and experience have encouraged me throughout my academic research and daily life. I would also like to thank Dr. Maxime Lamothe for providing guidance and feedback throughout my study.

I am also grateful to Dr. Sarra Habchi and Mathieu Nayrolles for allowing me to research at Ubisoft and guiding me throughout my internship there. Special thanks to Dr. Mei Nagappan and Dr. Michael Godfrey for their time reading this thesis and giving me constructive feedback.

I wanted to thank the brilliant students at the Software Repository Excavation and Build Engineering Labs (The Software REBELs) who motivated me to work on this thesis; including Mahmoud, Farshad, Mahtab, Nimmi, Mingyang, Zhili, Gengyi, Sean, Eve, Arsalan, Wen, and Gareema. Thanks should also go to Rasoul Akhavan Mahdavi and Ali Abyaneh for being amazing housemates and friends.

Most importantly, none of this could have happened without my family. My grandmother, who raised me and my mother, for her incredible support and guidance.

Dedication

This is dedicated to all the brave Iranians who are fighting for their freedom, to Mahsa Amini, and to Woman, Life, Freedom.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development is a complex endeavour. Various software artifacts need to be carefully developed in order to produce a software system. Source code, which describes system behaviour, is the software artifact that is traditionally associated with software. However, source code is not the only software artifact. Indeed, test code which is used to verify the system by exercising it using simulated conditions, and non-source code artifacts such as machine learning models, user interface objects, and infrastructure specification files are also commonly part of software systems. To weave these artifacts into a cohesive system, software organizations rely on *build systems*. These build systems specify and resolve internal and external dependencies and the conditions under which they should be used. In addition, build systems orchestrate the invocation of order-dependent commands that preprocess, compile, assemble, link, analyze, and package software artifacts into deliverables [26].

At their core, build systems specify and reason about which commands should be invoked using a *dependency graph*, *i.e.,* a directed graph where nodes represent software entities and directed edges indicate dependencies between artifacts. While the dependency graph is crucial to build system execution, it is also useful for performing other sorts of software analyses, such as failure prediction [87], maintenance analysis [7], quality improvement [39], and impact analysis [80].

Complex software programs employ many compile-time configuration settings to build different software products (a.k.a., variants) from the same artifacts (i.e., source files) [78]. For example, the Linux kernel has more than 10,000 compile-time configuration settings [74]. Additionally, in software development projects that involve personnel from different disciplines, the breadth of software artifacts can be vast [82]. For example, producing high-

budget video games ('AAA games') requires the careful coordination of personnel with divergent expertise, such as technical software staff (e.g., developers, QA, and operators), as well as creative staff (e.g., graphic artists, composers and musicians, script writers, and level designers). AAA games are typically composed of millions of lines of code, as well as hundreds of thousands of non-code files like textures and animations [61].

1.1 Problem Statement

Since a build dependency graph shows dependency relationships between files in a software project, it can be used to assess the risk of software changes. However, in complex software programs with many compile-time configuration settings or code-adjacent artifacts, current approaches that extract and analyse the build dependency graph are incomplete:

Thesis Statement: A build dependency graph that captures build-time configuration settings and code-adjacent artifacts can be leveraged to accurately assess the risk of software changes.

Change Impact Analysis (CIA) is one way to determine the consequences of a change on a software application [12]. Many CIA techniques have been proposed [7, 8, 29, 35, 49, 76]. However, to the best of our knowledge, prior approaches do not consider build-time configuration settings or artifacts other than source files in the project. While build impact analysis has been shown to be effective [7, 81], current techniques rely on a dynamic analysis of build execution, which cannot expose the impact of a change on different environmental and configuration settings. Furthermore, these techniques focus only on source files. Hence, in this thesis, we explain how to enhance the build dependency graph by 1) statically analyzing the build description files, and 2) extracting code-adjacent artifact dependency relationships.

1.2 Thesis Overview

We now provide a brief overview of the thesis. Figure 1.1 provides an overview of the scope of this thesis. We first provide the necessary background for our topic:

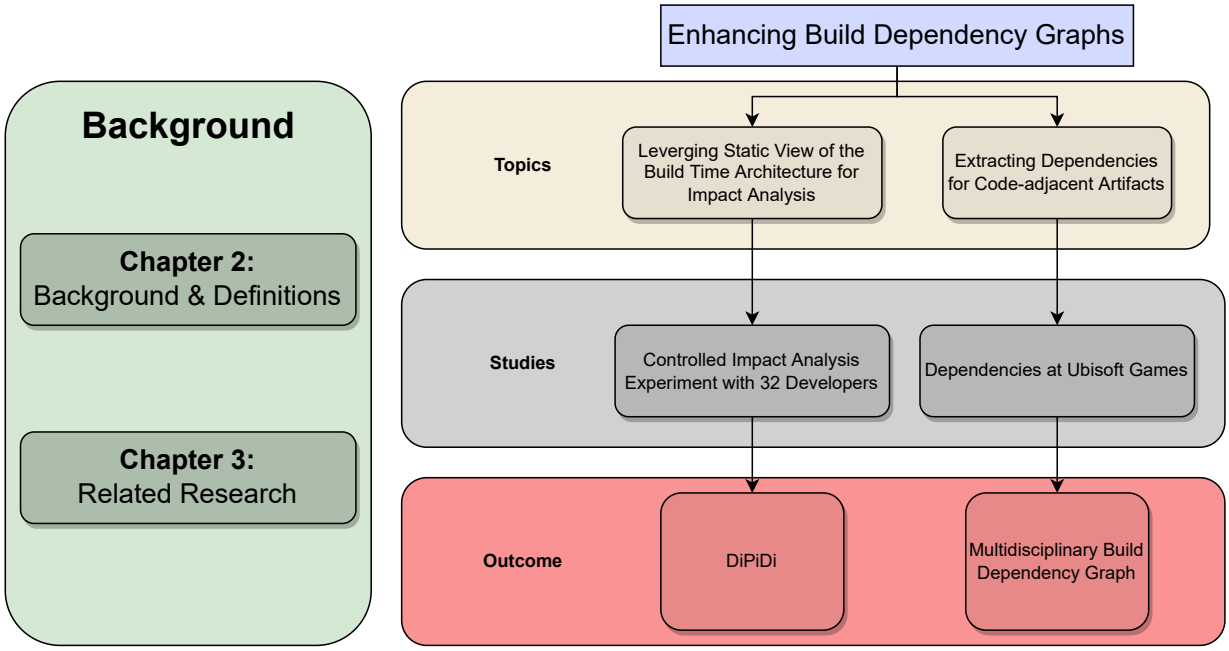


Figure 1.1: An overview of the scope of this thesis

Chapter 2: *Background and Definitions*
Before discussing how to enhance the build dependency graph, we first provide readers with background information and definitions of terms that we use throughout this thesis.

Chapter 3: *Related Work*
To situate this thesis with prior studies, we present a survey of research on build dependency graphs.

Next, we shift our focus to the main part of this thesis. In this thesis, we focus on how to improve the build dependency graph by considering build-time configuration settings through static analysis of build files and adding code-adjacent artifacts to the graph. Each empirical study is presented in its own chapter, as explained in the subsection below.

1.3 Statically Analyzing Build Files

Complex software programs have multiple dependency paths to their source files from their *deliverables*, i.e., software artifacts that users can interact with, such as executable files or libraries. Build systems derive default configuration settings by analyzing the execution environment or reading user override settings. Build systems use these settings to reason about whether source files (or conditionally compiled code snippets) should be included or excluded from the produced deliverables. Under some conditions, a source file may play a role in one compiled deliverable without affecting others. For example, in the Linux kernel, the source files written specifically for the ARM architecture will be excluded from the x86 version of the kernel [59]. In these complex systems, a change in a source file may have unexpected side effects on deliverables outside of the current compilation path. Software systems that support multiple variants can therefore create complex arrangements of effects and side effects, where the deliverables exposed to a code-change can be unclear [15]. In this thesis, we propose an approach to assess the impact of software changes to the source code of systems using the build system specification files. One of the key roles of the build system is finding and selecting files based on build scripts, build-time configurations, and environmental variables [9, 73, 86]. By statically analyzing the build system and constructing the *Build Dependency Graph* (BDG), we can assess the exposure of a change on all software variants.

Chapter 4: *Assessing the Exposure of Software Changes, The DiPiDi Approach*
To evaluate the proposed approach, we conduct an experiment to assess the effect of our approach on the effectiveness and efficiency of determining the exposure of source code changes on projects that are using CMake build system. To that end, we recruit 32 participants and form three participant groups – those with no tool assistance, those with the assistance of a CIA tool, and those with the assistance of a prototype implementation of our approach called DiPiDi – and compare their efficiency and effectiveness on prescribed tasks.

1.4 Code-Adjacent Artifacts

Software projects require a variety of expertise to develop. For example, to produce a video game, engineers, like software developers, and artists, like 3D designers, should work on the same project simultaneously. A change to any of the artifacts (*i.e.,* work products) of any of the teams can impact the work of other teams. As a result, any analytics tasks

should consider intra- and inter-dependencies among artifacts produced by different teams. To this end, in this thesis, we introduce the multidisciplinary dependency graph.

Chapter 5: *Dependency Extraction and Analysis for Multidisciplinary Teams: A Case Study at Ubisoft*
Multidisciplinary teams require a multidisciplinary dependency graph. Consider a change to a source code file that repositions an object in a game. This repositioning may have a transitive impact on other objects within the location in the game. To trace the impact of that change, we need a graph that captures the dependencies in code, data, and their interdependencies. While dependency graphs have been explored in the general development context [7, 39, 87], the multidisciplinary software context introduces challenges in the extraction and analysis of dependency graphs that need to be addressed. In this chapter, we show how such a graph can be extracted from a large video game project and explore properties of the extracted graph.

1.5 Thesis Contributions

This thesis has both technical and empirical contributions as described below:

1.5.1 Technical Contributions

To evaluate the approaches proposed in this thesis, we develop following tools:

1. A prototype implementation of our approach to statically extract and analyze the build dependency graph from CMake build system specification files containing all the environmental and configuration settings (Chapter 4).
2. A script to find dependency relationships in code-adjacent artifacts and add the corresponding nodes and edges to the dependency graph (Chapter 5).

1.5.2 Empirical Contributions

This thesis shows that:

1. Considering build-time configuration settings by statically analyzing build specification files improves the efficiency and effectiveness of developers assessing the impact of software changes (Chapter 4).

2. Changes to code and code-adjacent artifacts do not modify the multidisciplinary graph in 66% of the cases, suggesting that the graph can be used for analytics tasks; however, rapid incremental approaches are needed to update the graph and ensure its usefulness for all types of changes (Chapter 5).

3. Extracting and analyzing inter-artifact dependencies using a multidisciplinary dependency graph is important because 41% of the changes in the code files affect other artifacts as well (Chapter 5).

1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background knowledge and definitions of key terms. Chapter 3 surveys the related research on build dependency graphs. Chapter 4 presents the results of an experiment to assess the impact of our proposed approach on statically analyzing the build dependency graph and build-time configuration settings. Chapter 5 presents an approach and evaluation of a dependency graph which contains code-adjacent files as well as the code files. Finally, Chapter 6 draws conclusions and discusses paths for future work.

Chapter 2

Background and Definitions

In this chapter, we define the core concepts of a build system. First, we provide an overview of building a software system, and then we dive deep into the build system and its internal data structure.

2.1 Build System

A *build system* is a program that uses a set of specifications that outlines how a software system is assembled from its development artifacts (*e.g.,* source files). The build system plays a critical role in modern software development. It is used in testing, packaging, and deploying a software program. In the remainder of this section, we describe the internal architecture of build systems in more detail.

2.1.1 Build Specification Files

Build specification files describe necessary commands and the order in which they must be executed in order to produce the final product correctly. Each build system has its own specification language. Figure 2.1 shows a sample of a specification file for the CMake build system [44].

```
if(FEATURE_CURL)
  add_executable(etl ${CLIENT_SRC} dl_main_curl.c)
else()
  add_executable(etl ${CLIENT_SRC} dl_main_stubs.c)
endif()
```

Figure 2.1: A sample snippet of CMake build script from the ET: Legacy project. In this sample, `etl` is the deliverable, `FEATURE_CURL` is a build configuration, and `CLIENT_SRC` is a variable pointing to a list of zero or more source files.

2.1.2 Deliverables

The goal of the build system is to convert the artifacts in the project into *deliverables*. These deliverables are in the form of executable files or libraries which will later be used by other programs. The terms deliverable and target are often used interchangeably.

2.1.3 Configuration Setting

The *configuration setting* is a set of environmental and user-specified settings provided to the build system. These settings will help the build system to produce different software products (*a.k.a.* variants) from the same artifacts.

2.2 Analyzing Build Specification Files

Build specification files contain dependency information between software artifacts in a project. This information can be leveraged for software analytics tasks such as impact analysis and build failure prediction. In the remainder of this section, we explain how to parse, extract, and analyze these files.

2.2.1 Abstract Syntax Tree

The *Abstract Syntax Tree (AST)* is a connected acyclic graph that represents the structure of a file and is the output of the parser. This data structure has been widely used by programming languages and software engineering tools [13, 85].

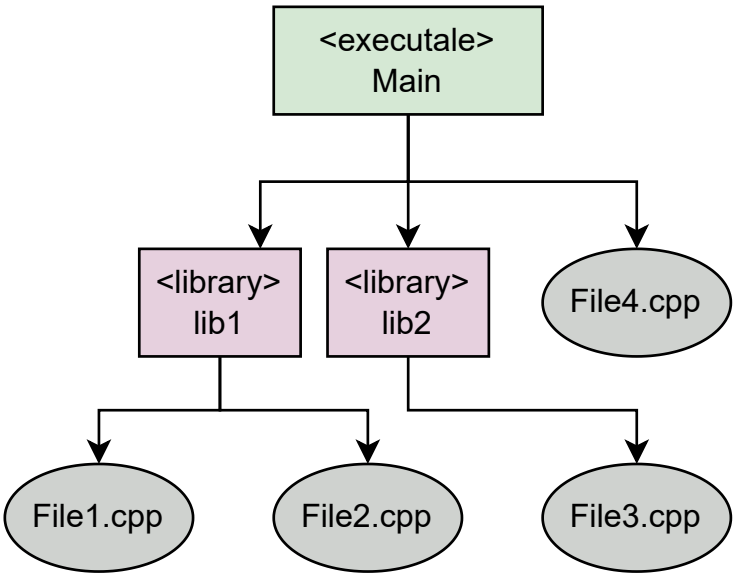


Figure 2.2: A sample of the build dependency graph, which shows the Main executable that depends on two libraries and a file. Additionally, lib1 depends on two files.

Every build system has its own entry file for anchoring the dependency graph construction. For example, GNU Make by default expects a file named `Makefile` to exist in the directory in which the command was invoked. The entry file describes how to build the project using the build system-specific language. The project can contain helper build files in other folders or split the entry file and relocate it in other folders. All of those files should be addressed and included in the entry point file. To capture the content of the build files, we parse the entry file and build an Abstract Syntax Tree (AST) using a parser that understands the build system grammar. The output of the parser is an AST for one build specification file.

2.2.2 Build Dependency Graph

In a software project, artifacts may depend on each other. For example, a C++ file may include a header or another source file. At higher levels, an executable target may depend on libraries and source files.

To successfully build a project, its build system should construct the deliverables in the correct order. Building a deliverable earlier than the ones which it depends on will result in a build failure. Thus, build systems use an internal data structure called the

build dependency graph to specify and reason about which commands should be invoked in which order.

The build dependency graph is a directed acyclic graph $BDG = (T, D)$, where graph nodes represent build targets $T = T_{\mathrm{f}} \cup T_{\mathrm{p}}$, where T_{f} is the set of concrete files produced or consumed by the build process, T_{p} is the set of phony targets in the build process, and $T_{\mathrm{f}} \cap T_{\mathrm{p}} = \varnothing$. Directed edges represent dependencies $d(t, t') \in D$ from target t to target t' which means t should be updated when t' changes. Figure 2.2 shows an example build dependency graph. The build system starts processing the nodes in the dependency graph in topological order. For instance, in Figure 2.2, the build system first generates `lib1` and `lib2` before constructing the `Main` executable. Any further changes to children of `lib1` will result in the reconstruction of `Main` executable as well.

2.2.3 Graph Traversal

The build dependency graph shows dependencies from targets to other targets or files. However, in software analytics tasks like impact analysis, we are interested to see how much of a graph will be exposed to a change on a file. Thus, in this thesis, we use the Depth First Search (DFS) graph traversal algorithm to find exposed nodes to a change.

2.3 Chapter Summary

This chapter provides background knowledge of build systems and analyzing their specification files. More specifically, we describe how build systems produce deliverables by parsing the build specification files, considering the configuration settings, and constructing and traversing the build dependency graph.

In the next chapter, we survey prior research on understanding and improving software analytics tasks using the build dependency graph to situate our empirical studies of enhancing the build dependency graph with respect to the broader body of knowledge.

Chapter 3

Related Research

In this section, we situate our study and its results with respect to the literature on the (3.1.1) Co-evolution of Source and Build Code, (3.2.1) Dependency Graph, (3.2.2) Static Analysis of Build Code, and (3.2.3) Dynamic Analysis of Build Code.

3.1 Build System

3.1.1 Co-evolution of Source and Build Code

There are plenty of empirical studies on the relationship between source code and its corresponding build code. These studies have shown that changes to source code files may lead to changes in the build files that are required to build software programs successfully. McIntosh *et al.* [55] showed this relationship and concluded that, like source files, build code evolves and may have defects. Hochstein *et al.* [40] found that 19%–58% of commits change build files only, and 37%–65% of them touch at least one build file. Robles *et al.* [68] found that many commits mainly involve a build file, showing frequent changes to the build procedure. Also, studies have shown the relationship between the complexity of source and build code [6, 54]. However, to the best of our knowledge, no prior work has studied the relationship between source code changes and their exposure under different configuration settings.

3.2 Analysis of Build Code

3.2.1 Dependency Graph

Dependency graphs have been used for software analysis tasks. For example, Ma *et al.* [51] introduced Service Dependency Graph (SDG) to analyze and visualize the dependency relationships between microservices. Dependency graphs have long been at the heart of software build systems. For example, Feldman introduced the Make build system [26], which uses a depth-first search of the file-level dependency graph to keep the program deliverables up to date with their dependencies. Zimmermann *et al.* [87] used complexity metrics extracted from dependency graphs to predict subsystem failures. Since incomplete graphs can produce unreliable analytics, we propose the multidisciplinary dependency graph and demonstrate its importance in video game projects, such as those at Ubisoft.

3.2.2 Static Analysis of Build Code

Build description files are often quite complex, making it difficult for any developer to fully grasp all of their intricacies. Thus, it is often challenging to both identify bad design practices within build files, and to improve them through refactoring efforts. To remedy this situation, tools like SYMake [76] and HireBuild [37] have been proposed in prior works. SYMake is a tool that can discover smells within build-system files and help developers to refactor these files by building a symbolic dependency graph from GNU Make specifications. Hassan *et al.* [37] developed a tool called HireBuild, which automatically fixes buggy build files using a history-driven approach. These studies used properties of build specification files to analyze the build systems themselves. In this thesis, we use build dependency graphs to analyze the impact of the changes on software systems.

3.2.3 Dynamic Analysis of Build Code

Impact analysis of changes has applications both for researchers and practitioners. Wen *et al.* [81] introduced BLIMP Tracer, an approach to integrate impact analysis with code review. They showed that changes that impact critical deliverables may require more reviewing efforts than others. In another study, Cao *et al.* [17] proposed a tool that can estimate the duration of an incremental build using the build dependency graph, history of the builds, and changed files. They created the graph using the output messages generated by GNU Make. MAKAO is a tool developed by Adams *et al.* [7] which focuses on visualizing

Makefile contents to aid in refactoring them using an aspect-oriented approach. However, these studies construct the graph based on a single execution environment and the build-time configurations provided to the build system for that specific invocation. Thus, the generated graph does not include the files and the dependencies for other configurations. The graph generated using the approach introduced in this thesis considers all the possible outcomes of the build system and produces a more global analysis result.

3.3 Chapter Summary

In this chapter, we survey prior research along the build systems and analysis of build code. While related work shows active research on using build dependency graphs for software analytics tasks, like impact analysis, the graph can be significantly enhanced by (1) statically analyzing the build code and considering the build-time configuration settings and (2) extracting and analyzing code-adjacent artifacts, like 3D objects in a game project or machine learning models in an AI-based project.

Broadly speaking, we describe our empirical studies that set out to enhance the build dependency graph in the remainder of this thesis. We begin, in the next chapter, by introducing an approach that extracts configuration settings from build code and assesses its impact on the efficiency and effectiveness of developers tasked with change impact and risk awareness exercises.

Chapter 4

Assessing the Exposure of Software Changes, The DiPiDi Approach

The research protocol used in this chapter has been accepted by the Registered Reports track of MSR 2021. Additionally, an earlier version of the work in this chapter has been accepted to the Springer Journal of Empirical Software Engineering (EMSE).

4.1 Introduction

Software programs with various compile-time configurations have multiple dependency paths to their source files from their *deliverables*, i.e., software artifacts that users can interact with, such as executable files or libraries. Build systems derive default configuration settings by analyzing the execution environment or reading user override settings. Build systems use these settings to reason about whether source files (or conditionally compiled code snippets) should be included or excluded from the produced deliverables. Under some conditions, a source file may play a role in one compiled deliverable without affecting others. For example, in the Linux kernel, the source files written specifically for the ARM architecture will be excluded from the x86 version of the kernel [59]. In these complex systems, a change in a source file may have unexpected side effects on deliverables outside of the current compilation path. Software systems that support multiple variants can

therefore create complex arrangements of effects and side effects, where the deliverables exposed to a code-change can be unclear [15].

Software engineering practices that assess source code changes, like code review, are expensive and time-consuming [16, 20]. Extra time and effort must be spent by developers on activities like finding which deliverables are exposed to a change. In this chapter, we define the exposure of a change as the set of deliverables affected by a change, including executables and libraries, as well as the different build-time configuration and environment settings under which the changes propagate. Changes that impact critical deliverables or configurations may require more quality assurance effort than others to mitigate their exposure risk [81].

When modifying complex software programs, source code changes may be localized or broad. Figure 4.1 shows an example of a dependency graph for the ET: Legacy project.[1] A change to the `dl_main_curl.c` file impacts the deliverable `etl` only if the `FEATURE_CURL` option is `ON`. On the other hand, changes to files represented by `$CLIENT_SRC` will always impact the deliverable. A change that only impacts one variant of a system may not be as important as a change that affects all variants. Exposing the effect of a change under different configuration settings can help developers assess the impact of that change.

Therefore, we propose DiPiDi, an approach to assess the exposure of changes to the source code of systems using the build system specification files. One of the key roles of the build system is finding and selecting files based on build scripts, build-time configurations, and environmental variables [9, 73, 86]. By statically analyzing the build scripts and constructing the *Build Dependency Graph* (BDG), we can assess the exposure of a change on all software variants.

To evaluate the proposed approach, we conduct an experiment to assess the effect of DiPiDi on the effectiveness and efficiency of determining the exposure of source code changes on projects that are using CMake build system.[2] To that end, we form three participant groups – those with no tool assistance, those with the assistance of a CIA tool, and those with the assistance of DiPiDi – and compare their efficiency and effectiveness on prescribed tasks. The participants are asked to identify the impacted deliverables and variants for given source code changes while we monitor their performance. A tool that could significantly improve effectiveness and efficiency for these tasks could be useful in many applications both for researchers who design experiments based on source code change (e.g., mutation testing) [70] and practitioners in the allocation of quality assurance

[1] https://github.com/etlegacy/etlegacy

[2] This study has been reviewed and received ethics clearance through the University of Waterloo Research Ethics Committee (ORE# 43727)

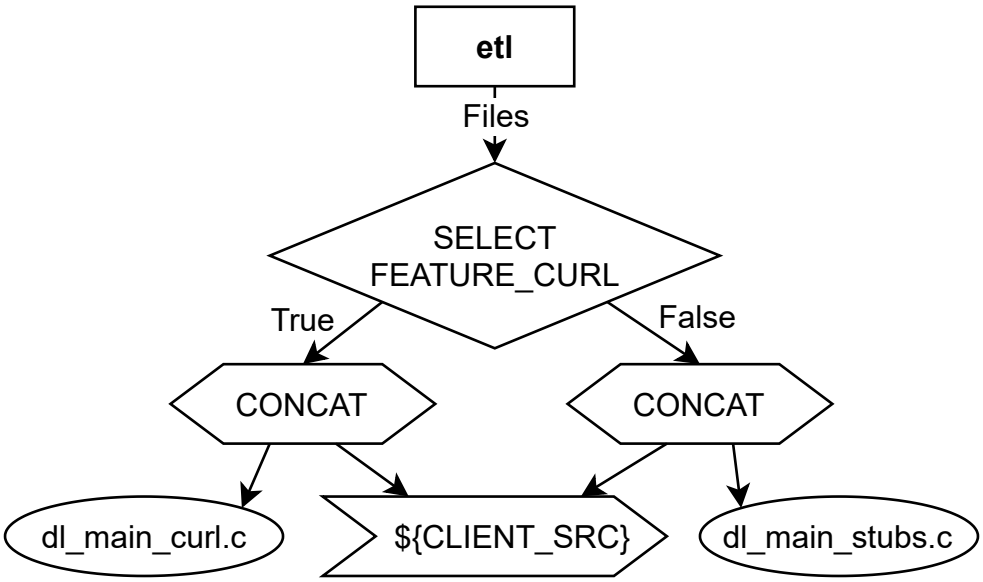


Figure 4.1: A build dependency graph generated by DiPiDi for the code showed at Figure 2.1. Arrows show dependency relation between a source node to the destination.

resources.

Result: Our results indicate that without tool support, identifying impacted deliverables is a difficult task, even for experienced developers. Members of the No Tool group obtained the lowest F_1-score in Task Type A and the highest rank distance in Task Type B despite having more experienced developers and professional CMake users than other groups. Moreover, our results suggest that DiPiDi helps developers to identify impacted deliverables more effectively than current solutions. Indeed, the identified impacted deliverables by the members of the DiPiDi group are 32, 40, 36 average percentage points better in terms of precision, recall, and F_1-score over the members of the Existing Tool group. Moreover, we find that developers using our approach identify impacted targets more efficiently than others. DiPiDi results in 42% average task time reduction when compared to the approach used in the positive control group.

The remainder of this chapter is organized as follows. We first describe our research questions in Section 4.2. In Section 4.3, we present and describe our approach called DiPiDi and its prototype implementation. In Section 4.4, we describe the design of the experiment that we use to evaluate DiPiDi. In Section 4.5, we present the results of our experiments. Section 4.6 discloses the threats to the validity of our approach and experiments, and

finally, Section 4.7 concludes the chapter.

4.2 Research Questions

In this study, we aim to determine whether a static analysis of build systems can improve the effectiveness and efficiency of software developers striving to assess the exposure of a source code change.

Despite the importance of understanding exposure, we conjecture that it is difficult to assess without tool support. To this end, we propose DiPiDi to improve awareness of the exposure of changes. We hypothesize that DiPiDi will allow developers to more efficiently and effectively determine the exposure of source code changes.

A source code change, or patch, that impacts an application under a specific and rare configuration would likely not merit as much developer attention as a source code change that always impacts the application. A change that impacts more deliverables and/or configurations (high-exposure) has a broader "surface area" and a greater potential to impact users, should a defect be introduced, than a change with low-exposure. Therefore, we believe that knowing which deliverables are affected by a source code change or a patch can allow developers to make more informed decisions when making source code changes. To investigate whether DiPiDi approach help developers to identify the impacted deliverables, we formulate the following research question:

RQ1: Does DiPiDi help developers assess the exposure of source code changes more effectively?

While finding all of the deliverables impacted by a change is important, it also is time-consuming because it requires project-wide knowledge, an understanding of the relations between the files and the build system. Developers attempting this task must identify the modified source code throughout the project and trace them through the build dependency graph, while taking care to consider build-time configuration settings. Some of these configurations may be related to the environment of the user, like the operating system. So, a change may have a side-effect on one machine without appearing on others. On the other hand, build-scripts may use wildcard addressing, like **.cpp*, for the source files, making it challenging to follow a complete compilation path from a deliverable to the changed source file. Therefore, developers may rely on heuristics (e.g., directory structure), or worse, ignore this important step in assessing the risk of a change. We pose the following research question to explore the efficiency of developers while using DiPiDi:

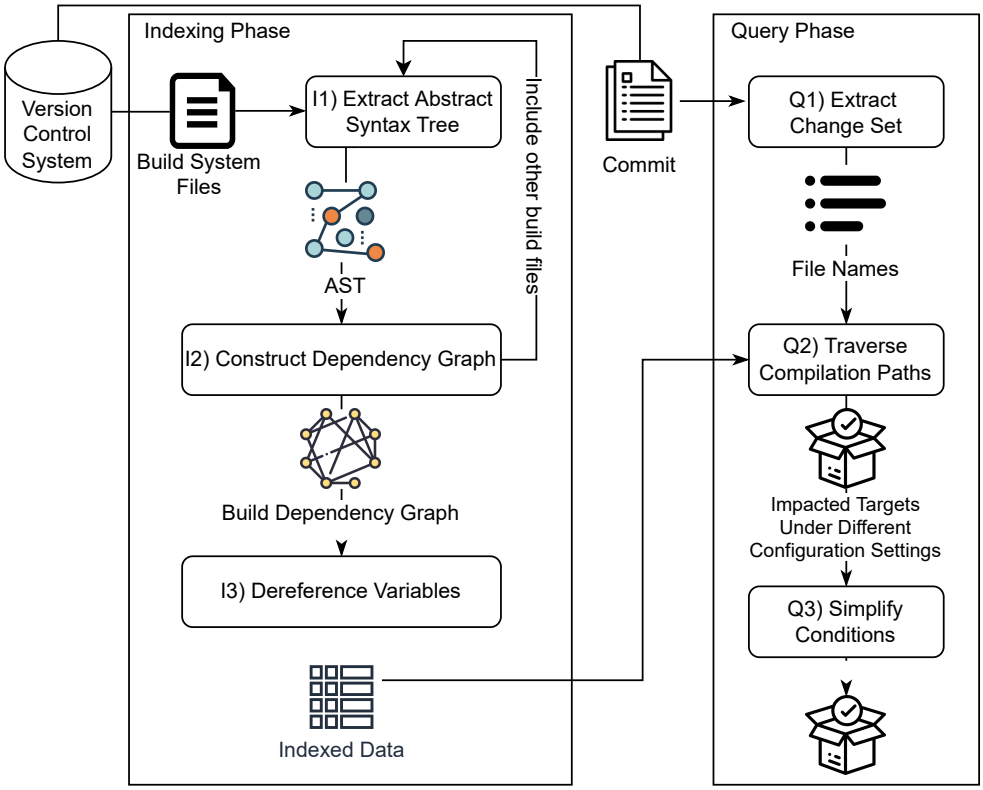


Figure 4.2: An overview of the DiPiDi approach

RQ2: Does DiPiDi help developers assess the exposure of source code changes more efficiently?

4.3 DiPiDi

An overview of the DiPiDi approach can be found in Figure 4.2. The approach has two main phases, the *Indexing Phase* and the *Query Phase*. The purpose of the *Indexing Phase* is to construct an internal representation of the build system. This internal representation includes all the possible compilation paths from each deliverable to the source files. This data can be stored and used later in the *Query Phase*. The purpose of the *Query Phase* is to allow DiPiDi to leverage the data constructed by the *Index Phase* and to output the impacted deliverables under different configuration settings given a set of changed file.

Implementation: In order to conduct our study, we produce a prototype implementation of DiPiDi for the CMake build system. CMake is a cross-platform build system that builds deliverables from artifacts, like source files [44]. CMake has two distinct phases. First, it generates platform-based low-level build specifications (e.g., Makefiles, Visual Studio #.sln files, or Ninja files) [53]. Then, CMake invokes the low-level build tool like `make` to build the project. Our implementation is available online on our public GitHub repository.[3]

We explain each step of the approach presented in Figure 4.2 in more detail below.

4.3.1 Indexing Phase

We explain our approach for each step in the *Indexing Phase* in more detail below. Some steps may require an implementation tailored to the build system being used. In those cases, we also explain our implementation for the prototype of DiPiDi.

I1) Extract Abstract Syntax Tree

Every build system has its own entry file to start building the project. For example, GNU Make looks for a file named `Makefile` in the root of the project. The entry file describes how to build the project using the build system specific language. The project can contain helper build files in other folders or split the entry file and relocate it into multiple folders. All of those files should be addressed and included in the entry point file. To capture the content of the build files, we parse the entry file and build an Abstract Syntax Tree (AST) using a parser that understands the build system grammar. The output of this stage is an AST for one build system related file.

Implementation: Projects using CMake should contain `CMakeLists.txt` in their root directory as the entry file for CMake. Other helper files which have `.cmake` extensions can be in other folders. The tool first parses the CMake specifications starting with the `CMakeLists.txt` file in the project root directory. We use ANTLR [63] to parse and build the *Abstract Syntax Tree (AST)* from the CMake file. The grammar for CMake is straightforward since CMake commands follow the same structure which can be captured by the following parser rule:

```
command_invocation
  : Identifier '(' (single_argument|compound_argument)* ')'
  ;
```

[3] https://github.com/software-rebels/cmake-inspector

I2) Construct Dependency Graph

Next, we traverse the AST to construct the *Build Dependency Graph*, which represents the relationship between the deliverables, source files, and the conditions in each compilation path from deliverables to source files. Table 4.1 shows the different node types used in DiPiDi to construct the Build Dependency Graph from the AST. In this step, DiPiDi also creates a lookup table for each of the variables and targets found while traversing the AST. Some build systems like CMake support scoping for the variables, while others like GNU Make do not. To enable scoping, the lookup table dynamically changes as we parse other files or functions.

As we reach each AST node, based on the name of the command, we select a corresponding node from Table 4.1 and use the lookup table to find the variables and other nodes that this node may depend on. In this step, we cannot assign values to the variables since they might have different values based on the paths we took to reach to them. As an example, consider a variable called `srcs` holding a list of source files. Based on the operating system, the build system may append some additional files, like `foo_arm.cc`, to that variable. Thus, we only keep the nodes and their dependencies. At this level, we may need to include and parse other build-related files found while traversing the AST by repeating the previous step.

At the end of this step, DiPiDi has a graph and a lookup table representing the whole project under analysis, variables, source files, conditions, and targets.

I3) Dereference Variables

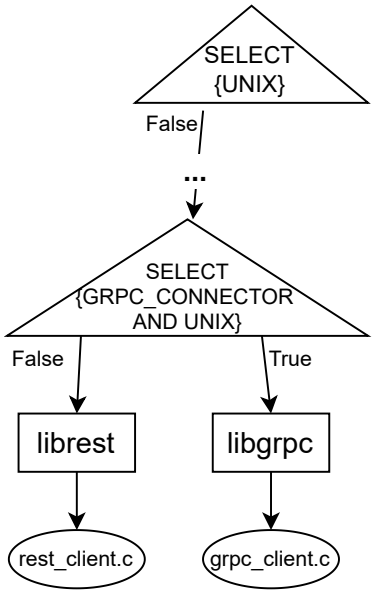
Often in large software applications, there are build-time configuration and environmental settings that help the build system to reason about different variants of the system [40, 50]. These settings create different dependency paths from the deliverable to the source files. In the generated *Build Dependency Graph*, the target nodes which represent the deliverables reside at the top and the leaves are source files represented by LiteralNodes. Using this graph and starting from a target node, we traverse the graph down to the leaves and resolve variables to their values under different build-time configuration settings (i.e., flatten the variables).

By flattening the variables, we obtain all of the possible values for each variable for all configuration settings. This information is then saved and can be accessed through an API when attempting to determine the exposure of a source code change.

[4] https://gitee.com/openeuler/iSulad

Table 4.1: Type of nodes in Build Dependency Graph generated by DiPiDi after traversing the AST

Type	Description	Example Command
TargetNode	Represents a target or deliverables in the project. This node may depend on other nodes to show dependency between a deliverable on libraries, variables, or a list of source files.	add_executable
RefNode	Shows explicitly defined or environmental variables. This node often depends on another node such as a ConcatNode to represent a list or a LiteralNode to show the value of the variable	set
OptionNode	Shows the user-defined build-time configurations in the project.	option
LiteralNode	Represents literal strings or numbers. RefNodes or TargetNodes may point to these nodes to show the value of a variable or source files for a target.	"foo.cc"
SelectNode	Shows conditional paths which have three properties: a condition, a `True` path, and a `False` path.	if
ConcatNode	Represents multiple possible values for a node which should be concatenated together and it points to two or more other nodes	list
CustomCommandNode	All other commands in CMake are represented by this node which can point to an arbitrary number of nodes showing different arguments for a command	find



(a) Part of a dependency graph from the iSulad project.[4]

```
def flatten(node, cond):
  if instance(node, SelectNode):
    if satisfiable(cond + node.cond):
      result += flatten(node.trueNode, cond + node.
   cond)
    if satisfiable(cond + Not(node.cond)):
      result += flatten(node.falseNode, cond + Not(
   node.cond))
  elif instance(node, ConcatNode):
    result += flatten(child) for child in node.
   getChildren()
  elif instance(node, LiterlNode):
    result += node.getValue(), cond
  elif ...
    pass
  return result
```

(b) Flattening algorithm for the SelectNode

Figure 4.3: When flattening the second SelectNode in (a), the approach should remember the `UNIX=False` assumption from the first SelectNode, prune the `True` path and only consider the `False` path.

```
{
"dl_main_curl.c": {
        "FEATURE_CURL": ["etl"]
    },
"dl_main_stubs.c":{
        "NOT FEATURE_CURL": ["etl"]
    },
"common.c": {
        "": ["etl"]
    }
}
```

Figure 4.4: An example of the output of the tool based on the given graph in Figure 4.1. Each key in the root dictionary is a source file in the project. For each source file, another dictionary with conditions as keys and targets as values represents the impacted target given a change which includes the source file.

To evaluate the expressions and conditions while flattening the variables, we used Z3 [57], a library that determines whether a formula is satisfiable, developed by Microsoft Research. Z3 supports formulas involving Boolean, numbers, and strings. We keep track of the evaluation of each condition along the path to prune the build dependency graph while reaching each SelectNode. Figure 4.3(a) shows two SelectNodes in one compilation path. The algorithm does not have any assumption about the variables when it reaches the first SelectNode, which has a condition on the `UNIX` variable. Thus, it expands both paths and calls the algorithm to flatten each path with different assumptions, `UNIX=True` for the True path and `UNIX=False` for the False path. Given the assumption for the False path, when the algorithm reaches the second SelectNode, which has a condition on `UNIX And GRPC_CONNECTOR`, it does not expand the True path because it is not satisfiable, as `UNIX` is False. The output of this phase is all the compilation paths from each target down to the source files with the conditions that are being held as True during the path. An example of the output is shown in Figure 4.4.

4.3.2 Query Phase

This phase uses the index data generated from the previous phase to find the impacted deliverables. The steps of this phase are described below:

Q1) Extract Change Set

A commit in the version control systems contains a list of changed files. Since DiPiDi operates at the file level, which is the same granularity as the build system, we need the changed file names to start the impact analysis. The output of this step is a list of changed source files in a commit.

Q2) Traverse Compilation Paths

Given a list of file names and the output of the *Index Phase*, we can search all of the compilation paths that include the changed file and create a list of exposed targets. In this step, the user can optionally add some assumptions on the configuration settings. Since we store the required conditions for each path, we can use the Z3 library to filter out the paths which are not reachable given the conditions set by the user. The output of this step is a list of exposed targets and the conditions under which each target will be impacted by the change.

Q3) Simplify Conditions

Since we add each condition to our assumptions while traversing a compilation path, the list of conditions generated by the previous step may contain duplicates and can be simplified. In Z3, we can pass functions to the reasoning engine for custom processing steps. These functions are also known as tactics.[4] We use the following tactics to simplify the assumptions:

- `propagate-values`: This tactic propagates the value of each variable between assumptions. For example, if we have an assumption that `a = 0` and `b > a`, we can simplify the second assumption to `b > 0`.
- `propagate-ineqs`: We then propagate the inequalities and remove the subsumed ones. For example, if we have `a > 10` and `a > 2` in our assumptions, we can remove the second one since it is always True if `a` is greater than 10.
- `ctx-solver-simplify`: Finally, we remove the assumptions that are always True. As an example, if we have `a, b, a AND b` in our assumptions, we can remove the third one as it is True.

[4] https://www.philipzucker.com/z3-rise4fun/strategies.html

The output of this step is the same as the previous one with some simplification on the condition list.

4.4 Research Protocol

To test our hypotheses, we conduct randomized controlled experiments with the three groups defined. Study participants are asked to perform a set of prescribed tasks with their usual development setup without additional help (control group), with a baseline change impact analysis tool (positive control group), and with DiPiDi (treatment group). We measure the effectiveness of our tool by comparing the responses of the participants with an established ground truth. We measure the efficiency of our participants by comparing the duration of each task across the groups.

4.4.1 Variables

This section presents an overview of the study variables, which are further described below.

Independent Variable

In our study design, the tool support provided to the participants varies (*No Tool*, *With Existing Tool*, and *With DiPiDi*) which is represented by the *Tooling Level* independent variable. All tooling levels have access to the same information and interface. The only difference in access is the additional output of the Existing Tool/DiPiDi for the relevant groups. More specifically, each group is defined as follows:

1. **No Tool**. This group has access to the code change and other files in the project, including the build specifications. They can use their preferred development environment to perform the tasks. This group is a control group and represents the current practices used by software developers attempting to determine which deliverables are affected by a source code change.

2. **Existing Tool**. This group has access to the same environment as the *No Tool* group, as well as the output of the change impact analysis generated using the Understand Tool [72]. This group is a positive control group and represents the current approaches used by software engineering research to aid software developers attempting to determine which deliverables are affected by a source code change.

3. **DiPiDi**: This group – the treatment group – has access to the same environment as the *No Tool* group, as well as DiPiDi. The participant can interact with the tool using the Query Interface as described in Section 4.3. Our tool can print the impacted deliverables at the file level. Although the file granularity may overestimate the true impact of a change, it is the granularity at which the build system operates.

Participants in all groups may use any external tool that they feel may be helpful. Thus, even the results from the No Tool group can be viewed as a baseline set of current approaches used by the developers. We collect the names of the tools that our participants used and report them in Section 4.5.

For this user experiment, we use a Between-subjects study design. This means that each participant is only exposed to a single tooling group. Participants are asked to complete five highly cognitive tasks, which take time to finish. Additionally, the DiPiDi and the Existing Tool group participants must learn to use the tool. Designing a within-subjects study would have required training and preparation for each setting for each participant, making the study longer and, likely, more difficult to complete. For example, if a participant who only recently figured out how to work with DiPiDi is then asked to complete the next task using a different (baseline) tool, the participant should go through additional steps to learn and set up this tool, adding an extra burden to the study. While a within-subjects study requires fewer participants, we did not have such a concern when designing the study and submitting the protocol for review. At the time, we were planning to recruit at least 66 people, which was suitable for a between-subjects study. However, due to unforeseeable circumstances, we did not attract all of our anticipated subjects in the end. To keep the methodology as close to the original protocol as possible, we retained our Between-subjects study design and attempted to recruit as many participants as possible.

Dependent Variables

Our dependent variables are outlined in Table 4.2. We discuss our reasoning for these variables below.

1. **Exposure analysis effectiveness**: The score from each task indicates how close the answers of the participants are to the ground truth. We could alternatively determine if a participant provides fully correct answers for each task and consider the ratio of correct answers to total tasks. However, we believe that our approach, which indicates how close participants are to fully correct answers, allows us to obtain a finer-grained insight into how participants complete their tasks. Thus, we consider our task scores

Table 4.2: The dependent variables of the study

Name	Description	Scale	Operationalization
Number of correctly identified deliverables	Ratio of the impacted deliverables correctly identified by the participants under a specific build-time configuration over the known impacted deliverables (RQ1)	ratio	Computed at the end using the harmonic mean (F-measure) for task types A & C. See Sections 4.4.3 & 4.4.6
Relative rate of correctly identified deliverables	Normalized pairwise disagreements between participant rankings of patches in terms of the number of impacted deliverables, and known correct rankings (RQ1)	ratio	Calculated at the end for tasks of type B. See Section 4.4.6
Exposure analysis effectiveness	The sum of the number of correctly identified deliverables and relative rate of correctly identified deliverables (RQ1)	ratio	Computed at the end using the number of correctly identified deliverables and the relative rate of correctly identified deliverables.
Task time	The time needed for each participant to complete a task subtracting pauses (RQ2)	ratio	Measured by our web application. The participant can pause a task and resume manually. See Section 4.4.2
Exposure analysis efficiency	Ratio of the total score of the participant over the sum of all Task times (RQ2)	ratio	Total score is the sum of the scores of all of the individual tasks. See Section 4.4.6

(i.e., *Number of correctly identified deliverables* & *Relative rate of correctly identified deliverables*) to be good proxies for exposure analysis effectiveness.

2. **Exposure analysis efficiency**: We define exposure analysis efficiency as the duration from the initiation to completion of each task in seconds. As a result, completing tasks more rapidly will result in higher efficiency. This way, we consider both the fully correct answers and the partial ones, especially in the rank-based tasks (see Section 4.4.3).

Confounding Variables

Table 4.3: The confounding variables of the study

Name	Description	Scale	Operationalization
CMake experience	Participant's experience in working with CMake build system	ordinal	Measured: 3-point scale ("none", "tried", "used in professional development"); questionnaire
Code changes	Changed code in diff format along with the other source files of the project	nomina	Design: each participant gets patches from three real-world projects
Configuration settings	Environmental and build configuration settings of the build system: default configuration, custom	nomina	Design: for applicable tasks, each participant gets two configurations for build settings.
Current programming practice	How often the participant currently programs	ordinal	Measured: 3-point scale ("not", "sometimes", "often"); questionnaire
Development experience	Participant's software development experience in years	ordinal	Measured: 5-point scale ("less than a year" ... "10 years or more"); questionnaire
Fitness	Physical fitness of the participant, like tiredness, during the experiment	ordinal	Measured: 5-point scale ("very tired" ... "very fit"); questionnaire

Perceived task difficulty	Participant's overall perception of the task provided during the experiment	ordinal	Measured: 3-point scale ("easy", "average", "hard"); questionnaire at the end
Project-specific experience	Participant's past experience with the provided project and patch	ordinal	Measured: 3-point scale ("none", "user", "contributor"); questionnaire at the end

Because different code changes might affect the results of our participants, we control the code changes made available to them. The confounding variables that we consider for our study are presented in Table 4.3. We present patches from three different projects to ensure our results are not biased towards any single project. We also control build-time configuration settings to evaluate tooling levels with multiple build configurations without introducing confounding factors. We gather demographic information like the *Development experience* in order to control their correlation with the dependent variables. We also use these variables to inform our data preprocessing (e.g., provide context to determine why a participant might not have finished a task) and for further analysis. We use this data to augment the statistical analysis and make decisions about whether a participant is suitable for a task.

4.4.2 Materials

In this section, we describe the materials that we use in this study.

DiPiDi

We developed a prototype implementation of DiPiDi to reveal the exposure of a change in a structured manner. In a nutshell, our tool processes build specifications statically to produce a *Build Dependency Graph (BDG)*, which we traverse to assess exposure. Before conducting the experiments, we perform the Indexing Phase on the projects that are being presented to our participants and save the output. Participants in the DiPiDi tooling level of the experiment use tool's querying features to perform the assigned tasks.

Existing tool

To assess whether the improvements in the DiPiDi tooling level (treatment) group are related to the approach implemented by our tool, we select a recent and available impact analysis tool to employ in the *Existing tool* (positive control) group.

Unfortunately, most of the proposed impact analysis tools are prototypes [49]. Additionally, due to our project selection and since our implementation of the DiPiDi approach supports CMake build specifications, the impact analysis tool must support the C++ programming language. For example, we had originally selected Frama-C; a tool proposed by [43]. Frama-C is an industrial-grade static analysis tool, which can perform impact analysis on C and C++ projects. However, Frama-C only works on C++ projects with the help of an early access plugin, which has limited support called Frama-Clang, which converts C++ code to plain C code before running other analysis in Frama-C. This plugin is in its early stage of development and has known issues, as mentioned on the official Frama-C website.[5] While we originally believed that this plugin would allow us to complete comparisons with DiPiDi, in our case, Frama-C could not parse or convert any of the projects that we analyzed in the study. This appears to be due to new syntax introduced in C++17 which is currently not supported by the Frama-Clang plugin.

Therefore, as a replacement, we decided to use Understand, a commercial tool developed by [72] and used in previous studies [27, 62]. Understand is a comprehensive static analysis tool with more than 100 features. However, acquiring a license, installing, and applying the Understand to each of the studied projects would be unwieldy for our participants; thus, it is not applicable to use in our study as is. Fortunately, Understand's features are also available through a Python API. Therefore, we develop a presentation layer for Understand's impact analysis API and represent the result in a web application for the participants. This allows our participants to access useful Understand functionality without the burden of installing and applying it. More specifically, for each project:

1. We extract the list of function-level dependencies from Understand.
2. We keep track of where functions are defined in each file as reported by the Understand tool.
3. We persist the result in a structured format (i.e., JSON) that can be consumed by our web application.

[5] https://frama-c.com/fc-plugins/frama-clang.html

Later during the study, the participants can paste a commit ID into the web application to produce Understand-based impact result for the changed program elements. The web application extracts a list of changed functions from the commit and identifies impacted files by traversing the dependencies that Understand computes. Note that the existing tool provided to the participants is simply a presentation layer for the Understand tool—all of the results presented to the participants are therefore calculated by Understand.

The difference between Understand and Frama-C is that Understand operates at the function level, while Frama-C can analyze impacts at the statement level. However, when using Frama-C, the user should install the tool, import the project, and manually select the statements that changed in the commit. By leveraging the API of Understand, however, we make the results of the existing tool accessible through a web interface. The script we use to persist the structured function-level data is available in our repository.[6]

While DiPiDi identifies the impacted deliverables by statically analyzing the build code and considering all build-time configurations, Understand (and other impact analysis tools available today) identify the impacted files by analyzing the source code of the project, without considering build configurations. Then, it is the responsibility of the developer to find the impacted deliverable by matching the file names with the build code.

Studied Projects

Table 4.4: Summary of the selected projects

Name	Line of Codes	Commits
ET: Legacy	3,706,703	11,047
libuv	113,414	4,928
Box2D	128,474	1,282

We select three projects using GitHub search. We first select projects that mentioned CMake in their README file, and then sort by the number of stars for each project. A summary of the selected projects is presented in Table 4.4.

As explained in Section 4.4.3, participants are asked to rank three patches based on their impact on the project, e.g., a patch that impacts three platform-specific versions of the project has higher rank than a patch that impacts one platform-specific version.

[6] https://github.com/software-rebels/cmake-inspector/blob/master/UNDGraph.py

Participants are also asked to identify the configuration settings in which the changes in the patch propagate to the project deliverables. Thus, for each studied project, we iterate over patches in reverse chronological order, selecting patches that impact a different number of deliverables under different configuration settings until three patches have been selected (nine patches in total). To identify the impacted deliverables, we manually inspect the source files and find the deliverables that are impacted by the changed code. We use this as our ground truth. While DiPiDi reports changes at the file-level, in this study, participants are asked to report impacted deliverables at the code level, a subset of reported deliverables by the tool.

Experiment UI

Figure 4.5 shows the web interface for our prototype. As soon as DiPiDi completes the indexing phase, the web interface connects to the tool using a Remote Procedure Call. In the first section, the user can either choose a changed file or select a commit. In the second section, the user can add the build configuration settings, which can be Boolean, string, or arithmetic conditions. Although more complex types of expressions are possible, we leave their evaluation for future work, since simple expressions are already pushing the limits of what our control group can handle. Additionally, we only support the `equal` operator in the web interface; however, DiPiDi supports other operators (e.g., `>, <, etc.`). The web application issues the request to a backend service, which processes the DiPiDi query. The results are then communicated to the frontend, and the impacted deliverables are presented in the third section. On the backend side, DiPiDi first iterates over the indexed data to identify the targets that are impacted by the changed files. Then, DiPiDi applies the specified conditions (if any were provided) using the Z3 library. If the conditions are still specifiable, DiPiDi adds the target to the impacted list and returns the final list to the web application. This application is available in our repository.[7]

We additionally developed an interactive Web based application to allow us to conduct our experiment with a diverse range of participants and allow our participants to rely on their own development environments. The application retains a log of answers and the duration of each task. The experiment UI randomly assigns each participant to a tooling level group and randomly assigns tasks to the participants, all the while logging which project and tasks are assigned to whom. Participant information was only be made available to the researchers after all the results had been scored to reduce experimenter bias [69]. The interactive UI is also available in our repository.[8]

[7] https://github.com/software-rebels/dipidi-experiment-ui
[8] https://github.com/software-rebels/dipidi-participants-ui

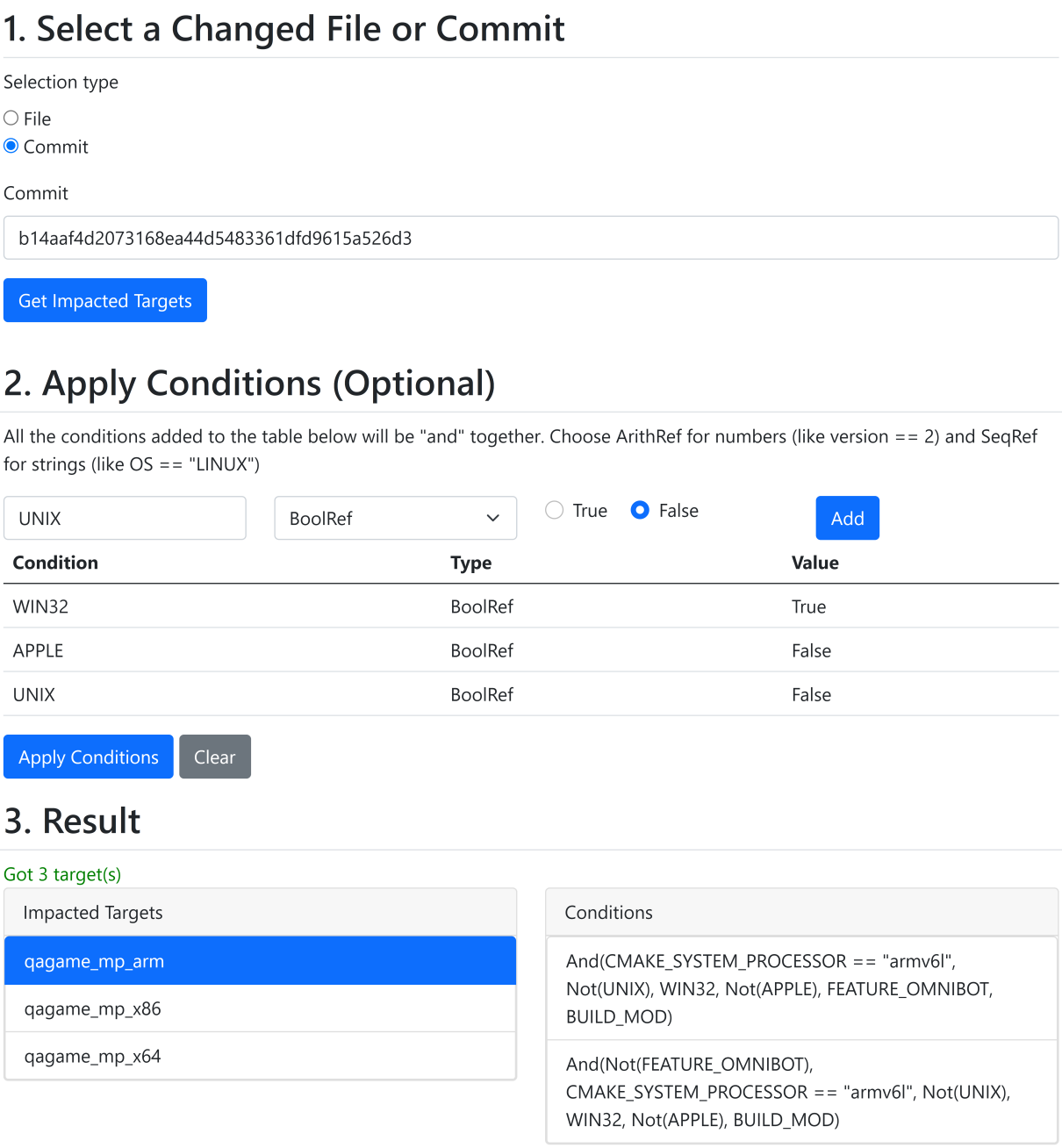


Figure 4.5: DiPiDi Web Query Interface

4.4.3 Tasks

We ask our participants to complete five tasks, one Type A task, two Type B tasks, and two Type C tasks. After a participant initiates our experiment through our experiment UI, they are randomly assigned to a tooling level and the tasks are randomly ordered and logged. The order of the tasks is randomized to account for learning effects that could occur if developers improve by learning from previous tasks. Furthermore, we construct each task using three different open-source projects, and randomly assign each task to each participant. Therefore, participants cannot share answers with each other, and tasks are less biased towards a specific project or task. Participants must obtain the data and files required to complete each task through our experiment UI and must also provide their answers through it.

Our tasks are constructed to answer both RQ1 and RQ2. The results obtained for each task can be used to answer our first research question (i.e., RQ1), while the duration of the tasks can be compared for each group to answer RQ2. The three task types are as follows.

Task Type A: The purpose of this task is to compare the exposure assessment effectiveness and efficiency of the participants in different tooling levels. The participant is provided with the names of changed files and a set of build specifications. The participant is then asked to list impacted deliverables (without having the source code). The experiment UI provides a text input field for the participant to identify those deliverables.

Task Type B: The purpose of these tasks is to determine the effect of presenting exposure reports on the effectiveness and efficiency of developers assessing the relative exposure of patches. The participant is assigned three patches and a set of build specifications. We ask the participant to rank the patches listed in the experiment UI based on (a) the number of impacted deliverables; and (b) the number of impacted application variants (e.g., number of affected OS). We ensure that the patches do not affect the same number of deliverables and application variants. Furthermore, the patches are sampled from a different project than the ones studied for other tasks.

Task Type C: The purpose of these tasks is to determine the impact of DiPiDi when participants are particularly interested in the exposure in a given setting. Participants are presented with three patches and asked to identify those that (a) affect a specified set of deliverables; (b) affect a specific variant of the software; and (c) identify the configuration settings under which the changes will propagate. For this task type, we use a different project than for tasks of types A and B to make sure that all of the participants see examples from each of the three projects that we selected for this study.

4.4.4 Participants

Table 4.5: Demographic information about the participants

Tooling Level	ID	Programming Experience	CMake Familiarity	Current Programming Practice
DiPiDi	P1	five years or more	Tried it at least once	More than once per week
	P2	five years or more	Used in professional development	More than once per week
	P3	five years or more	Tried it at least once	More than once per week
	P4	two to five years	Tried it at least once	More than once per week
	P5	five years or more	None	More than once per week
	P6	five years or more	None	More than once per week
	P7	two to five years	Tried it at least once	Sometimes
	P8	two to five years	Tried it at least once	More than once per week
	P9	five years or more	Tried it at least once	More than once per week
	P10	a year to two years	None	Sometimes
	P11	two to five years	Tried it at least once	Sometimes
Existing Tool	P12	two to five years	Tried it at least once	More than once per week
	P13	five years or more	Tried it at least once	More than once per week
	P14	two to five years	Tried it at least once	Sometimes
	P15	five years or more	Tried it at least once	More than once per week
	P16	two to five years	Tried it at least once	More than once per week
	P17	a year to two years	Tried it at least once	More than once per week
	P18	two to five years	None	More than once per week
No Tool	P19	five years or more	Used in professional development	More than once per week
	P20	two to five years	Used in professional development	More than once per week
	P21	five years or more	Used in professional development	More than once per week
	P22	five years or more	Used in professional development	More than once per week
	P23	five years or more	Tried it at least once	More than once per week

P24	two to five years	None	More than once per week
P25	five years or more	Tried it at least once	More than once per week
P26	two to five years	Tried it at least once	More than once per week
P27	two to five years	Tried it at least once	More than once per week
P28	two to five years	None	More than once per week
P29	five years or more	Tried it at least once	More than once per week
P30	five years or more	Tried it at least once	More than once per week
P31	two to five years	Tried it at least once	More than once per week
P32	five years or more	Used in professional development	More than once per week

Since our tasks are centred around specific software engineering practices, our participants should have the programming experience necessary to allow them to find the deliverables impacted by a source code change. We therefore populate our pool of participants with software developers, or individuals with programming experience.

We solicited participation from CMake user web forums, the developer mailing lists of large projects that are implemented in CMake (e.g., KDE, Qt), at a user summit of a code reviewing platform, via our personal contacts on social media, and a local group of graduate students, all of whom have developed software in a practical setting. A total of 72 participants enrolled in the study. We piloted the experiment UI and tasks with two participants. The pilot results are not included in our reported findings below. The remaining 70 participants were invited to participate in the study. Out of those, 34 participants completed the set of tasks. Of those who finished, two participants skipped at least 3 tasks, so we exclude them from further analysis. In the end, 32 participants remain – eleven in the DiPiDi group, seven in the Existing Tool group, and fourteen in the No Tool group. Table 4.5 shows an overview of the profiles of the participants in this study.

4.4.5 Execution Plan

We provided our participants with access to our web application in batches of three. This staged approach allowed us to fix any potential problems without invalidating too large of a subset of our participant data. Based on the feedback that we received, we clarified the task descriptions with additional detail, but the tasks themselves remained the same. We enhanced the experiment UI to indicate when the backend is processing the issued query, as processing queries took on the order of five to ten seconds, and users would mistakenly submit multiple requests. The application has the following procedure for each participant:

Welcome Page

Participants are first presented with an outline of the tasks and an estimate of the time required to complete the tasks. In addition, we request the consent of participants to participate in the experiment. The participants are asked to refrain from sharing task information with other participants. For ethical compliance reasons, participants are also informed that they may stop the experiment at any time for any reason.

Onboarding

After obtaining consent from the participants, we provide a more detailed explanation of the specific set of tasks to be completed during the experiment. Based on the tooling level assigned to the participant, we explain the steps required to prepare the environment and the tool (if applicable). We inform participants that they may use their preferred development tools (e.g., CLI tools, IDE). Participants are also informed that each task is timed, that their responses will remain anonymous unless they explicitly request otherwise, and that they may skip individual tasks.

Tasks

We present our participants with the tasks outlined in Section 4.4.3 in a random order. For each task, our application provides a hyperlink to download the source code. A timer begins as soon as the task page is loaded. We also record when checkpoints are reached during the experiment. Before showing the description of the task, we provide the download link and the necessary steps to prepare the environment. The participants must click on the "ready" button to initiate the experiment. We also log the moments that the participants begin to enter their responses. The page describes the task and shows the configuration settings that the participant should consider. We present the results of the tools in the experiment UI for participants in the 'Existing Tool' and 'DiPiDi' tooling levels in an interactive way through a Web interface. The application provides input spaces for the participant to enter their responses. The application logs the time that the participant spent on each task. The participant may click a pause button to pause the timer if a distraction of any kind interrupts their focus. A skip button allows the participant to move on if they feel that they cannot complete a task. A sample of each task is provided in appendices B.1 to B.6.

Questionnaire

Prior to the start of the experiment, the participants are asked demographic questions about their background and programming experience. The questionnaire is included in Appendix A. After a participant completes their five tasks, we follow up with a questionnaire which is included in Appendix B. The purpose of the post-study questionnaire is to collect tool usage questions about the CLI tools, IDEs, and/or other tools that were used to complete the tasks. Additionally, we ask whether the participants found the provided tool useful. We also ask participants to comment on any problems that they may have encountered during the experiment. Finally, we thank the participants and invite them to provide other feedback if they desire. The results of the post-questionnaire are presented in Table 4.9.

4.4.6 Analysis Plan

In this section, we describe the analysis plan we use in this study.

Data Cleaning

We assign each participant five tasks to complete. However, it is possible for a participant to exit the application before completing all of their assigned tasks. Since the experiment UI accepts input from participants in any text format, we manually check that answers are acceptable before analyzing them. Next, we review the participant's questionnaire submission and feedback for mentions of problems that may (partially) invalidate their submission, removing their invalid answers when appropriate. Additionally, we use outlier detection approaches, i.e., Tukey's fences [79] and box plots, which do not require regression models. If there are outliers, we analyze them by hand to gain insight into them. Finally, we remove those data if we find enough evidence to do so after both outlier detection and manual evaluation.

Measuring Effectiveness

For rank-based tasks, i.e., *task type B*, we use Kendall's tau ranking distance formula [42] to compute the distance between participant answers and the ground truth. Kendall's tau ranking is defined as:

$$K_d(\tau_1, \tau_2) = \sum_{\{i,j\} \in P, i<j} \bar{K}_{i,j}(\tau_1, \tau_2)$$

where P is the pairwise set of elements in τ_1 and τ_2, $\bar{K}_{i,j}(\tau_1, \tau_2)$ is 0 if i and j are in the same order in τ_1 and τ_2 otherwise it is 1. For example, the Kendall's tau distance between `2,1,3` and `1,2,3` is one because pair $\{2, 1\}$ are in different order. We report the distance as a number between zero and three for those tasks.

For list-based tasks, i.e., *task types A and C*, like previous studies, we compute precision and recall. As discussed, the goal of this study is to expose the change under different configuration settings and help developers to identify impacted deliverables for a specific configuration setting. To compute the correctness and completeness of the participant's Estimated Impacted Deliverables (EID), we compare them to Actual Impacted Deliverables (AID) using the following precision (correctness) and recall (completeness) formulas:

$$Precision = \frac{EID \cap AID}{EID}; Recall = \frac{EID \cap AID}{AID}$$

Due to the natural trade-off between precision and recall, we calculate the F_1-score (i.e., the harmonic mean of the precision and recall) to get an overall impression of task effectiveness.

4.4.7 Deviations From the Registered Report

The study design and analysis protocol used in this study has been reviewed and accepted at the MSR 2021 registered report track [56]. To complete the study, we had to deviate from our original registered report protocol during the course of this thesis. In this section, we summarise the deviations from the original protocol.

Replacing the Frama-C Tool

While the Frama-C tool was our original choice to compare DiPiDi to an existing tool, we could not make use of it as discussed in Section 4.4.2. We decided to make use of another existing tool capable of analysing code impacted by a code change. The Understand tool has features that allow developers to trace a change and find the parts of a program that it impacts. However, installing the tool, and learning to use it, was not feasible for the participants given the constraints of the study (time and computing environment). Thus, we developed a UI tool that consumes the output of the Understand API, and

represents the result in a web application for the participants. Thus, the tool that we develop is a presentation layer for the Understand results. We therefore switched Frama-C for Understand.

Unfortunately, our initial tool selection could not analyze the studied projects; however, we believe that switching from Frama-C to Understand will not substantially impact the performance of the positive control group because (1) both tools are commercial grade and (2) both tools can perform similar styles of change impact analysis via source code analysis. While our original choice may have been easier to use for our participants, we believe that our presentation layer wrapper bridges that gap.

Change of Studied Projects

Originally, we wanted to conduct the study on projects from the KDE and QT communities. However, we found that projects in those communities use customized CMake commands to maximize reuse and productivity among the projects.[9]

Developing support for this set of commands required additional engineering effort for DiPiDi. Unfortunately, we did not have sufficient time to invest the engineering time to implement these supports for custom commands. Therefore, we systematically selected alternative projects that use the 'vanilla' version of CMake specifications. To identify candidate projects, we sorted repositories that are hosted on GitHub and use CMake, by the number of stars, which we believe is a good proxy of the popularity of a project. We believe that improvements that can work on popular projects are more likely to benefit a larger number of developers. From that list of projects, we selected three projects of varying size and domain for our experiment. Table 4.4 provides an overview of the studied projects.

Number of Participants

In our registered report, we set out to conduct our study with 66 participants to be able to compare the groups with large effect sizes using one-way ANOVA. Since participants are required to be developers who are familiar with build systems, we faced difficulties recruiting such a large number of developers for this study. We recruited participants using a variety of communication channels, such as social media (Twitter, LinkedIn, Reddit), mailing lists of open-source projects, developer forums, and developer conferences. After

[9] https://linux.die.net/man/1/kdecmake

leveraging those channels, we ended up with 72 candidates who signed up to participate in the study. Of those, 32 completed at least 4 of the 5 tasks, 11 in the DiPiDi group, 7 in the Existing Tool group, and 14 in the No Tool group.

Due to the limited number of participants, we could not conduct our planned ANOVA analysis. Therefore, we follow our contingency plan and conduct a preliminary analysis of our results instead. The details of our analysis can be found in Section 4.5.

4.5 Results

In this section, we present the results of our experiment with respect to our two research questions.

4.5.1 RQ1: Does DiPiDi help developers assess the exposure of source code changes more effectively?

The participants in the DiPiDi tooling level outperformed the other two groups in terms of their accuracy in identifying the impacted deliverables and assessing the magnitude of the impact. As shown in Table 4.7, the DiPiDi group outperforms the Existing Tool group by 42 and 31 percentage points in terms of F_1-score for Task Type A and Task Type C, respectively. Moreover, the DiPiDi group outperforms the Existing Tool group by 0.62 units of distance in the impact ranking task (Task Type B).

As described in Section 4.4.3, we assign one Task Type A out of three, two Task Type B out of six and two Task Type C out of nine to each participant. To calculate the metrics shown in Table 4.7, we compute the average (mean) performance measure across participants in each group to aggregate the measures to the granularity of group comparison. For Task Type B, the distance represents the number of pairwise ranking swaps required to change the order of the participant's answer to match the ground truth. Since we asked the participants to order exactly three commits in Task Type B, the upper bound for this number is three, meaning the order is reversed.

The effectiveness of DiPiDi in Task Type A is also illustrated in Figure 4.6, which shows the distribution of the Precision, Recall, and F_1-score for Task Type A and each tooling level. In the DiPiDi group, 10 out of 11 participants perform better than the Existing Tool and No Tool groups, achieving an F_1-score of 1 as shown in Figure 4.6c. However, it also shows a tail extending to 0.33 (P1) in the DiPiDi group. We reached out to P1 to

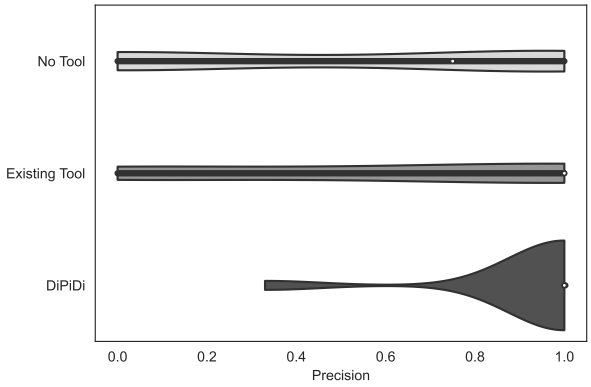
understand if there was any problem with the tasks. P1's experience and familiarity with CMake were limited to a classroom setting. P1 reported that it was difficult to understand the tasks, but despite P1's lack of experience, DiPiDi did help P1 to complete the tasks to a certain degree.

For Task Type B, Figure 4.7 shows that DiPiDi is effective in identifying the most impactful commits. We believe that accuracy in assessing the riskiness of changes relative to each other can help reviewers and quality assurance teams to manage their resources. Current impact analysis approaches, including the one used for the Existing Tool tooling level, do not consider the build-time configuration settings and, therefore, report the impacted file or statements for a single set of configuration settings (often, the default settings).

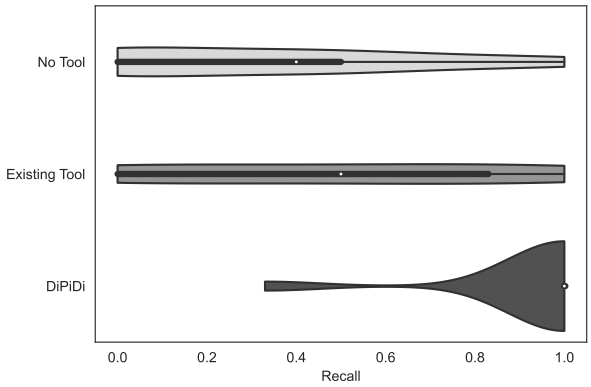
Finally, in Task Type C, Figure 4.8 shows that participants in the DiPiDi tooling level outperform others. The F_1-score for 10 out of 11 participants is greater than 0.9 in the DiPiDi group. Surprisingly, the No Tool group outperforms the Existing Tool group. Since in Task Type C, participants are asked to identify the patches that impact the deliverables under a specific set of configuration settings, a tool that does not consider all the build-time configurations, like the existing tool, may have misled the participants.

Table 4.6 shows the participants' expertise in each group. We consider participants to be experienced developers if they have more than five years of programming experience. Additionally, we identify the participants who use CMake in a professional setting. Participants can be neither experienced developers nor professional CMake users if they have two to five years of programming experience and tried CMake at least once. As shown in Tables 4.6 and 4.7, assessing the exposure without any tooling support is difficult, even for those participants with extensive professional experience and those who use CMake in a professional setting. Table 4.9 shows an overview of the post-study questionnaire results. In general, participants in the DiPiDi group find the tool useful and find the tasks less difficult in comparison to other groups. Although we do not draw any firm conclusions about this, the fact that fewer participants find the study difficult suggests that performing with build-related tasks without tool support is daunting.

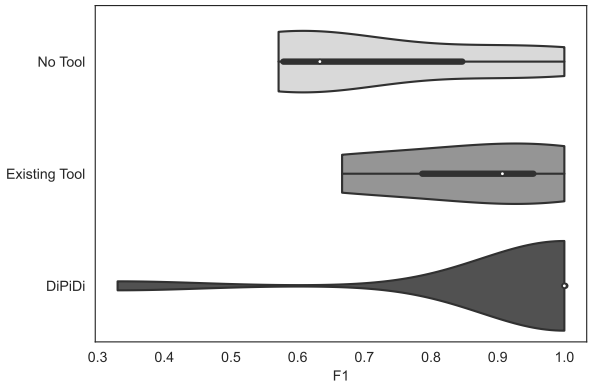
The DiPiDi approach helps practitioners and researchers to identify the impacted deliverables given a change under different build-time configuration settings. In an experimental evaluation, our prototype implementation of DiPiDi outperforms a current impact analysis tool by 36 average percentage points in F_1-score when identifying impacted deliverables. More importantly, participants in the DiPiDi group could assess the riskiness of changes relative to each other with less error.



(a) Participant precision for Task A



(b) Participant recall for Task A



(c) Participant F1 Score for Task A

Figure 4.6: Participants in the DiPiDi group outperform two other groups in all the three metrics. While the Existing Tool group performs better than the No Tool group, the difference is not negligible.

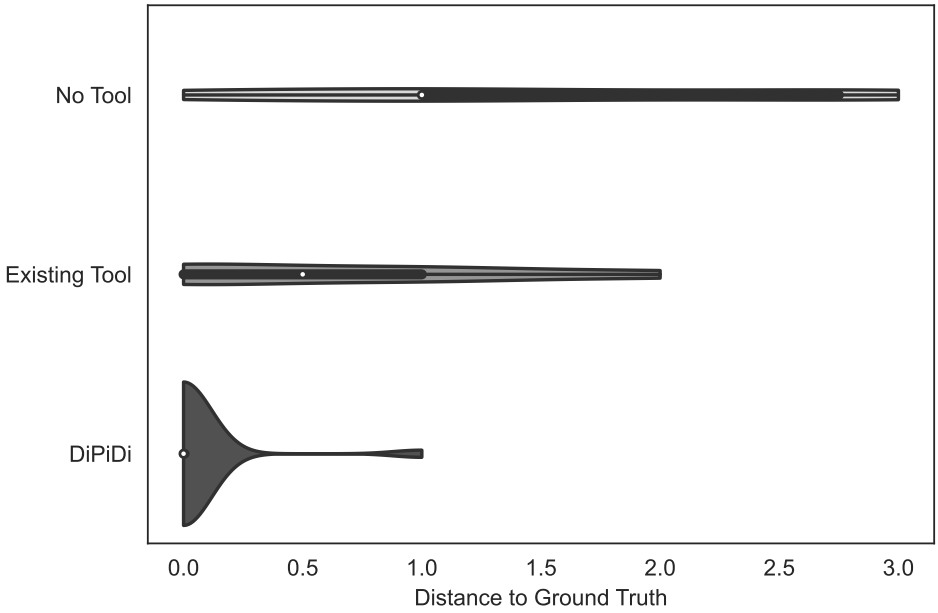
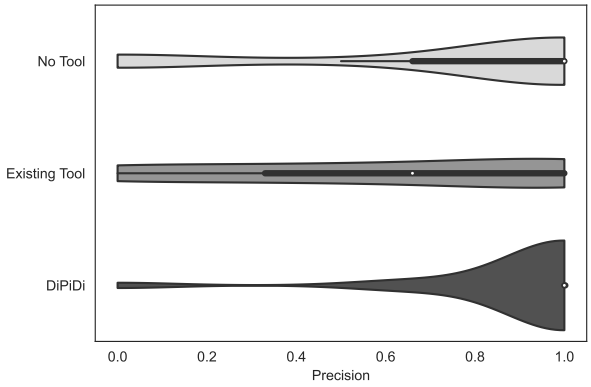


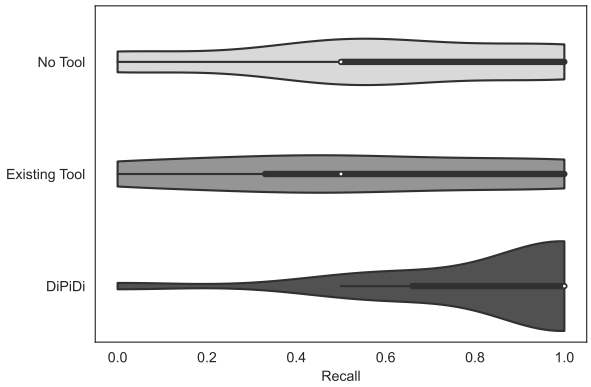
Figure 4.7: Distance between the participant's responses to the ground truth calculated using Kendall tau rank distance formula. The larger the distance, the more dissimilar the responses and the ground truth.

Table 4.6: Participants' expertise based on the demographic questions. Participants can be in more than one experience category.

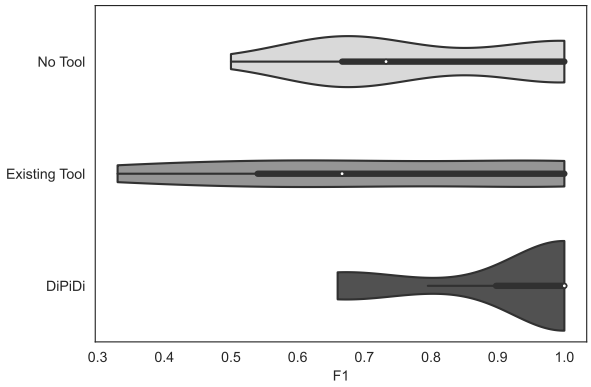
Tooling Level	Total	Experienced Developers	Professional CMake Users
No Tool	14	8	5
Existing Tool	7	2	0
DiPiDi	11	6	1



(a) Participant precision for Task C



(b) Participant recall for Task C



(c) Participant F1 Score for Task C

Figure 4.8: Participants in the DiPiDi group outperform two other groups in all the three metrics in Task C. Interestingly, the No Tool group outperforms the Existing Tool.

Table 4.7: Summary of the result for each task per tooling level

	A			B	C		
Tooling Level	Precision	Recall	F_1	Distance	Precision	Recall	F_1
No Tool	0.52	0.33	0.40	1.5	0.77	0.58	0.66
Existing Tool	0.60	0.47	0.52	0.67	0.61	0.52	0.57
DiPiDi	0.94	0.94	0.94	0.05	0.92	0.85	0.88

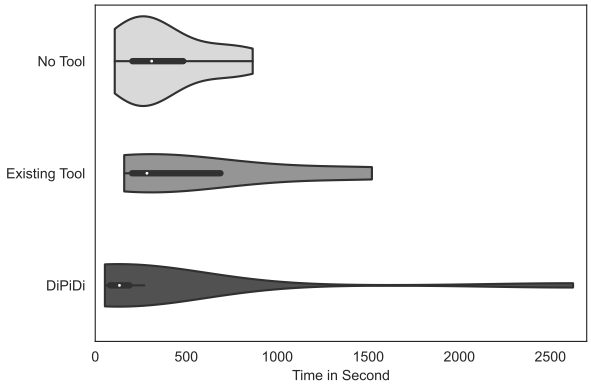
4.5.2 RQ2: Does DiPiDi help developers to assess the exposure of source code changes more efficiently?

The DiPiDi approach helps developers to assess the exposure of changes more efficiently than baseline approaches. Table 4.8 shows that the DiPiDi group spent on average 137, 219, 151 fewer seconds on tasks A, B, C, respectively. Figure 4.9 shows the distribution of duration for each task type in each tooling group. As shown, the majority of the participants in the DiPiDi group complete the tasks more quickly than the participants in the other groups. More specifically, 70%, 81%, and 80% of the participants in other groups performs slower than DiPiDi group in Task Type A, B, and C respectively.

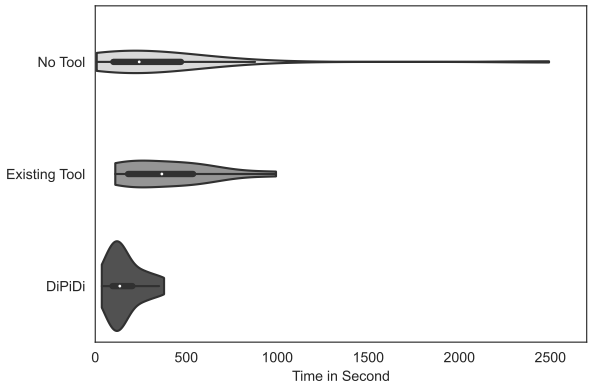
However, P6 took 20 times longer to complete the Task A than other participants in the DiPiDi group. Indeed, P6 completed Task A in 2625 seconds, while the average of the other participants in this group is 127.5 seconds. P6 reported in the feedback form that 'It was hard to understand what to look at in the beginning. My first task is probably affected by that'. P6 performs very well in other tasks and, based on their feedback, believes the tool is very useful. Similarly, P25 in the No Tool group took 3371 seconds to complete Task Type B, which like the above case, was their first task. The average for the No Tool group in Task Type B without P25 is 469 seconds. P25 reported that the experiment was 'Time-consuming and difficult'. Removing those two cases reduce the standard deviation for Task Type A and Task Type B to 67 in DiPiDi group and 256 in No Tool group, respectively and makes all standard deviations less than equal to the average.

Interestingly, except for Task Type B, the No Tool group performs more efficiently than the Existing Tool group. While the Existing Tool group achieves slightly better correctness scores than the No Tool group as shown in Figures 4.6 and 4.7, we suspect that the additional information provided to the participants by the existing tool reduced their efficiency.

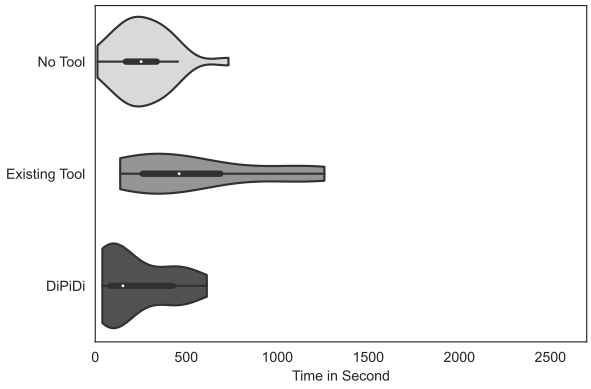
Still, even for the tasks that require more creative ways of interacting with the tool (e.g., Task Type C), considerable efficiency improvements are detected. For Task Type B,



(a) Duration for Task A in Seconds



(b) Duration for Task B in Seconds



(c) Duration for Task C in Seconds

Figure 4.9: Participants in the DiPiDi group finish the tasks faster compared to other groups. While the No Tool group performs more efficiently than the Existing Tool, they are not necessarily more effective.

Table 4.8: Time it takes for each group of participants to do the tasks

	A			B			C		
Tooling Level	Skips	Time	S.D	Skips	Time	S.D	Skips	Time	S.D
No Tool	3	383	248	2	364	787	3	269	261
Existing Tool	2	570	510	2	401	299	1	552	486
DiPiDi	0	354	720	1	163	133	2	229	236

the average time taken for the No Tool and Existing Tool groups is 1.5 standard deviations larger than that of the DiPiDi group. However, for Task Type A and C the difference between the means is less than one standard deviation. We suspect that the differences are not as pronounced because DiPiDi reports many possible compilation paths and participants confirm their answer with the code and the build script in addition to the output of the tool, a time consuming affair.

The DiPiDi approach increases the efficiency of identifying impacted deliverables under different build-time configuration settings. We show that our prototype implementation reduces the time required to assess the exposure of changes by 42% on average. More notably, participants in the DiPiDi group assess the riskiness of changes relative to each other with 92% less error in 59% of the time with 1.7 standard deviations difference over the existing approaches.

4.5.3 Discussion

Table 4.9: Post Questionnaire Result

Tooling Level	**ID**	**Another Tool**	**Tool Usefulness**	**Fitness**	**Difficulty**	**Experience**
DiPiDi	P1		Somehow	Tired	Hard	None
	P2			Energetic	Easy	None
	P3		Very Useful	Neutral	Average	None
	P4	Intellij IDE	Somehow	Neutral	Average	None
	P5	VSCode	Very Useful	Very Energetic	Average	None
	P6	VSCode	Very Useful	Very Tired	Hard	None
	P7		Very Useful	Energetic	Average	User

	P8		Very Useful	Neutral	Average	User
	P9			Neutral	Easy	None
	P10		Very Useful	Neutral	Easy	None
	P11	VS Code	Very Useful	Tired	Average	None
Existing Tool	P12	VSCode	Somehow	Neutral	Average	None
	P13	VSCode	Very Useful	Neutral	Hard	None
	P14	VSCode	Somehow	Very Tired	Hard	None
	P15	Github Online	Very Useful	Energetic	Average	User
	P16			Very Tired	Average	None
	P17	Clion	Not Useful	Very Tired	Hard	None
	P18		Somehow	Tired	Hard	None
No Tool	P19	Sublime		Tired	Average	User
	P20	Bash		Tired	Average	User
	P21			Neutral	Easy	User
	P22	Nvim, ripgrep, fzf		Neutral	Average	User
	P23	Git		Tired	Hard	None
	P24			Tired	Hard	None
	P25			Very Tired	Hard	
	P26			Very Tired	Average	None
	P27			Neutral	Hard	None
	P28			Neutral	Hard	None
	P29			Neutral	Hard	None
	P30			Very Tired	Hard	None
	P31			Very Tired	Hard	None
	P32			Very Tired	Hard	None

In this section, we discuss the results of the study, including the pre- and post- questionnaire data.

Three of the 32 reported that they had difficulty understanding what to do in their first task. However, since we shuffle the order of tasks before assigning them to the participants, the effect of this difficulty is distributed throughout the tasks. For example, Figures 4.9a and 4.9b shows two tails in the DiPiDi and No Tool groups. In both cases, the participant's first task takes longer than the average because they are struggling to understand the tasks.

Some participants with strong programming experience familiar with build systems performed well even without tool support. For example, P32, who worked with C and build systems during their time researching operating systems in grad school, achieved Precision of 1 and Recall of 0.83 in Task Type A without tool support.

Table 4.9 shows the results of our post-experiment questionnaire for each participant. We assign numbers to the fitness level, i.e., 1 = very tired, and 5 = very energetic. The average fitness level for DiPiDi, Existing Tool, and No Tool tooling groups are 3, 2.14, and 2 respectively. This shows that participants assigned to the DiPiDi group felt slightly less fatigued after the study.

Interestingly, some participants who were in the No Tool group and found the experiment difficult, suggested that having a tool that can track the dependencies would be very useful. For example, P30, said 'The tracking of configurations and conditions was almost infeasible. Maybe a visualization tool where user can navigate dependencies and targets can help'. P14 also reported that 'Looking for variables, targets, and file names at the same time was exhausting'. Feedback like this provides more motivation for the need for build-aware tools, such as DiPiDi.

4.6 Threats to Validity

In this section, we discuss the threats to the validity of our study.

4.6.1 Threats to internal validity

Participants may vary in their capacity to estimate exposure. Due to the challenges associated with recruiting a large sample of software developers, participant characteristics that may interact with or confound our dependant variables, (e.g., experience), could not be controlled to a statically significant degree. Nevertheless, we strove to mitigate this threat by randomly assigning tasks to participants and by recruiting participants with varying levels of experience. Additionally, due to the Hawthorne effect, our participants were likely to behave differently in our experimental setting because they were aware that they were being monitored. We attempted to mitigate this threat by giving developers realistic tasks, letting them work on their own computers at a time and place of their choosing. Furthermore, we did not discuss the hypotheses of the study with the participants until after they completed their tasks.

We observed differences in the self reported fitness levels in each tooling groups. There are two potential reasons for these differences. First, the tooling provided by DiPiDi may reduce the cognitive load on the participants in that group. Or second, it is possible that this is simply a random occurrence due to the participants being randomly assigned to a group. We suspect this is the former because we observed a trend in the fitness level based on the tooling group with No Tool presenting the least fit participants.

Finally, in this study, we did not have participants with the necessary project expertise to observe how DiPiDi could affect developers that are intimately familiar with a code base. Nonetheless, we show that the DiPiDi approach is helpful for newcomers, making it suitable for onboarding new people.

4.6.2 Threats to external validity

Although we believe that the DiPiDi approach is general enough to apply to most build systems, our prototype implementation only supports CMake. Therefore, our findings might be limited in scope to the CMake context. On the other hand, CMake shares several concepts with other build systems, especially those based on a platform abstraction layer. For example, GNU Autotools also uses a target-based representation and generates low-level build code (i.e., Makefiles) from higher abstractions and contextual information from the build execution environment. While we believe the results are likely to generalize, replication of the study in the context of other build systems may be fruitful.

4.6.3 Threats to construct validity

Our selected measurements may not fully capture the phenomena that we set out to measure (i.e., effectiveness and efficiency of assessing patch exposure). Nonetheless, we selected a broad range of measurements and tasks that we believe to be meaningfully representative of the underlying phenomena of interest.

In this study, we assume that the build specifications are a complete representation of the build-time variability of the studied systems. However, in reality, build systems may use dynamic features to alter the state of build specifications during build execution. These dynamically generated specifications will not be included in the graph since we generate it statically. To tackle this problem, we choose to prototype DiPiDi on CMake because we believe it is less prone to dynamic build time variability than older build technologies (e.g., Make). In fact, CMake is designed to statically represent this dynamic behaviour using

its built-in abstractions for platform-specific language toolchains. Nonetheless, porting DiPiDi to other technologies may present these kinds of challenges.

4.7 Chapter Summary

To assess the risk of a change, it is important to identify the set of deliverables and configurations that are impacted. To do so, We introduced DiPiDi, an approach that we developed to assess the impact of changes by statically analyzing the build system specification files. To evaluate our approach, we implemented a prototype of our approach and designed an experiment to evaluate whether DiPiDi is associated with improvements to the effectiveness and efficiency of developers performing impact assessment tasks. The result of that experiment suggests that (1) DiPiDi approach helps practitioners and researchers to identify the impacted deliverables given a change under different build-time configuration settings. Our prototype implementation of DiPiDi outperforms current impact analysis tool by 36 average percentage points in F_1-score when identifying impacted deliverables. More importantly, participants in the DiPiDi group could assess the riskiness of changes relative to each other with fewer errors; and (2) the DiPiDi approach increases the efficiency of identifying impacted deliverables under different build-time configuration settings. We show that our prototype implementation reduces the time required to assess the exposure of changes by 42% on average. More notably, participants in the DiPiDi group assess the riskiness of changes relative to each other with 0.05 units of distance in 53% of the time with 1.5 standard deviations difference over the existing approaches.

Chapter 5

Dependency Extraction and Analysis for Multidisciplinary Teams: A Case Study at Ubisoft

An earlier version of the work in this chapter has been submitted to an international conference.

5.1 Introduction

In software development projects that involve personnel from different disciplines, the breadth of software artifacts can be vast [82]. For example, producing high-budget video games ('AAA games') requires the careful coordination of personnel with divergent expertise, such as technical software staff (e.g., developers, QA, and operators), as well as creative staff (e.g., graphic artists, composers and musicians, script writers, and level designers). AAA games are typically composed of millions of lines of code, as well as hundreds of thousands of non-code files (*a.k.a.* code-adjacent artifacts) like textures and animations [61].

Multidisciplinary teams require a multidisciplinary dependency graph. Consider a change to a source code file that repositions an object in a game. This repositioning may have a transitive impact on other objects within the location in the game. To trace the impact of that change, we need a graph that captures the dependencies in code as

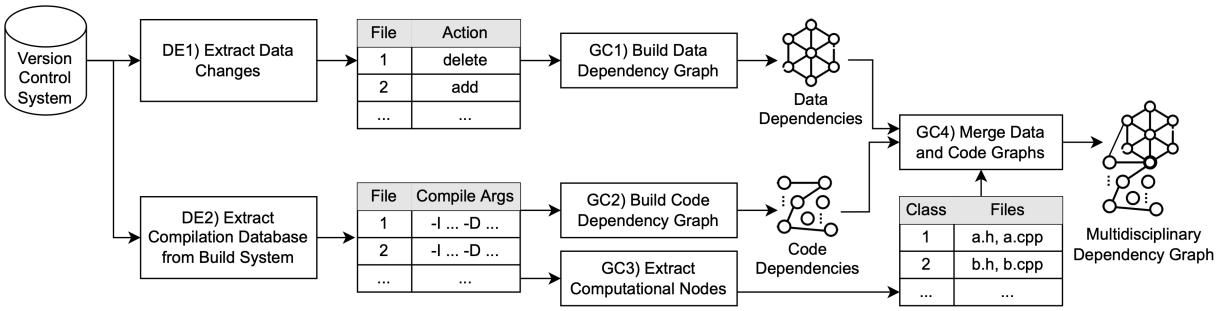


Figure 5.1: An overview of the graph extraction approach

well as data and their interdependencies. Inaccurate impact analysis due to an incomplete dependency graph will lead to under- or over-estimating the risk of a change.

While dependency graphs have been explored in the general development context [7, 39, 87], the multidisciplinary software context introduces challenges in the extraction and analysis of dependency graphs that need to be addressed. In this chapter, we show how such a graph can be extracted from a large video game project and explore properties of the extracted graph. A preliminary analysis of 4,256 revisions of the studied game project shows that 66% of the changes do not modify the structure of the graph. This number is 78% in a code-only dependency graph [17]. The 12 percentage point difference suggests that tools and approaches proposed in the previous works that use a prior version of the graph to analyse the current changes are applicable in multidisciplinary graphs; however, rapid incremental approaches are needed to update the graph and ensure its usefulness for all types of changes. We also show that 41% of the changes in the code files affect the data files as well, further emphasizing the importance of the multidisciplinary dependency graph.

The remainder of this chapter is organized as follows. We first describe our approach to extract the multidisciplinary graph in Section 5.2. Then, we present our research questions in Section 5.3. Section 5.4 discloses the threats to the validity of this study, and finally, Section 5.5 concludes the chapter.

5.2 Extracting the Multidisciplinary Graph

In this section, we present our graph extraction approach. Figure 5.1 provides an overview of our approach.

5.2.1 Data Extraction

We first group artifacts into code and data categories. Source code, header files, and libraries are categorized as code artifacts, while the rest of the files are categorized as data artifacts. This may include machine learning models, animations, sound, 3D models, and textures. Based on this categorization, the data extraction is split into tasks that we explain below.

DE1) Extract Data Changes

Nowadays, software projects may contain hundreds of thousands of non-code artifacts. These artifacts may depend on each other. For instance, the 3D model of a table in a game contains a surface, four legs, and a texture. The texture describes the characteristics of the surface like its hardness and smoothness. Similar to source code, to enable reuse, the surface, legs, and the texture are separate files on which the table model depends.

To keep the graph updated, we construct the next version of data dependency graph incrementally based on the previous graph and the changes. Thus, in this step, we extract all changed files and the corresponding actions in one specific commit from the version control system. This data helps the next step to build the data graph based on the action for each changed file.

DE2) Extract Compilation Database

We rely on the import statements in each source file to find other dependent source files. For example, in C++, the `#include` preprocessor directive imports referenced file within the context of the file in which the statement appears. The preprocessor resolves imported files by searching a list of directories called search path. While there are default search locations, the search path is often updated within the build system configuration files. In this study, we extract this information from the build system.

The `compile_commands.json` file is a clang standard file[1] that the build system can generate. For example, the studied project uses Sharpmake[2], a build automation tool similar to CMake. Sharpmake generates the aforementioned file by passing `-compdb` argument. This text file contains the following information for each compilation unit:

[1] https://clang.llvm.org/docs/JSONCompilationDatabase.html
[2] https://github.com/ubisoft/Sharpmake

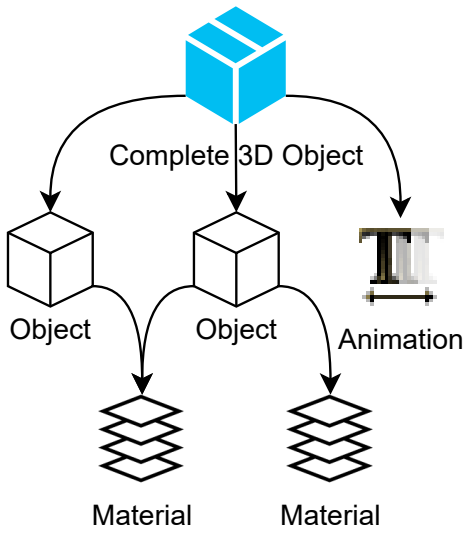


Figure 5.2: A 3D object (.uasset) may depend on multiple object files (.OBJ), which themselves depend on material files (.MTL)

- **file**: Path of the file in the compilation process.
- **directory**: The location from which relative paths are resolved.
- **command**: An array of arguments passed to the compiler for this specific compilation unit.

5.2.2 Graph Construction

Each node in the multidisciplinary dependency graph represents a code or data file. Edges in this directed graph indicate a dependency from a source file to a destination file. In the following steps, we describe how to construct the graph given the data extracted in the previous steps.

GC1) Build Data Dependency Graph

There are plenty of file types among the data artifacts. For example, animation files are different from 3D model files in terms of content and format. An `OBJ` 3D object file depends

on `MTL` material files using `mtllib` statement, and `MTL` files may use `.MPC` color texture files using `map_Ka` statement [2]. To account for this variety, we build a specific parser for each file extension. The parser analyses data files with the aim of identifying dependent files. At higher levels, files depend on multiple artifacts to build more complex objects. For example, `.uasset` file format from the Unreal game engine may contain animations, sounds, and textures to draw an object within the game. Figure 5.2 illustrates an example of such a hierarchy. We 1) read and parse each data file with the appropriate parser, 2) add the file to the graph, 3) mark it as visited to avoid adding duplicate nodes, and 4) recursively run the same algorithm on its children. Since this process is time-consuming and there are plenty of data files in a typical game project, we build the data graph incrementally. We start with the onerous creation of the initial graph but can then update it based on changed files. More specifically, after each commit, we perform one of the following actions:

- **Add**: For each added file, we add a new node to the graph and parse it.
- **Remove**: When a file is removed, the corresponding node and its edges are removed from the graph.
- **Rescan**: For each edited file, we remove the node and its edges, and reparse the file from the scratch. By removing the node and the edges, we ensure that we do not create duplicate dependencies from the edited node.

The output of this step is a directed graph representing the dependency among the data files.

GC2) Build Code Dependency Graph

File-level dependencies in code files are indicated using the `#include` preprocessor directive. In this step, we iterate over the compilation database file generated by step DE2 to identify source code files. We also analyze the '-I' arguments in the command section for each file and store them in a list with the same order as the compilation database. The '-I' argument specifies additional search locations for the preprocessor. Next, we resolve each file name passed to `#include` by checking the search locations in the same order of precedence as the preprocessor. If a node with the exact path does not exist in the graph, we create the node and recursively call the algorithm for this newly created node to traverse its dependencies. Finally, we connect the source node to the dependency under analysis to connect it to the rest of the graph. Note that it is guaranteed that the algorithm stops eventually, since we do not analyze existing (visited) nodes. The output of this step is an adjacency list, which represents the file-level dependency graph for code files.

GC3) Extract Computational Nodes

Eventually, the artifacts produced by different teams must be integrated to build the final software [60]. For example, in a game project, while artists are producing graphical artifacts, developers are working on the game engine and AI engineers are constructing better machine learning models. The game engine often enables a data-driven development approach [32]. Thus, developers provide generic and game-specific computational nodes to the engine, which can be later used by artists. For instance, the Move node takes a 3D model and changes its position from point A to B. These computational nodes are the bridge between code and data nodes [64].

In our study, computational nodes are defined as classes that inherit from the 'Node-Graph' class. The exact class name also appears in the data files, which are inputs to these nodes. We process each source file to detect classes that are defined in header files and implemented in C++ files. While this step is project specific, the general idea remains the same. In this project, the game has been developed with a data-driven approach. Developers do not dynamically load graphic files to avoid unexpected memory and CPU usage. Thus, dependencies flow from data to the code. In other projects, developers using dynamic loading may introduce dependencies from other directions as well. In either case, the output of this step is a map holding the one-to-many relationship between a class (*i.e.,* computational node) and code files.

GC4) Merge Data and Code Graphs

The intersection between the data and code graphs are the computational nodes (*i.e.,* edges from data to computational node as inputs, and from computational node to the code files where the node is implemented). Using the map generated by GC3, we identify the corresponding files for those computational nodes, and add edges from the data graph to the code graph (generated in GC2).

Since the dependency flow is from data to code, data nodes tend to have a larger outdegree, *i.e.,* number of edges directed out of the node, whereas code nodes have a larger indegree, *i.e.,* number of edges directed into the node. However, the centrality of code nodes, *i.e.,* how critical a node is in the graph, tends to be larger than that of the data node, since reuse of code seems more prevalent than reuse of data.

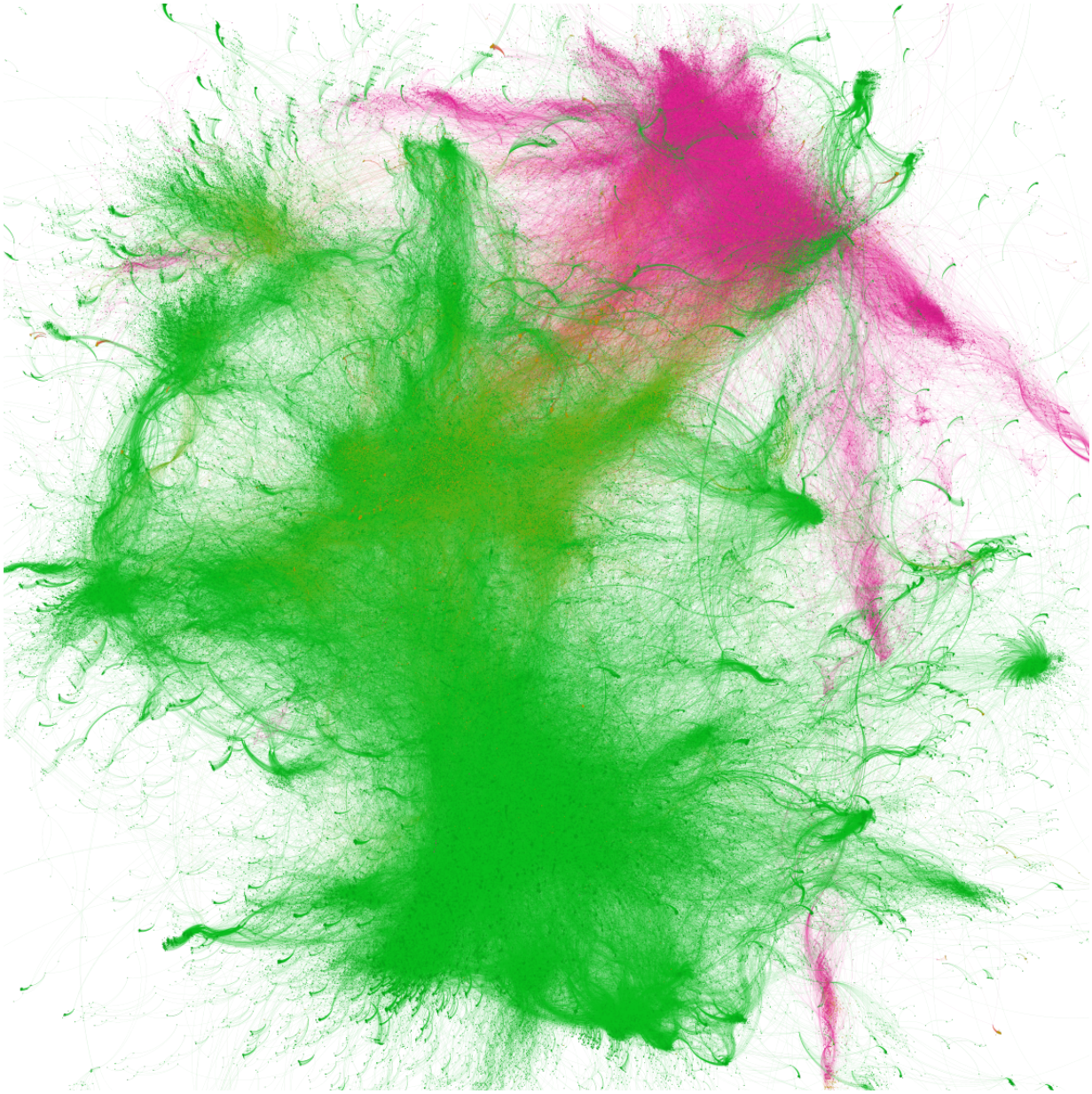


Figure 5.3: A multidisciplinary graph extracted from a game project. Pink nodes are code files, green are data files (*e.g.,* textures, animations, and music), and orange are the computational nodes that connect code and data.

5.3 A Case Study

In this section, we present a case study of the multidisciplinary graph extracted from a large game project at a multinational video game organization. Video game development is a good example of the type of development that involves a complex combination of code and data.

Figure 5.3 provides a visual overview of the multidisciplinary graph extracted from the studied game, which has been under active development since 2019. At the time of this writing, the project contains 13,555 commits to the code directory and 82,126 commits to the data directory. The latest version of the multidisciplinary graph includes 2,050,093 nodes and 7,516,457 edges. Data nodes dominate the graph, accounting for 98% of the nodes, while code and computation nodes represent the other 2%.

5.3.1 RQ1: How often is the graph itself changed?

Motivation: We plan to leverage the multidisciplinary graph to enable or enhance various software analytics tasks, *e.g.,* code quality assessment and failure prediction. The envisioned approaches will leverage a prior version of the graph to provide value for a current or future version. If most of the graph is frequently changed, the previous graph version might be obsolete and inadequate for our analytics task, and as a result, our analytical insights will be of limited value. To better understand the stability of the graph, we set out to analyze how often graph-modifying changes occur in a real project.

Approach: We analyze the replayed history of dependency graphs for both the code and data changes generated in Section 5.2. We merge the commit identifiers for changes in data and code locations within the codebase and sort them in ascending order by commit time. Then, for each unique commit, we look up the closest multidisciplinary graph in our historical records. Finally, using the depth-first search algorithm, we traverse both the prior and modified graphs and compare the nodes to gain more insight into how graphs are being modified.

Results: The graph is modified by 58% of the studied commits. A close inspection reveals that only 34% of the total commits invalidate the graph for dependency analytics (*e.g.,* change impact analysis). Indeed, the rest of the modifications only add new nodes (files) that are not yet connected to the rest of the graph. In other words, these new files are either orphan nodes or do not impact other nodes if they are changed (*i.e.,* they do not have predecessors in the graph). Thus, the graph can still be safely used for dependency analysis; however, incremental updates are required to keep the graph valid for analytics

tasks. Additionally, faster parsers and graph travel algorithms can help updating the graph for each change before doing any analysis, which is beyond the scope of this study.

5.3.2 RQ2: How often does the impact of a change cross disciplinary boundaries?

Motivation: Change impact analysis helps developers to understand the amount of work required to implement a change [11], perform risk assessment [5], and identify test results that have been invalidated and must be refreshed [71]. However, in multidisciplinary projects, changes may impact other file types as well. Therefore, in this question, we assess the frequency of changes with an impact that crosses the boundary between code and data.

Approach: We extract the graph for each change in the data or code locations within the codebase between early March and late May 2022. Out of 4,256 commits, we only keep ones that contain changed or deleted files because adding a new file without changing current ones will not impact existing nodes. Since the direction of edges in the graph is from a dependent node to its dependency, we need to reverse all the edges to analyze the impact. Next, we locate each node corresponding to the changed files and compute the transitive closure of each changed node to identify all the reachable nodes. Finally, we check the type of the nodes, *i.e.,* code, data, or computational, and compare it to the type of the change node. If node types differ, a cross-disciplinary impact has been detected.

Result: We find that 237 of the 2,620 analyzed commits (9%) have an impact that crosses disciplinary boundaries. While this is a small proportion of the overall commits, it accounts for 47% of the commits that change the source code. Note that changes in data cannot impact code nodes, since there are no dependencies from code to data. By comparison, commits that do not cross disciplinary boundaries impact a mean of 762,292 nodes and a median of 386,542 nodes. This suggests that the multidisciplinary graph adds substantial perspective for analytics.

5.4 Threats to Validity

In this section, we discuss the threats to the validity of our study.

5.4.1 Construct Validity

We extract the code dependency at the file level by reading the content of the source files and looking for import statements. In some cases, the import statement is inside a conditional statement. For example, some platform-dependent libraries are different between the deliverables built for Windows or PlayStation. While this may add extra nodes and edges to the graph, it does not affect the concepts introduced in this chapter.

Additionally, the compilation database generated in Section 5.2 is for the executable version of the game for Windows operating system. Each game can be compiled for different platforms, which generates different a compilation database. However, all the development related to the game in Ubisoft are platform-independent. Thus, this should not affect the results and concepts introduced in this study.

5.4.2 Internal Validity

Since the graphs generated for each revision is big enough for any graph traversal algorithm to perform quickly, we limited our study to the revisions generated from March 1st until the end of April. While there could be other explanations for the observations we present in this chapter, like a maintenance period or inactivity, we talk to the project managers and make sure that the project is under normal development.

5.4.3 External Validity

The results for the preliminary analysis and the research question comes from one project at Ubisoft. While this may hurt the generalizability of the study, it does not affect the concept of the multidisciplinary dependency graphs and its usage for future research and practical tools. Nonetheless, replication studies may help to improve the strength of generalizations that can be drawn.

5.5 Chapter Summary

In projects with a variety of artifacts, a dependency graph containing the relationship between the boundary of artifacts is needed. Therefore, we introduce the multidisciplinary dependency graph to connect the work products of different teams into a cohesive model of the system. Through a case study of a game project, we demonstrate the importance of the multidisciplinary graph in analytics tasks for multidisciplinary projects.

Chapter 6

Conclusion

Large-scale software systems often produce different variants that execute on different hardware and software platforms, or that restrict access to features. The projects might also involve teams specializing in different disciplines (*e.g.,* developers and artists), who produce various code-adjacent artifacts (*e.g.,* source code and 3D models). In these software projects, a change in the source code may impact a subset of the variants of the system while others remain unchanged. Additionally, that change may also impact the work of other teams, crossing the boundaries of code and other artifacts.

Software analytics tasks like impact analysis, planning for quality improvements, and predicting build failures rely on dependency information about software artifacts. However, current approaches that extract and analyze dependencies ignore these complexities. In this thesis, we empirically study how to enhance the dependency graph by including code-adjacent artifacts and different configuration settings. In the remainder of this section, we outline the contribution of this thesis and draw paths for future research.

6.1 Contribution and Findings

This thesis aims to enhance the build dependency graph to increase the accuracy of analytics tasks. To do so, we statically analyze the build specification files in highly configurable software projects and projects with multidisciplinary personnel. We claim that:

Thesis Statement: A build dependency graph that captures build-time configuration settings and code-adjacent artifacts can be leveraged to accurately assess the risk of software changes.

We propose an approach to extract and analyze the build-time configuration settings and code-adjacent artifacts to enhance the build dependency graph. In doing so, we perform two empirical studies. Below, we reiterate the key findings of the studies presented in this thesis:

1. *Assessing the Exposure of Software Changes, The DiPiDi Approach*
   By statically analyzing build specification files, we produce a dependency graph containing different build-time configuration settings. We show that such a graph can increase the efficiency and effectiveness of developers while assessing the risk of software changes. (Chapter 4)
2. *Dependency Extraction and Analysis for Multidisciplinary Teams: A Case Study at Ubisoft*
   Teams working on a software project may develop non-code artifacts, depending on each other or the code artifacts. A change in one of the artifacts may impact other types as well. In this thesis, we introduced the multidisciplinary dependency graph to represent the relationship between all the artifacts in one graph. We also show the importance of the multidisciplinary graph in software analytics tasks. (Chapter 5)

6.2 Opportunities for Future Research

This thesis provides approaches to enhance the dependency graph and provides evidence of its importance in research and practical tools. With this purpose in mind, there are plenty of opportunities for future research and practical tools. Below, we describe three paths for future work.

6.2.1 Enhanced Risk Assessment

We believe a change that impacts more software variants is inherently riskier than a change that impacts fewer variants. Additionally, a change that impacts the work of other teams may be riskier than a change that is local to a single team. For example, in a game project, a change to the code that impacts all of its targeted gaming platforms requires more effort from the quality assurance team than a change that only affects a single (and potentially

unpopular) gaming platform. Similarly, a change impacting rendering 3D objects requires more attention than a change to a single animation file. Both metrics (number of impacted variants and number of impacted artifact types) can be extracted by traversing the enhanced graph introduced in this thesis.

6.2.2 Leveraging Graph Metrics for CI Failure Prediction

A CI/CD pipeline contains steps to build the program, integrate multiple modules, run tests, and deploy software. We believe a change with higher centrality, *i.e.,* how critical a node is in the graph, higher degree, *i.e.,* number of in and out edges, and a higher number of predecessors is more likely to cause a CI failure. In large software projects, a CI pipeline may take hours to complete. When multiple changes are pushed to the repository in a short period, they should wait in a pending state until the CI pipeline starts for them. Since the CI/CD service server has limited resources, prioritizing the jobs with a higher likelihood of failure will result in faster notifications to developers while changes are still fresh in their minds.

6.2.3 Refactoring Build Code

Since the graph introduced in this thesis contains all the configuration settings and compilation paths for a software project, it can be leveraged to detect bad smells in the build specification files. For example, a code file might be excluded in all the configuration settings, so it should be safe to remove it from the project.

6.2.4 Graph Evolution

In this thesis, we did not study how many differences occur among the dependency graphs across different configuration settings. This is because the size of the impact on the structure of the graph was not the goal of this study. Nevertheless, we show that developers using our approach identified the impacted deliverables and the configuration settings more accurately and efficiently. Future work studying graph evolution would be interesting as it can be leveraged to study the impact of the changes on the build process itself.

References

[1] Json compilation database format specification.

[2] Mtl material format (lightwave, obj).

[3] *CMake Official Grammar*. https://cmake.org/cmake/help/v3.0/manual/cmake-language.7.html#syntax.

[4] *GNU make*. https://www.gnu.org/software/make/.

[5] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 746–755. IEEE, 2011.

[6] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *Electronic Communications of the EASST*, 8, 2008.

[7] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages 114–123. IEEE, 2007.

[8] Syed Nadeem Ahsan and Franz Wotawa. Impact analysis of scrs using single and multi-label machine learning classification. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 1–4, 2010.

[9] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Detecting semantic changes in makefile build code. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 150–159, 2012.

[10] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 432–441, New York, NY, USA, 2005. Association for Computing Machinery.

[11] Robert S Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[12] Robert S Arnold and Shawn A Bohner. Impact analysis-towards a framework for comparison. In *1993 Conference on Software Maintenance*, pages 292–301. IEEE, 1993.

[13] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[14] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 559–562. IEEE, 2015.

[15] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering*, 22(6):3117–3148, 2017.

[16] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 146–156. IEEE Press, 2015.

[17] Qi Cao, Ruiyin Wen, and Shane McIntosh. Forecasting the duration of incremental build jobs. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 524–528. IEEE, 2017.

[18] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333. IEEE, 2017.

[19] Jacob Cohen. Statistical power analysis. *Current directions in psychological science*, 1(3):98–101, 1992.

[20] Jason Cohen. Modern code review. *Making Software: What Really Works, and Why We Believe It*, pages 329–336, 2010.

[21] KDE Community. About kde, 2013.

[22] The Qt Company. Qt, 2020.

[23] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs. how the current code review best practice slows us down. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 27–28. IEEE, 2015.

[24] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robust approach for variability extraction from the linux build system. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 21–30, 2012.

[25] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American statistical association*, 56(293):52–64, 1961.

[26] Stuart I Feldman. Make—a program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

[27] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *2011 IEEE fourth international conference on software testing, verification and validation workshops*, pages 450–457. IEEE, 2011.

[28] Krita Foundation. Digital painting. creative freedom., Aug 2020.

[29] Malcom Gethers and Denys Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[30] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny Van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. *ACM SIGPLAN Notices*, 49(10):599–616, 2014.

[31] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[32] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.

[33] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 25–28. IEEE, 2018.

[34] Carl A Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(1):94–131, 2000.

[35] Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 318–328, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Frank E Harrell Jr, Maintainer Frank E Harrell Jr, and Depends Hmisc. Package 'rms'. *Vanderbilt University*, 229, 2017.

[37] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089. IEEE, 2018.

[38] Lile Hattori, Dalton Guerrero, Jorge Figueiredo, Joao Brunet, and Jemerson Damásio. On the precision and accuracy of impact analysis techniques. In *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, pages 513–518. IEEE, 2008.

[39] Yoshiki Higo and Shinji Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *2009 16th Working Conference on Reverse Engineering*, pages 315–316. IEEE, 2009.

[40] Lorin Hochstein and Yang Jiao. The cost of the build tax in scientific software. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 384–387. IEEE, 2011.

[41] Kdenlive. Libre video editor, 2020.

[42] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[43] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[44] Kitware. *CMake*, 2020. https://cmake.org.

[45] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[46] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[47] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[48] ET: Legacy. Et: Legacy, 2021.

[49] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[50] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114, 2010.

[51] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, Wen-Tin Lee, Shin-Jie Lee, and Nien-Lin Hsueh. Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 81–86. IEEE, 2018.

[52] Sasu Mäkinen, Henrik Skogström, Eero Laaksonen, and Tommi Mikkonen. Who needs mlops: What data scientists seek to accomplish and how can mlops help? In *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, pages 109–112. IEEE, 2021.

[53] Ken Martin and Bill Hoffman. *Mastering CMake: a cross-platform build system*. Kitware, 2010.

[54] Shane McIntosh, Bram Adams, and Ahmed E Hassan. The evolution of ant build systems. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 42–51. IEEE, 2010.

[55] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 141–150. IEEE, 2011.

[56] Mehran Meidani, Maxime Lamothe, and Shane McIntosh. Assessing the exposure of software changes: The dipidi approach. *arXiv preprint arXiv:2104.00725*, 2021.

[57] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[58] Sarah Nadi and Ric Holt. Mining kbuild to detect variability anomalies in linux. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, page 107–116, USA, 2012. IEEE Computer Society.

[59] Sarah Nadi and Ric Holt. The linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730–746, 2014.

[60] Matthew O'Connell, Cameron Druyor, Kyle B Thompson, Kevin Jacobson, William K Anderson, Eric J Nielsen, Jan-Reneé Carlson, Michael A Park, William T Jones, Robert Biedron, et al. Application of the dependency inversion principle to multi-disciplinary software development. In *2018 Fluid Dynamics Conference*, page 3856, 2018.

[61] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. Towards language-independent brown build detection. 2022.

[62] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. A curated benchmark collection of python systems for empirical studies on software engineering. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 1–4, 2015.

[63] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[64] Partha Sarathi Paul, Surajit Goon, and Abhishek Bhattacharya. History and comparative study of modern game engines. *International Journal of Advanced Computed and Mathematical Sciences*, 3(2):245–249, 2012.

[65] Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 10–19. IEEE, 2009.

[66] Christopher Preschern. Patterns to escape the# ifdef hell. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–12, 2019.

[67] Vaclav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE software*, 21(4):62–69, 2004.

[68] Gregorio Robles, Jesus M Gonzalez-Barahona, and Juan Julian Merelo. Beyond source code: the importance of other artifacts in software development (a case study). *Journal of Systems and Software*, 79(9):1233–1248, 2006.

[69] Robert Rosenthal. *Experimenter effects in behavioral research*. Irvington, 1976.

[70] Per Rovegård, Lefteris Angelis, and Claes Wohlin. An empirical study on views of importance of change impact analysis issues. *IEEE Transactions on Software Engineering*, 34(4):516–530, 2008.

[71] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, 2001.

[72] SciTools. Understand, 2022.

[73] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 724–734, New York, NY, USA, 2014. Association for Computing Machinery.

[74] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line. In *Proc. SPLC Workshop on Open Source Software and Product Lines*, 2007.

[75] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[76] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. Build code analysis with symbolic evaluation. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 650–660. IEEE, 2012.

[77] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. Symake: a build code analysis and refactoring tool for makefiles. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 366–369. IEEE, 2012.

[78] Qiang Tu and Michael W Godfrey. The build-time software architecture view. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 398–407. IEEE, 2001.

[79] John W Tukey et al. *Exploratory data analysis*, volume 2. Reading, Mass., 1977.

[80] Shuying Wang and Miriam AM Capretz. A dependency impact analysis model for web services evolution. In *2009 IEEE International Conference on Web Services*, pages 359–365. IEEE, 2009.

[81] Ruiyin Wen, David Gilbert, Michael G Roche, and Shane McIntosh. Blimp tracer: integrating build impact analysis with code review. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 685–694. IEEE, 2018.

[82] Mark Werner. Barriers to a collaborative, multidisciplinary pedagogy [software development teams]. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*, pages 203–210. IEEE, 1996.

[83] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Build system analysis with link prediction. SAC '14, page 1184–1186, New York, NY, USA, 2014. Association for Computing Machinery.

[84] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006.

[85] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[86] B. Zhou, X. Xia, D. Lo, and X. Wang. Build predictor: More accurate missed dependency prediction in build configuration files. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 53–58, 2014.

[87] Thomas Zimmermann and Nachiappan Nagappan. Predicting subsystem failures using dependency graph complexities. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 227–236. IEEE, 2007.

APPENDICES

Appendix A

Demographic Questions

1. How much experience do you have in programming?
   - None
   - Less than a year
   - a year to two years
   - two to five years
   - five years or more
2. How much are you familiar with CMake?
   - None
   - Tried it at least once
   - Used in professional development
3. How often do you currently program?
   - Never
   - Sometimes
   - More than once per week

Appendix B

Post Study Questionnaire

1. If you used any other tool(s) (CLI/IDE) please name it here:
2. If we provided a tool for you to use, how useful it was?
3. How do you feel? (1=Very Tired, 2=Tired, 3=Neutral, 4=Energetic, 5=Very Energetic)
4. How difficult were the tasks? (1=easy, 2=average, 3=hard)
5. How much experience did you have with the projects provided to you?
6. Did you encounter any problem during the experiment?
7. Any feedback about the experiment?
8. Can we contact you for a follow up interview?

B.1 Task A

You will be provided with the names of changed files and a set of build specifications. Your task is to list impacted deliverables (targets). Deliverables are defined in CMake files (CMakeLists.txt or .cmake files) using `add_library` or `add_executable` commands. You can find these files in the project repository. These commands take a target name and a list of files which impact the target. Some files may be excluded under different configuration. As an example, a code file related to ARM processor may not be included in the deliverable for Intel CPUs. Read more at https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html#binary-targets. The experiment UI provides text inputs for you to list those deliverables.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from https://github.com/libuv/libuv

Given the following commit id and the build time configuration, please find the impacted targets (deliverables). There maybe more input fields than necessary to complete the task.

1. Change Commit ID: cdced3a3ad1b3e4287f92c9d434b543a9e509938
2. Build Configuration: APPLE==False

Input1: ...
Input2: ...

B.2 Task B - (Impacted Deliverables)

You will be shown three commits and a set of build specifications which are the conditions passed to the build system and may change the build process. These conditions are defined using option or if commands. Read more at https://cmake.org/cmake/help/latest/command/if.html. We ask you to rank the commits listed in the experiment UI based on the number of impacted deliverables. Rank the commits in an ascending order (1=Most Impact, 3=Less Impact)

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from https://github.com/libuv/libuv

Given the following build time configurations, please rank the commits based on the given criteria.

1. Build Configurations: MAKE_SYSTEM_NAME==APPLE
2. Criteria: Impacted Deliverables

1. e89abc80ea43065a726ade191b810af53ec6158a: ...
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ...
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ...

B.3 Task B - (Impacted Variants)

You will be shown three commits and a set of build specifications which are the conditions passed to the build system and may change the build process. These conditions are defined using option or if commands. Read more at https://cmake.org/cmake/help/latest/command/if.html. We ask you to rank the commits listed in the experiment UI based on the number of impacted application variants (e.g., number of affected OS). Rank the commits in an ascending order (1=Most Impact, 3=Less Impact)

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from https://github.com/libuv/libuv

Given the following build time configurations, please rank the commits based on the given criteria.

1. Build Configurations: LIBUV_BUILD_TESTS==False
2. Impacted Application Variants (Operating systems)

1. e89abc80ea43065a726ade191b810af53ec6158a: ...
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ...
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ...

B.4 Task C - (Identify Commits Affect Deliverables)

You will be shown three commits and asked to identify the commits that affect a specified set of deliverables.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from https://github.com/libuv/libuv

Identify the commits which affect these deliverables: ['uv']

1. e89abc80ea43065a726ade191b810af53ec6158a: ?
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ?
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ?

B.5 Task C - (Identify Commits Affect Variant)

You will be shown three commits and asked to identify the commits that affect a specific variant of the software.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from https://github.com/libuv/libuv

Identify the commits which affect this variant: BSD Operating System

1. e89abc80ea43065a726ade191b810af53ec6158a: ?
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ?
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ?

B.6 Task C - (Configuration Setting)

You will be shown three commits and asked to identify the configuration settings under which the changes will affect at least one target. The build configurations may exclude or include a file in the build process for an specific target using conditional commands in the CMake files. Read more at https://cmake.org/cmake/help/latest/command/if.html CMake website.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from https://github.com/libuv/libuv

For each of the given commits, identify at least one configuration setting under which the change will propagate to at least one deliverable(target). If the change will propagate irrespective of the conditional settings, enter the term "ALL".

1. e89abc80ea43065a726ade191b810af53ec6158a: ...
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ...
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ...