Quantifying, Characterizing, and Leveraging Cross-Disciplinary Dependencies

Empirical Studies from a Video Game Development Setting

by

Gengyi Sun

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, 2023

Author's Declaration

Abstract

Continuous Integration (CI) is a common practice adopted by modern software organizations. It plays an especially important role for large corporations like Ubisoft, where thousands of build jobs are submitted daily. The CI process of video games, which are developed by studios like Ubisoft, involves assembling artifacts that are produced by personnel with various types of expertise, such as source code produced by developers, graphics produced by artists, and audio produced by musicians and sound experts. To weave these artifacts into a cohesive system, the build system—a key component in CI—processes each artifacts while respecting their intra- and inter-artifact dependencies. In such projects, a change produced by any team can impact artifacts from other teams, and may cause defects if the transitive impact of changes is not carefully considered.

Therefore, to better understand the potential challenges and opportunities presented by multidisciplinary software projects, we conduct an empirical study of a recently launched video game project, which reveals that code files only make up 2.8% of the nodes in the build dependency graph, and code-to-code dependencies only make up 4.3% of all dependencies. We also observe that the impact of 44% of the studied source code changes crosses disciplinary boundaries, highlighting the importance of analyzing inter-artifact dependencies. A comparative analysis of cross-boundary changes with changes that do not cross boundaries indicates that cross-boundary changes are: (1) impacting a median of 120,368 files; (2) with a 51% probability of causing build failures; and (3) a 67% likelihood of introducing defects. All three measurements are larger than changes that do not cross boundaries to statistically significant degrees. We also find that cross-boundary changes are: (4) more commonly associated with gameplay functionality and feature additions that directly impact the game experience than changes that do not cross boundaries, and (5) disproportionately produced by a single team (74% of the contributors of cross-boundary changes are associated with that team).

Next, we set out to explore whether analysis of cross-boundary changes can be leveraged to accelerate CI. Indeed, the cadence of development progress is constrained by the pace at which CI services process build jobs. To provide faster CI feedback, recent work explores how build outcomes can be anticipated. Although early results show plenty of promise, prior work on build outcome prediction has largely focused on open-source projects that are code-intensive, while the distinct characteristics of a AAA video game project at Ubisoft presents new challenges and opportunities for build outcome prediction. In the video game setting, changes that do not modify source code also incur build failures. Moreover, we find that the code changes that have an impact that crosses the source-data boundary are more prone to build failures than code changes that do not impact data files. Since such changes are not fully characterized by the existing set of build outcome prediction features, state-of-the-art models tend to underperform.

Therefore, to accommodate the data context into build outcome prediction, we propose RavenBuild, a novel approach that leverages context, relevance, and dependency-aware features. We apply the state-of-the-art BuildFast model and RavenBuild to the video game project, and observe that RavenBuild improves the F1-score of the failing class by 46%, the recall of the failing class by 76%, and AUC by 28%. To ease adoption in settings with heterogeneous project sets, we also provide a simplified alternative RavenBuild-CR, which excludes dependency-aware features. We apply RavenBuild-CR on 22 open-source projects and the video game project, and observe across-the-board improvements as well. On the other hand, we find that a naïve Parrot approach, which simply echoes the previous build outcome as its prediction, is surprisingly competitive with BuildFast and RavenBuild. Though Parrot fails to predict when the build outcome differs from their immediate predecessor, Parrot serves well as a tendency indicator of the sequences in build outcome datasets. Therefore, future studies should also consider comparing to the Parrot approach as a baseline when evaluating build outcome prediction models.

Acknowledgements

I would like to thank my supervisor, Professor Shane McIntosh, for his pivotal role in my academic journey, from accepting me as a master's student to providing internship opportunity and offering a position as a Ph.D. student in his research group. He is always passionate about his work and passes on a positive attitude to his students. I will learn from the attitude and keep going.

I'm profoundly thankful to Dr. Sarra Habchi for her mentorship and guidance. Her comments are accurate and pinpointing to the core concept, and always lead me to improve the work.

Special thanks to Professor Chengnian Sun and Professor Pengyu Nie for being my thesis readers and providing constructive feedback to this thesis.

My sincere thanks go to Xueyao and Zhili for their companionship during the challenging times of the pandemic. To Mahtab, Adrian, and Albert, who supported me when I am stressed. To Nimmi, Farshad, Mehran, and all my SWAG labmates, your encouragement was a constant source of motivation. To Yangyang, Zhuoran, Zhuo and Xiya, for our friendship since high school and every Christmas holiday we spent together since then. To Huilong, for supporting me going further in academics.

I would like to thank my parents, who support me at all time and in all ways since I was born. My life would not have been so wonderful without you respect and trust me making my life decision when I was only 13. Special thank to my mother, you are my life example as a wise and strong woman.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software organizations adopt continuous integration (CI) [19] to automate the build and test jobs as modern software development becomes increasingly complex. While source code is an essential component, it does not represent the entirety of artifacts within software systems. Non-code artifacts, such as machine learning models, multi-media files, and infrastructure specification files are also commonly part of software systems. These artifacts must be assembled while respecting their intra- and inter-dependencies. To weave these artifacts into a cohesive system, build systems specify and resolve the internal and external dependencies and their usage conditions. In addition, build systems preprocess, compile, assemble, link, analyze, and package software artifacts into deliverables according to a *dependency graph*, *i.e.,* a directed graph with nodes representing software entities and directed edges indicating dependencies between artifacts.

At Ubisoft, dependency graphs are composed of artifacts that are produced by personnel with various types of expertise, such as source code produced by developers, graphics produced by artists, and audio produced by musicians and sound experts. Meidani [50] showed that nodes from different disciplines are connected through a special type of node that lies on discipline boundaries (*i.e.,* boundary nodes), resulting in a multidisciplinary dependency graph.

Rapidly developed software requires more frequent CI repetitions. CI enables rapid feedback by invoking build and test jobs for the change sets that development teams produce. For large organizations, adopting CI can accelerate development [72], but it is not without costs. The execution of a slow CI job can delay time-to-feedback [25], forcing developers to perform a mentally taxing context switch [85] to another task or remain effectively idle. Moreover, CI execution consumes a large quantity of computational resources [32, 34], which can quickly accrue annual costs on the order of millions of dollars [35]. Ubisoft heavily invests in the execution of builds for its games. For example, for Project X (the latest installment in a ten-year video game franchise), thousands of build jobs are submitted daily, consuming an average of 49 build hours on the main branch. The build duration for Project X ranges between 10 to 30 minutes, delaying developers from receiving immediate feedback and incurring expensive computational costs.

1.1 Problem Statement

Since a multidisciplinary build dependency graph shows the intra- and inter-dependency relationships among artifacts produced by teams with different expertise, it can be used to measure the change impact on other artifacts, providing insights into the riskiness of a change set:

Thesis Statement: The presence and magnitude of cross-boundary change can be leveraged to enhance Continuous Integration.

In a multidisciplinary context, it is not unexpected for change sets to have impacts that cross disciplinary boundaries; however, the frequency of cross-boundary changes and their implications are not well understood. Thus, we conduct an empirical analysis of cross-boundary changes.

Dependency analytics can provide estimates of the impact of changes on interconnected files [7, 74], which can potentially be leveraged to assist CI acceleration. To demonstrate the application of dependency analytics, we propose RavenBuild, which predicts the build outcome with features that are context, relevance, and dependency-aware.

1.2 Thesis Overview

Figure 1.1 provides an overview of the scope of this thesis. We first provide the background, definitions, and related work of our topic:

Chapter 2: *Background and Definitions*
Before discussing the work in this thesis, we first provide the readers with background information on dependency graphs, cross-boundary changes, and build outcome prediction.

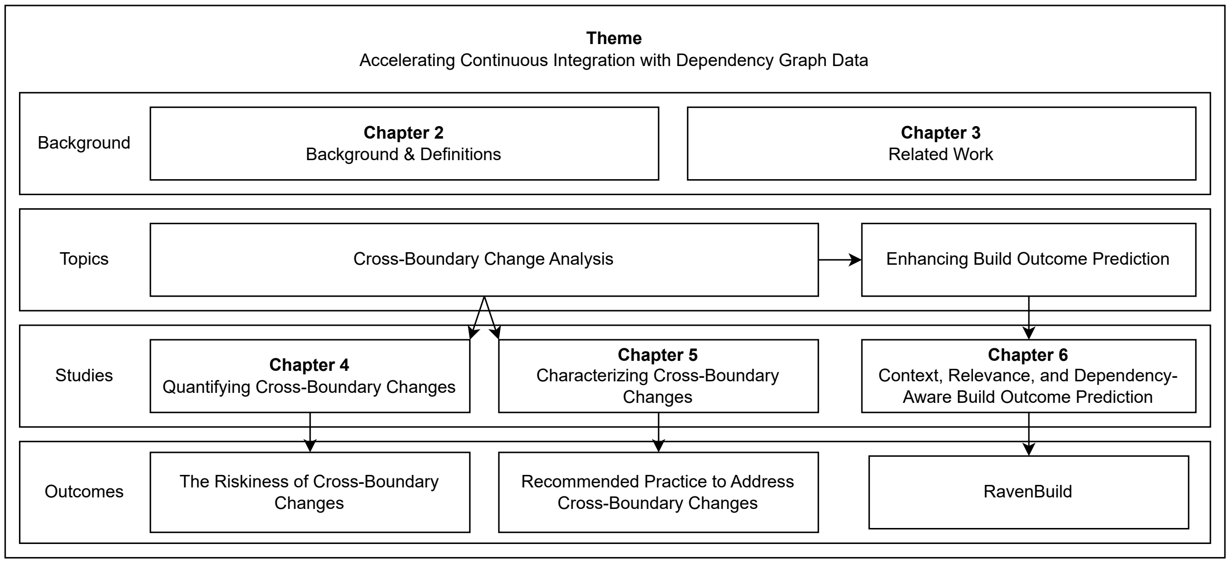
Theme
Accelerating Continuous Integration with Dependency Graph Data
Background
Chapter 2
Background & Definitions
Chapter 3
Related Work
Topics
Cross-Boundary Change Analysis
Enhancing Build Outcome Prediction
Studies
Chapter 4
Quantifying Cross-Boundary Changes
Chapter 5
Characterizing Cross-Boundary Changes
Chapter 6
Context, Relevance, and Dependency-Aware Build Outcome Prediction
Outcomes
The Riskiness of Cross-Boundary Changes
Recommended Practice to Address Cross-Boundary Changes
RavenBuild

Figure 1.1: An overview of the scope of this thesis.

Chapter 3: *Related Work*
In this chapter, we discuss how dependency analytics can assist in prior software engineering research and prior work on build outcome prediction.

Chapter 4: *Quantifying Cross-Boundary Changes*
In this chapter, we present our quantitative analyses on multidisciplinary dependency graphs. We study the missing information of a code-only dependency graph and the occurrence of code changes that have an impact that crosses the disciplinary boundaries. Finally, we discuss the impact scale and the riskiness of changes that do or do not cross discipline boundaries.

Chapter 5 *Characterizing Cross-Boundary Changes*
In this chapter, we present our characterization of cross-boundary changes. We study the activities performed by cross-boundary changes and the contributors that produce cross-boundary changes.

Chapter 6: *Context, Relevance, and Dependency Aware Build Outcome Prediction*
In this chapter, we propose RavenBuild, which enhances build outcome prediction with file type-agnostic and dependency-aware features that accommodate change sets containing non-code files only to complement the prior works.

1.3 Thesis Contributions

This thesis has both empirical contributions as described below:

1.3.1 Empirical Contributions

This thesis demonstrates that:

1. In our video game development setting, the vast majority of nodes (97.2%) and edges (95.7%) will be missed if non-code artifacts are excluded from the dependency graph (Chapter 4).

2. Changes crossing disciplinary boundaries are a common daily occurrence that represents 44% of code changes and 6% of overall changes, moreover, they impact a significantly larger amount of nodes compared to other changes (Chapter 4).

3. Cross-boundary changes tend to break builds and induce defects with significantly greater rates than changes that do not cross boundaries (Chapter 4).

4. Cross-boundary changes are more often associated with GamePlay, Feature, and UI tags, as well as file additions and updates, whereas code changes that do not cross boundaries are more often associated with Tools and bugfix tag. Non-code changes are rarely tagged (Chapter 5).

5. Cross-boundary changes are from the Programming group only, while non-code changes have various group origins. Contributors from one team commit most of the cross-boundary changes (Chapter 5).

1.3.2 Technical Contributions

The technical contribution of this thesis is build outcome prediction with features that are context, relevance, and dependency-aware, namely RavenBuild. We apply the state-of-the-art BuildFast model [13] and our proposed RavenBuild enhancements to Project X, and observe that RavenBuild improves BuildFast by 46% in the F1-score of the failing class, 76% in the recall of the failing class, and 28% in AUC. We also provide a simplified alternative, RavenBuild-CR, which only adopts context-aware and relevance-aware features. We apply RavenBuild-CR on 22 open-source projects and Project X, and observe across-the-board improvements as well (Chapter 6).

Chapter 2

Background and Definitions

In this chapter, we introduce the background and definitions of this thesis. First, we describe a build system and a build dependency graph. Then we talk about the characteristics of a video game project and the challenges of applying build outcome prediction to the project.

2.1 Build System

A build system is a software tool used to automate the process of compiling, linking, and packaging computer program source files into an executable. It is an essential part of the software development process, especially in large software organizations where projects are complex and involve files from multiple disciplines, internal and external dependencies, and platform-dependent configurations.

2.1.1 Build Dependency Graph

A build dependency graph is a visual or logical representation of the dependencies between various components in the software build process. It specifies the relationships between different files, such as source codes, libraries, and modules that are necessary to successfully build the final executable.

The build dependency graph is a directed acyclic graph where nodes represent build targets and edges represent dependencies. The build system processes build targets in the dependency graph in topological order.

2.2 Build Outcome Prediction

Build outcome prediction is the process of anticipating the outcome of a software build process before its execution. It involves using historical data, machine learning models, or rules to forecast the outcome of a build.

The goal of build outcome prediction is to save time and resources by taking appropriate execution strategies to address the anticipated outcome. For example, the passing builds can be either skipped or run in a batch to save executions, and the failing builds can be preflighted (*i.e.,* build locally before merging into the codebase) to avoid breaking build pipelines.

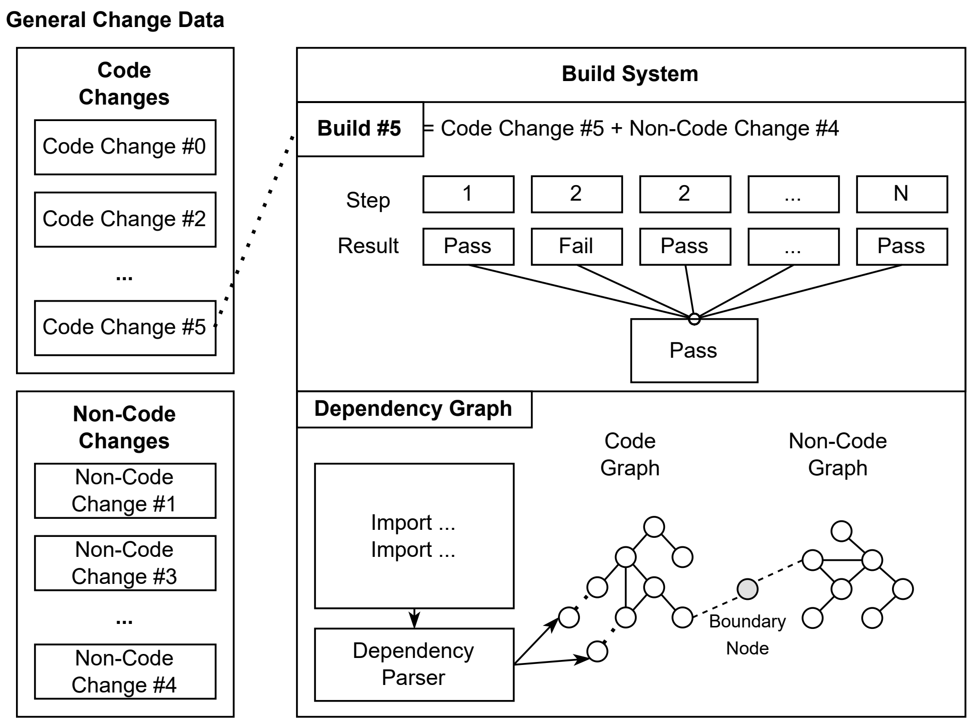

Figure 2.1: Characteristics of Project X build system.

2.3 Characteristics of Project X

Project X is the most recent installment of a popular video game in a franchise that was first released more than ten years ago by Ubisoft. Project X contains millions of production files, including source code, graphical assets, and audio samples. Throughout its development history, more than 10 million build steps have been logged, providing a rich source of build data.

Project X is rapidly evolving. Thus, to mitigate concept drift [46], we select a sample of 4,640 consecutive code and non-code builds from a recent 80-day period. During the period, an average of 58 build jobs were submitted daily, each taking an average of 21 minutes to execute. Each build in Project X is executed and logged in a stepwise fashion. On average, each build comprises 27 steps, with each step taking an average of 46 seconds to execute. In total, the code and non-code build jobs on the main branch of Project X consume 21 build hours daily. This excludes builds that are performed on other branches in Project X and the other games being developed by Ubisoft. With this in mind, it becomes clear why opportunities to reduce the project and organizational build workload are being actively explored by Ubisoft.

General Change Data: Every change made to Project X is logged in the General Database, and is indexed by a change number. The database includes typical change metadata, such as the author, timestamp, and updated file lists. The General Database provides the necessary data to compute process features for build outcome prediction, such as the number of lines of code added/edited/deleted, file revisions, committer identity, and file types.

Both code and non-code changes are stored in the General Database. At build time, the change will take the most recent successfully built change from the other discipline to assist the build process. Figure 2.1 shows an example of the build process, code change #5 triggers build #5 that is built on non-code change #4, which already passed its build #4.

Build System: The build process of video games is particularly complex due to the size of the game project and the variety of file types that each must be transformed and assembled in a precise (and often unique) manner. Depending on the complexity of the change set, and the corresponding build, it may invoke hundreds of steps. To extract the build outcome, we first group the build steps in the Build Step Log Database by their change numbers and step names, as shown in Figure 2.1.

Builds can fail for many reasons. For example, network failures or bandwidth limits can introduce non-deterministic build failures even if the code and artifacts are not to blame. If a step fails, the Project X build system may automatically re-invoke it. Thus, the existence of a failing step does not necessarily indicate that the build outcome is a failure. Olewicki *et al.* [58] refer to these non-deterministic build outcomes as “brown builds”. To mitigate the noisy influence of brown builds, we label builds with failure outcomes only when the latest invocation of a build step that is associated with the build is failing.

Multidisciplinary Dependency Graph: Video games are composed of a broad mix of digital artifacts in addition to source code [54], such as images, textures, and other so-called assets. Since these other digital artifacts are rapidly changing in tandem with the source code, and their representations are often in binary form, traditional software engineering features like code churn cannot always be collected for change sets in this context. To better describe the non-code changes, we extract the dependency graph of Project X with the non-code artifacts.

Similar to dependencies among code modules, other digital artifacts in video games also depend on each other in compositional and hierarchical ways, *e.g.,* a city is composed of buildings, each of which being composed of building frames, doors, and windows, which may inherit properties like the texture weight and hardness of their composing materials. Therefore, dependency graphs can be constructed from non-code artifacts as well. In this way, dependency graphs of video games are multidisciplinary, *i.e.,* they combine the work products of software engineers (code), artists (texture, images), game designers (level design artifacts), and many more.

The complete dependency graph of Project X is not directly generated by the game engine (a critical component used in the development and build process of a video game). To construct a multidisciplinary dependency graph of Project X, Meidani [50] analyzed the binary file outputted by the game engine, which contains dependency information of the non-code artifacts, and the compile commands of the source codes, which contain the code dependency information. Therefore, each dependency graph of Project X consists of a code dependency sub-graph and a non-code dependency sub-graph. At the disciplinary boundaries, code and non-code nodes interact with each other through boundary nodes. Boundary nodes do not exist as files, instead, they are generic and game-specific computational nodes provided by the developers for the artists to integrate their work into the game.

2.4 Challenges of Applying Build Outcome Prediction to Project X

A close inspection of the Project X setting reveals two characteristics that hinder the application of build outcome prediction.

Challenge 1: Changes to non-code files often break the build In video game development, non-code files are essential components that define the game's content, behavior, and appearance. The name "non-coe file" is a general name for textures, models, animations, etc. In Project X, there are more non-code files than source files (*e.g.,* `.cpp` and `.h`) in the repository. As an example, the latest multidisciplinary dependency graph in our dataset contains 1,104,037 nodes in total, while code nodes only account for 30,675 of them (2.8%).

In addition to the large quantity, in Project X, non-code files are modified more frequently than code files. Indeed, of the 277 types of modified files, header files (`.h`) and source code (`.cpp`) rank as the 11^{th} and 14^{th} most frequently modified file types, respectively.

If builds are triggered by changes that only modify non-code files, they are referred to as non-code builds. Just like builds triggered by code changes, non-code builds in video game development can also fail, since non-code builds also involve compiling the various non-code files into a format that the game engine can efficiently use during gameplay as specified by their dependencies. Indeed, build failures incurred by non-code builds account for 61% of all the build failures in our dataset.

Non-code builds and their failures pose a challenge for traditional build outcome prediction approaches because prior build outcome features are mainly designed to represent code changes (*e.g.,* using the number of changed source code lines) that do not apply to non-code files. It is also impractical to design such features on non-code files since there are too many non-code file types.

Challenge 2: Code changes with a cross-boundary impact break the build more often than code changes that have a code-only impact In video game development, the interaction between code and non-code files defines the game functions, manages the game assets, and determines the player experience of the game. Cross-boundary changes occur when a code change updates a code file that is depended upon by non-code files. The impacted nodes refer to the nodes that are dependent on the changed node.

Table 2.1 shows that cross-boundary changes impact a substantial amount of nodes in the dependency graph with a mean of 212,104 and a median of 120,368 nodes impacted,

Table 2.1: Build failure rates and the number of impacted nodes of non-code and code changes. non-code builds also incur build failures. Cross-boundary changes are more prone to build failures than other code changes.

	Total	Fail	Cross Boundary	Total	Fail	Mean Impact	Median Impact
Non-Code	4,010	11%	N/A	N/A	N/A	452	2
Code	630	46%	Yes	278	51%	212,104	120,368
			No	352	41%	49	0[a]

[a] Impacted nodes counts exclude the directly changed nodes.

whereas non-code changes and changes that do not cross boundaries only impact a mean of 452 and 49 nodes, and a median of 2 and 0 nodes, respectively. The statistical significance of the discrepancies is confirmed by Mann-Whitney U tests (two-tailed, unpaired $\alpha = 0.05$), yielding Holm-Bonferroni corrected p-values of 2.2×10^{-162} and 1.5×10^{-107}, respectively.

In addition to a larger scale of impact, Table 2.1 also shows that 278 cross-boundary changes account for 44% of the code changes, with 51% of cross-boundary changes and 41% of the changes that do not cross boundaries being implicated in build failures. The statistical significance of this discrepancy is confirmed by Boschloo's Exact test, yielding a p-value of 0.008. Therefore, cross-boundary changes are more likely to break the build pipeline than those changes that do not cross boundaries.

While cross-boundary changes do not (strictly speaking) present a challenge for the adoption of build outcome prediction in the Project X setting, they do present an opportunity that previous approaches have not yet exploited. Since changes that cross boundaries are significantly more likely to fail than changes that do not cross boundaries, we believe that leveraging this association will improve build outcome prediction in the Project X setting, and more broadly across Ubisoft.

2.5 Chapter Summary

In this chapter, we introduce the core concepts of build systems, build dependency graphs, and build outcome prediction. Additionally, the chapter outlines the distinct characteristics and challenges of video game development, highlighting complexities of applying build outcome prediction on such projects.

In the next chapter, we will describe the prior research on dependency analytics and application of dependency analytics such as code reviewer recommendation and CI acceleration. We also introduce prior research on CI and build outcome prediction.

Chapter 3

Related Work

In this section, we situate our work with respect to the literature on dependency analytics (Section 3.1), the benefits and costs of CI (Section 3.2), and the sets of features that are typically used for build outcome prediction (Section 3.3).

3.1 Dependency Analytics

Dependency graphs have long been at the heart of software build systems. For example, Feldman [23] introduced the Make build system, which uses a depth-first search of the file-level dependency graph to keep program deliverables up to date with their dependencies. Other researchers proposed methods to construct and symbolically analyze dependency graphs that are extracted from build tools statically [51, 70]. Inspired by this work, we incorporate static code analysis to extract the nodes that connect the non-code and code graphs to each other.

There is also research that leverages dependency graphs to tackle other software engineering challenges. For example, Zimmermann *et al.* [84] used complexity metrics extracted from dependency graphs to predict subsystem failures. Cao *et al.* [12] forecast the duration of the build based on the changed nodes in the dependency graph.

In this section, we discuss the previous research that has incorporated dependency graphs, and future work that could benefit from incorporating dependency analytics with respect to two existing software engineering challenges.

3.1.1 Dependency Analytics In Code Reviewer Recommendation

Code Reviewer Recommenders (CRRs) assist in the code review process by automatically suggesting potential reviewers for a given code change. CRRs recommend individuals who have the relevant expertise to review the code changes using (meta)data about a change, the review(er) tendencies of the modified files, and/or the relationship among reviewers and reviews.

Research on various approaches and their potential benefits for CRRs have long been explored. Some studies leverage the reviewer's experience and review history on changed files as a basis for recommendation [10, 15, 59, 71, 81]. Others focus on file paths [22, 76] or social network graphs [59, 64, 80]. Hirao *et al.* [37] demonstrated the performance of CRRs tend to improve if the linkage between reviews is considered. Zhang *et al.* [82] take a graph neural network learning approach called CORAL, which learns from a socio-technical graph constructed using reviewer experience, reviewer history, and changed file relations extracted from comment and review history. In addition to the existing works, our multidisciplinary dependency graph can be incorporated within such socio-technical graphs to provide direct dependencies among files, file authors, and reviewers, further enhancing the representativeness of the graph.

To assess the usefulness of CRRs and identify valuable factors in selecting code reviewers from the perspective of developers, Kovalenko *et al.* [43] conducted a survey of the considered factors when developers are selecting code reviewers. The results showed that working in an area that is dependent on and depended upon by the changed files ranked 4th and 6th, respectively. These factors highlight the importance of selecting reviewers with experience in dependency-related areas.

Indeed, dependency analytics can be a valuable tool in enhancing automated CRRs by providing the accurate impact of changes on interconnected files. Unlike relying on common file paths or co-change frequencies, dependency analysis considers the relationships between files. In this paper's context, when the change impact crosses boundaries and impacts a large set of files, relying on the file paths would omit review input from experts in other domains.

3.1.2 Dependency Analytics for the Acceleration of Continuous Integration

Continuous Integration (CI) acceleration refers to the optimization of CI processes in software development, including automated build, test, and deployment. It aims to reduce the time-to-feedback for developers and save resources while ensuring software quality.

Jin and Servant [77] conducted an evaluation of existing techniques for Continuous Integration (CI) acceleration and synthesized the design decisions that proved effective in accelerating CI processes. One extensively studied topic in CI acceleration is the concept of build skipping, which involves skipping builds either entirely (build selection) or partially (test selection).

To skip a build entirely, a model is employed to infer change information and predict the build result before the code change is processed by the build system. This model can be rule-based [5, 41, 66] or trained using statistical or machine learning approaches [4, 13, 30, 31, 39, 60, 67, 78]. Similar rules and features can be crafted from to exploit the multidisciplinary dependency graph. For example, as shown by Chapter 2, cross-boundary changes impact a substantial amount of nodes and incur more build breakages than changes that do not cross boundaries. We suspect that an indicator of cross-boundary impact will improve build outcome prediction, and have begun to explore its applications within the Ubisoft context.

Partial build skipping can be achieved through regression test selection [21], which involves selecting a subset of tests from the existing test suite to execute, thereby reducing feedback time and resource usage. Dependency analytics can also assist in test selection. The file-level dependencies captured in the dependency graph allow for effective regression test selection. Gligoric *et al.* [27] developed EKSTAZI—a test selection approach that analyzes dependencies to decide which tests must be executed. Gligoric *et al.* also discussed the limitation of FaultTracer [83], which constructs an extended call graph (*i.e.,* method-level) and identifies the suspicious changes and the impacted tests. Gligoric *et al.* conducted experiments to compare dependency granularities at the method, class, and file levels, concluding that using the file-level dependencies, *i.e.,* the same granularity as our multidisciplinary dependency graph operates is the safest because it also captures external file dependencies.

Furthermore, by adopting dependency analytics, the build system can identify unaffected files and artifacts, avoiding duplicate builds on unchanged binaries. This can substantially reduce waste in the build process. Indeed, Gallaba *et al.* [24] showed that unaffected build steps could be effectively skipped by caching the build environment and inferring file dependencies.

Dependency analytics can indeed assist CI acceleration by providing the accurate impact of changes on interconnected files. In this paper's context, resources spent on regenerating deliverables for changes that do not have a large-scale impact (*e.g.,* non-code changes and those that do not cross boundaries in our project) could be potentially saved.

We are actively pursuing follow-up work on making such improvements at Ubisoft.

3.2 Benefit and Cost of CI

CI is widely adopted in modern software organizations [35] to automate integration and release routines [45, 72]. Prior work reports that developers face trade-offs [34] when adopting CI. As the software evolves, the cost of maintaining [49] and executing [35] build tools tend to grow. Therefore, researchers have invested in reducing the time-to-feedback and resource consumption.

To accelerate CI feedback, build outcome prediction has been proposed [31, 66, 67]. Hassan and Zhang [30] use decision trees to predict the outcome of certification testing. To conserve computational resources, Jin and Servant [5, 38, 39, 41] propose to skip passing builds using machine learning classifiers. A less aggressive approach is to select and only (re-)execute the tests that failed in prior builds [20, 32, 52, 60] and/or skip the tests that are unaffected (i.e., likely to pass) [21]. Gallaba *et al.* [24] propose Kotinos—a CI service that skips unaffected build steps in a technology-agnostic fashion. Jin and Servant also propose HybridCISave [40], which skips the entire builds, as well as tests within builds, that are anticipated to pass.

The context-aware, relevance-aware, and dependency-aware enhancements that we propose in this paper complement and enhance build outcome prediction. In addition, while prior work has focused on the code and test aspects of change sets, to the best of our knowledge, this paper is the first to expand build outcome prediction to accommodate the data changes.

3.3 Build Prediction Features

Features that are extracted for build outcome prediction can be classified into those that characterize: (1) current build data, (2) previous build data, (3) historical data, and (4) the relevance between the last and the current build. Hassan and Wang [31] use features that characterize current and previous builds. Chen *et al.* [13] use current, previous, and historical features to provide faster time-to-feedback. To execute as many failing builds as early as possible, Jin and Servant [39] propose SmartBuildSkip that predicts the first builds in a sequence of build failures and the consequent build failures separately with current build data. To the best of our knowledge, only Ni and Li [56] use the relevance features and refer to them as “connection to last push”. Our proposed RavenBuild features differ in that RavenBuild characterizes the change-level relevance between the current build and its immediate predecessor, rather than the status of source or configuration files in the last build.

In addition to build outcome prediction features, Abdalkareem *et al.* [4, 5] use CI-skip rules to determine which build to skip. PreciseBuildSkip [41] also proposes CI-run rules to overrule skip rules for safety improvements. PreciseBuildSkip is not a suitable baseline for video-game repositories like Project X. Indeed, 61% of the build failures of Project X are due to changes made to data files only, which are not well-handled by the ruleset of PreciseBuildSkip. For example, according to the rule “NoSrcFileChange”, all data changes should be skipped. Developing such rules for the data files would be impractical because there are numerous types of data files, each requiring deep subject matter expertise to formulate effective and safe CI-skip and CI-run rules.

3.4 Chapter Summary

In this chapter, we introduce the prior work on dependency analytics that can potentially improve (1) code reviewer recommendation and (2) CI acceleration. Additionally, we outline the advantages and drawbacks of CI and presenting strategies, like build outcome prediction, to mitigate CI costs.

Indeed, dependency analytics can be leveraged to assist other software engineering researches. The next chapter will feature our analysis of build dependency graphs within a video game project.

Chapter 4

Quantifying Cross-Boundary Changes

An earlier version of the work in this chapter has been submitted to ICSE-SEIP 2024.

4.1 Introduction

Project of a multidisciplinary team should be represented by a multidisciplinary dependency graph. For example, a code change to the source file that reposition an object. 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 *intra-dependencies* within each domain, as well as *inter-dependencies* among all of the other digital assets involved. While dependency graphs have been explored in the general development context [7, 33, 84], the multidisciplinary context (of which 'AAA games' serve as an exemplar) introduces challenges in the extraction and analysis of dependency graphs that need to be addressed.

In this chapter, we present our quantitative analyses with respect to three research questions.

RQ1: What is missed by a typical code-only dependency graph?
Motivation: Non-code artifacts play an important role in video game development as they can also introduce build breakages and runtime issues. While it is clear that excluding these non-code artifacts will produce an incomplete graph, it is unclear the degree to which dependency analytics will be impacted. Therefore, to better understand their potential impact, we first set out to study the prevalence of non-code artifacts and their dependencies in the multidisciplinary dependency graph.
Results: In our context, non-code artifacts account for 97.2% of nodes, and edges connected to non-code artifacts account for 95.7% of edges. Hence, excluding non-code artifacts will likely have a large impact on dependency analytics.

RQ2: How often does the impact of a change cross disciplinary boundaries?
Motivation: In an interdisciplinary context, it is not unexpected for changes to cross disciplinary boundaries; however, the frequency of cross-boundary changes and their impact are not well understood. Thus, we set out to explore the prevalence and scope of cross-boundary changes to evaluate their importance in a multidisciplinary setting.
Results: In our context, cross-boundary changes occur at similar rates as changes that do not cross boundaries, but their impact is higher. On average, cross-boundary changes impact 212,104 nodes, with a median impact of 120,368 nodes. Statistical tests confirming the differences are significant (Mann-Whitney U test, $\alpha = 0.05$).

RQ3: How risky are cross-boundary changes?
Motivation: Our results from RQ2 show that cross-boundary changes typically impact a significantly larger number of nodes than changes that do not cross boundaries. This implies that cross-boundary changes may be riskier since they tend to have a more widespread effect on the system. To understand whether that risk manifests in concrete ways, we set out to study whether cross-boundary changes are more prone to introducing build breakages and defects than changes that do not cross-boundaries.
Results: Cross-boundary changes introduce build breakages (51% of the time) and defects (67% of the time) more frequently than changes that do not cross boundaries (44% and 37% of the time, respectively). Statistical tests confirm that both differences are statistically significant (Boschloo's test, $\alpha = 0.05$).

For each research question, we present our approach to addressing it and the results that we observe.

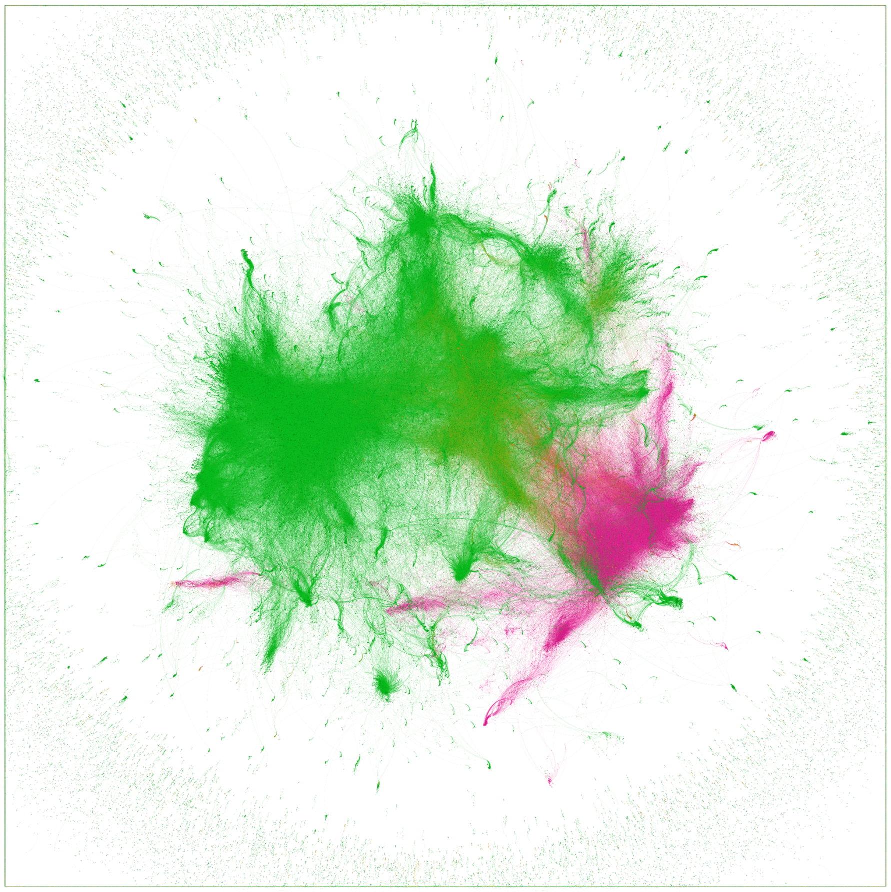

Figure 4.1: Example of a file-level multidisciplinary dependency graph, with green nodes representing data nodes, pink nodes representing code nodes, and orange nodes representing boundary nodes. Extracting a multidisciplinary dependency graph of this scale requires extensive knowledge of Project X.

4.2 RQ1: What is missed by a typical code-only dependency graph?

Approach: Using a dependency graph based on the most recent change set in our studied period as an example, we count the nodes and edges of different types in the graph. Though the numbers of nodes and edges fluctuate from change to change, we have confirmed with the project manager that the studied period is undergoing regular and steady development. Therefore, the graph is not as volatile as it would be in early development periods.

Table 4.1: Number of nodes and edges and their percentages in the multidisciplinary graph

	Type	Number	% of Total
Node	Code	30,675	2.8
	Non-Code	1,072,017	97.1
	Boundary	1,345	0.1
	Total	1,104,037	100%
Edge	(Code, Code)	141,412	4.30
	(Boundary, Code)	2,057	0.06
	(Non-Code, Boundary)	47,924	1.46
	(Non-Code, Non-Code)	3,097,819	94.18
	Total	3,289,212	100%

Results: Table 4.1 presents the numbers of the three types of nodes and the four types of edges in the graph. Figure 4.2 provides an overview of the graph from a visual perspective, where green nodes represent the non-code files, pink nodes represent the code files, and orange nodes represent the boundary nodes that connect code and non-code nodes.

Observation 1: Non-code artifacts are prevalent, constituting 97.2% of all nodes in the dependency graph, and the edges connecting to non-code nodes account for 95.7% of all edges. Among the 1,104,037 nodes in the dependency graph, only 30,675 nodes represent code files, while a substantial majority of 1,072,017 nodes represent non-code files. Additionally, 1,345 nodes facilitate the connection between code and non-code nodes across disciplinary boundaries (*a.k.a.* boundary nodes).

The dependency graph consists of a total of 3,289,212 edges. Among these edges, only 141,412 (4%) intra-connect code nodes to other code nodes, while the other 3,147,800 edges connect non-code or boundary nodes. The vast majority of edges (3,097,819 or 94%)

connect non-code nodes to other non-code nodes. Additionally, 2,057 edges connect code nodes to boundary nodes, and 47,924 edges connect boundary nodes to non-code nodes.

In our video game development setting, the vast majority of nodes (97.2%) and edges (95.7%) will be missed if non-code artifacts are excluded from the dependency graph.

Table 4.2: Build breakage rates and the number of impacted nodes of non-code and code changes.

	% of Total	Build Breakage (%)	Defect Inducing (%)	Cross Boundary	% of CB	Mean Impact	Median Impact	Build Breakage (%)	Defect Inducing (%)
Non-code	86	11	N/A	N/A	N/A	452	2	N/A	N/A
Code	14	46	50	Yes	44	212,104	120,368	51	67
				No	56	49	0[a]	41	37

[a] Counts of impacted nodes exclude the directly changed nodes.

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

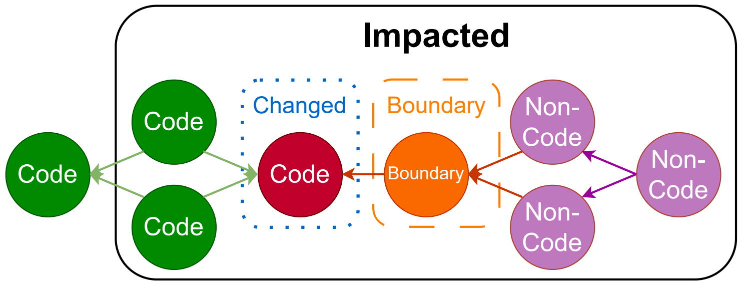

Figure 4.2: Example of a video game dependency graph, with the boundary node lie between code and data nodes.

Approach: We extract the dependency graph of changes on the main branch that span 11 weeks of development and locate the nodes that correspond to the list of committed files in each change. For each changed node, we traverse its incoming edges transitively to obtain a set of its dependent nodes in the graph. Then we compare the node type (*i.e.*, file type) of the changed node and the dependent nodes.

We categorize changes into two types: if a change adds, updates, or deletes any code files, we call it a code change, whereas changes that only modify non-code files are called non-code changes. We label changes as cross-boundary if any of the dependent nodes are of a different node type than the changed nodes, as shown in Figure 4.2.

Results: Table 4.2 compares the numbers of the dependent nodes associated with each change. Figure 4.3 plots the number of cross-boundary (CB) changes and code changes committed each day. The left y-axis shows the number of cross-boundary changes committed each day in blue, while the right y-axis shows the number of code changes committed each day. The x-axis indicates the progression of days. To enhance the clarity of the trend, we plot the trends using LOESS smoothing.

Observation 2: 44% of the code changes have a large cross-disciplinary impact that affects 212,104 nodes on average. We find that 6% of the analyzed commits have an impact that crosses disciplinary boundaries. While this is a small proportion of the overall commits, it accounts for 44% of the commits that change the source code. In addition, we find that none of the cross-boundary changes are non-code changes since the dependency flow is from non-code to code, *i.e.,* non-code nodes depend on code nodes. The comparison in Table 4.2 reveals that cross-boundary changes tend to affect a substantial number of nodes. Specifically, the mean number of impacted nodes for cross-boundary changes is 212,104, with a median of 120,368. In contrast, non-code changes and changes that do not cross boundaries have a substantially smaller impact, with means of 452 and 49 nodes, and medians of 2 and 0 nodes, respectively. These findings highlight the large difference in the scale of impact between cross-boundary changes and other types of changes.

The order of magnitude difference in the number of nodes impacted suggests that the multidisciplinary graph adds an important new perspective for analytics. The statistical significance of the discrepancies is confirmed by Mann-Whitney U tests (two-tailed, unpaired, $\alpha = 0.05$), yielding Holm-Bonferroni corrected p-values of 2.2×10^{-162} and 1.5×10^{-107}, respectively.

Observation 3: Cross-boundary changes are not isolated incidents, but rather frequently occur during the development process. In Figure 4.3, the LOESS smoothed curves representing the frequencies of committed cross-boundary changes and changes that do not cross boundaries closely align with each other. This alignment suggests that changes crossing disciplinary boundaries are a common occurrence that follows to the number of committed code changes that do not cross boundaries on a daily basis.

Changes crossing disciplinary boundaries are a common daily occurrence that represents 44% of code changes and 6% of overall changes, moreover, they impact a significantly larger amount of nodes compared to other changes.

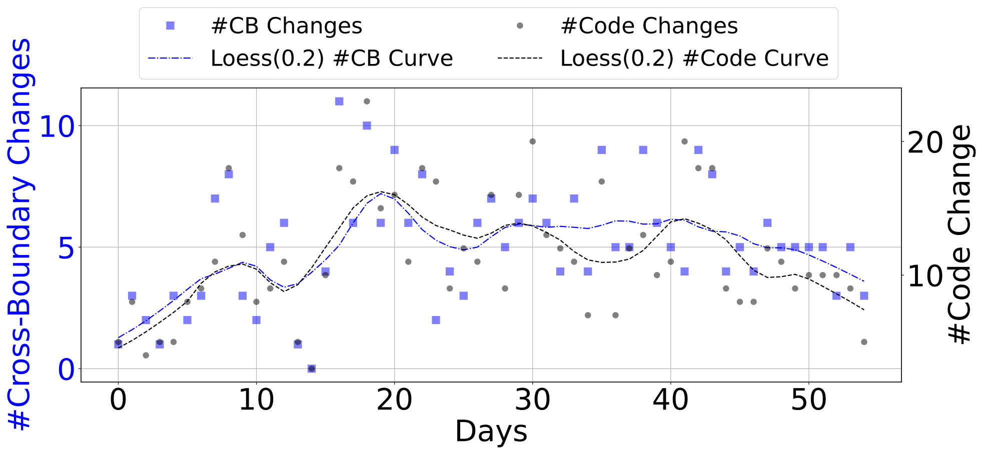

Figure 4.3: The daily occurrence of cross-boundary changes (blue y-axis on the left) and code changes that do not cross boundaries (black y-axis on the right).

4.4 RQ3: How risky are cross-boundary changes?

Approach: We study two types of risk: that of introducing build breakages and that of being implicated in defect-inducing commits. A build breakage occurs due to errors or test failures, while a defect-inducing commit occurs when a developer unintentionally introduces a defect or design flaw while making changes to the codebase, which is later discovered and fixed by another commit. Since non-code changes do not cross the disciplinary boundaries, we focus our comparison on cross-boundary changes and code changes that do not have an impact on non-code nodes (*i.e.,* code changes that do not cross boundaries). For build failures, we mine the build logs to extract build results for each change. As for defect-inducing commits, we rely on information from the CLEVER database [55] at Ubisoft, which identifies the defect-inducing commits with the SZZ algorithm presented by Kim *et al.* [42]. To assess the significance of the observed differences, we apply Boschloo's Exact test on the build-breaking and fix-inducing rates of cross-boundary changes and changes that do not cross boundaries.

Results: Table 4.2 shows the rates of build failures and defect introduction in cross-boundary changes and code changes that do not cross boundaries. While the table also reports build failure rates in non-code changes, the defect introduction rates of non-code changes cannot be computed since the SZZ algorithm cannot be applied to non-code changes.

Observation 4: Cross-boundary changes introduce build breakages at a greater rate than changes that do not cross boundaries (10 percentage points or 24% more). 51% of cross-boundary changes introduce build breakages, whereas 41% of code changes that do not cross boundaries and 11% of non-code changes introduce build breakages. The statistical significance of this discrepancy is confirmed by Boschloo's Exact test, both yielding a p-value smaller than 0.01.

Observation 5: Cross-boundary changes introduce defects at a greater rate than changes that do not cross boundaries (30 percentage points or 81%). Table 4.2 shows that 67% of cross-boundary changes and 37% of changes that do not cross boundaries introduce defects. The statistical significance of this difference is confirmed by Boschloo's Exact test, yielding a p-value of 8.0×10^{-15}.

The notable disparity observed in the build breakage and defect introduction rates indicates that cross-boundary changes have a greater tendency to break the build pipeline and introduce defects. We suspect that this increased risk can be attributed to the fact that cross-boundary changes impact a larger number of interconnected nodes within the dependency graph, exposing a larger proportion of the system to change, which in turn increases their riskiness.

Cross-boundary changes tend to break builds and induce defects with significantly greater rates than changes that do not cross boundaries.

4.5 Chapter Summary

In this chapter, we present the quantitative analysis of cross-boundary changes, emphasizing the need for a multidisciplinary dependency graph. We demonstrate that (1) code nodes represent less than 3% of the graph, (2) cross-boundary changes occur regularly with a large-scale impact on the development process, and (3) these changes are risky in terms of build breakages and inducing defects.

Chapter 5

Characterizing Cross-Boundary Changes

An earlier version of the work in this chapter has been submitted to ICSE-SEIP 2024.

5.1 Introduction

Acknowledging the common occurrence and associated risks of cross-boundary changes in video game development, it it crucial to understand the context in which these changes are implemented. For instance, repositioning an object is more likely to have cross-boundary impacts than implementing a helper function that performs arithmetic operations. Understanding the common context in which these changes are implemented helps with proposing treatments to mitigate potential risks.

In this chapter, we present our characterization of cross-boundary changes with respect to two research questions.

RQ4: Which activities are performed during cross-boundary changes?
Motivation: To explore why cross-boundary changes tend to be riskier, we set out to characterize the activities that are performed when changes cross disciplinary boundaries. Understanding the nature of these activities can provide valuable insights into the potential

effects and consequences of these changes, allowing stakeholders to take appropriate measures to address any associated risks more effectively.
Results: 43%, 68%, and 26% of the tagged cross-boundary changes are associated with gameplay functionality, feature additions, and file additions, respectively, whereas 36% and 68% of the tagged changes that do not cross boundaries are associated with tools and their configuration, as well as bug fixes. Non-code changes are rarely tagged.

RQ5: Who produces cross-boundary changes?
Motivation: Having gained an understanding of cross-boundary changes as a common and risky type of change in multidisciplinary development, our focus shifts to identifying the teams responsible for cross-boundary changes. We are particularly interested in determining whether contributors of cross-boundary changes are concentrated within a single team or dispersed across multiple teams. Identifying these contributor tendencies can help management to better account for the potential risks associated with cross-boundary changes through, *e.g.,* raising awareness of inter-team impact during code review.
Results: We find that 74% of the contributors of cross-boundary changes are members of a single team. Indeed, cross-boundary changes within the studied game tend to be concentrated rather than dispersed.

For each research question, we present our approach to addressing it and the results that we observe.

5.2 RQ4: Which activities are performed during cross-boundary changes?

Approach: When a commit is focused on a specific topic, it is common for the contributors at Ubisoft to apply tags to those commits. These tags help to organize the related commits, making it easier to track changes and understand the collective purpose behind them.

To address our research question, we focus on the `@Category` and `@Type` tags. The `@Category` tag describes the aspect of the game being modified (*e.g.,* GamePlay, Tools), whereas the `@Type` tag describes the development action being taken (*e.g.,* feature, bugfix). Contributors may apply multiple tags to their changes, therefore, the tags are not mutually exclusive. These tags are not a mandatory component of the commit, and hence, are unspecified for the bulk of the commits within our studied period.

Table 5.1: Percentage of changes associated with a tag or a file action.

(%)	Tag/ Action	Code CB	Code Non-CB	Non-Code
@Category	GamePlay	43	21^{***}	28
	Tools	13	36^{***}	28
	AI	28	27	22
	UI	13	5^{*}	25
	Scripts	4	3	0
	Graphics	1	5	0
	Servers	3	9	0
	DevOps	0	2	0
	Tagged	38	33	1^{***}
@Type	Feature	68	22^{***}	56
	Bugfix	24	68^{***}	37
	Refactor	7	6	0
	Test	1	4	0
	Nodes	5	2	0
	BuildSystem	0	4	4
	Doc	0	0	7
	Tagged	30	30	1^{***}
File Action	Addition	26	11^{***}	36^{***}
	Update	97	89^{***}	81^{***}
	Deletion	8	5	16^{***}

Tags and actions are not mutually exclusive.
Statistical significance of Boschloo's Exact test:
$^{*}p < 0.05;^{**}p < 0.01;^{***}p < 0.001; otherwise\ p \geq 0.05$

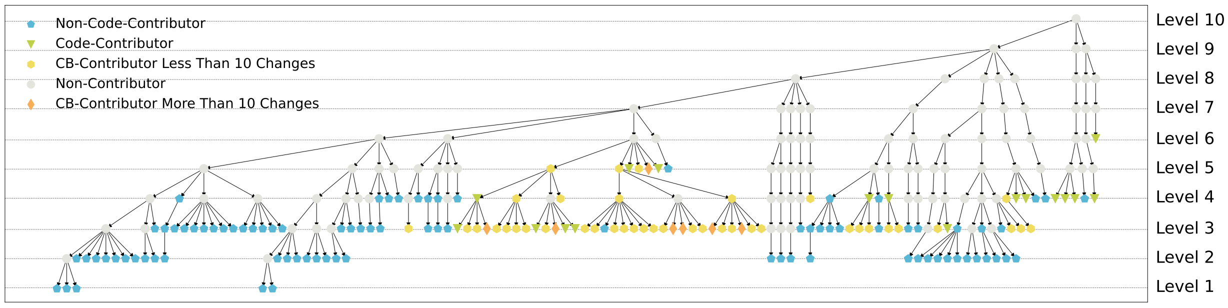

Figure 5.1: Organization graph of the contributors that committed at least one change during the studied period.

In addition to tags, we also compare the file actions (*i.e.*, add/ update/ delete) among cross-boundary changes, code changes that do not cross boundaries, and non-code changes. To assess the significance of the observed differences between cross-boundary changes and others, we apply Boschloo's Exact test to the collected statistics.

Results: Table 5.1 shows the rates of occurrence and the statistical test results for the studied tags and action types.

Observation 6: GAMEPLAY, TOOLS, *and* UI *categories interact with whether or not changes cross boundaries.* Table 5.1 shows that 38% of cross-boundary changes and 33% of code changes that do not cross boundaries are tagged with categories. Although there is a difference of five percentage points, Boschloo's test indicates that the null hypothesis (*i.e.*, that the rate of tagging is not significantly different between cross-boundary changes and code changes that do not cross boundaries) cannot be rejected. The GAMEPLAY, TOOLS, and UI `@Category` tags show a significant disparity between cross-boundary changes and code changes that do not cross boundaries. Indeed, cross-boundary changes are more likely to be associated with gameplay and UI. Table 5.1 shows that there is a statistically significant 22 and 8 percentage point increase being observed between cross-boundary changes and code changes that do not cross boundaries. On the other hand, cross-boundary changes are less likely to be associated with TOOLS. Table 5.1 shows that there is a statistically significant 23 percentage point decrease being observed between cross-boundary changes and code changes that do not cross boundaries. Finally, we observe that only 1% of non-code changes are tagged with a `@Category`, hence we refrain from comparing their tag distribution with the one of code changes.

Observation 7: Feature additions are more often associated with cross-boundary changes, whereas bug fixes are more often associated with code changes that do not cross boundaries. Table 5.1 shows that 30% of code changes are tagged with types. There is no rate difference between cross- and non-cross-boundary code changes, and Boschloo's test indicates that the null hypothesis (*a.k.a.* that the rate of tagging is not significantly different between cross-boundary and code changes that do not cross boundaries) cannot be rejected. The FEATURE and BUGFIX `@Type` tags show substantial disparities in rates across cross-boundary and non-cross-boundary changes. Indeed, we observe that FEATURE tags are more prevalent among cross-boundary changes, with a statistically significant 46 percentage point increase in the rates compared to code changes that do not cross boundaries. On the other hand, we observe that BUGFIX [1] tags are more prevalent among code changes that do not cross boundaries. Table 5.1 shows that there is a statistically significant 44 percentage point decrease in the rates of BUGFIX tags among cross-boundary changes compared to code changes not crossing boundaries. Finally, similar to the `@Category` tag, we observe that non-code changes are rarely tagged with a `@Type`.

Observation 8: File additions and updates are more prevalent among cross-boundary changes than code changes that do not cross boundaries. Table 5.1 shows that 26% and 97% of cross-boundary changes include file additions and updates, while 11% and 89% of code changes that do not cross boundaries include file additions and updates, respectively. There is a statistically significant 15 and 8 percentage point difference in rates of file additions and file updates among cross-boundary and non-cross-boundary code changes. Although there is a 3 percentage point difference in the rates of file deletions between the two code change categories, Table 5.1 shows the difference is statistically insignificant.

Observation 9: File additions and deletions are more prevalent among non-code changes than cross-boundary changes, whereas file updates are less prevalent among non-code changes than cross-boundary changes changes. Table 5.1 shows that 36% and 16% of non-code changes include file additions and deletions, while 26% and 8% of cross-boundary changes include file additions and deletions, respectively. In terms of file updates, 81% of non-code changes include file updates, while 97% of cross-boundary changes include file updates. There are statistically significant differences in rates of file additions, deletions, and updates among non-code changes and cross-boundary changes.

Cross-boundary changes are more often associated with GAMEPLAY, FEATURE, *and* UI *tags, as well as file additions and updates, whereas code changes that do not cross boundaries are more often associated with* TOOLS *and* BUGFIX *tag. Non-code changes are rarely tagged.*

[1]The defect-fix pairs are many-to-many relationships [18, 42, 68].

5.3 RQ5: Who produces cross-boundary changes?

Approach: In collaboration with the human resources team, we extract the organizational graph of contributors who committed at least once during the study period. Starting from the contributors, we traverse the graph upwards to their direct managers and continue transitively until all contributors share at least one common manager. The levels are defined from the bottom of the tree structure to the top. In this context, the leaf node is considered level 1, while the root node is placed at level 10. Nodes from the same sub-tree (*i.e.,* those who share a manager below level 10) are considered to be members of the same team.

It is important to note that the abstracted levels mentioned in this paper do not directly correspond to the actual employee levels within Ubisoft. To protect the privacy of contributors, their personal data are no longer recorded once they lose access to the project. Consequently, in this study, the data of contributors who have left the project are omitted. Further details regarding the potential impact of this omission on the validity of our conclusions will be discussed in Section 5.5.

Results: Figure 5.1 presents the complete graph, with the gray circle-shaped nodes representing the direct managers. The green triangle-shaped nodes represent the contributors who committed at least one code change. The yellow hexagon-shaped nodes represent the contributors who committed more than at least one cross-boundary change. The orange diamond-shaped nodes represent contributors who committed more than ten cross-boundary changes. Finally, the blue pentagon-shaped nodes represent the contributors who committed non-code changes.

Observation 10: Non-code changes are committed by individuals from various job groups, while code changes, specifically, cross-boundary changes, are exclusively committed by programmers. The contributors involved in the study are dispersed across six distinct job groups: IT, Programming, Quality Management, Animation, Audio, Game and Level Design, and Art. Contributors from the Quality Management, Animation, Audio, Game, and Level Design, and Art groups exclusively contribute to non-code changes, as represented by the blue pentagon in Figure 5.1. On the other hand, contributors from the IT and Programming groups have a more diverse range of contributions, including both non-code and code changes. It is important to note that during the studied period, the IT group contributed a negligible fraction (0.4%) of all changes, and only the Programming group contributes cross-boundary changes, as represented by the yellow hexagons and orange diamonds in Figure 5.1.

The contributors who have committed cross-boundary changes, namely cross-boundary contributors (*i.e.,* CB-contributor in Figure 5.1), are a subset of the contributors who have committed at least one code change, namely code contributors. 73% of code contributors are also cross-boundary contributors.

Observation 11: 74% of cross-boundary contributors originate from the same team. Figure 5.1 shows that 35 of the 47 cross-boundary contributors are from the same sub-tree at level 6. Additionally, 82% of cross-boundary changes are from this level 6 team. The other 12 cross-boundary contributors who produced 18% of cross-boundary changes can be traced back to five sub-trees at level 6.

Cross-boundary changes are from the Programming group only, while non-code changes have various group origins. Contributors from one team commit most of the cross-boundary changes.

5.4 Practical Implications

In this section, we discuss the potential benefits of dependency analysis in the broader software engineering context.

(1) Dependency graphs should be expanded to include the multidisciplinary context. Observation 1 shows that code files only represent a small fraction of dependency graphs that we extract. Thus, graphs that omit non-code artifacts will provide a narrow and skewed perspective when dependency analytics are performed. Observations 2 and 3 show that code changes frequently impact non-code artifacts (*i.e.,* cross-boundary changes), indicating non-code files are interacting with code files even when they are not modified. Moreover, non-code changes also incur build breakages. Therefore, non-code files should not be left out when analyzing dependency graphs. While the context of AAA game development, is somewhat unique, we suspect that this is not the only context where multiple disciplines interact to produce software systems (*e.g.,* operating systems with multimedia components [84], vehicle software systems with hardware involvements). Taking a multidisciplinary perspective on dependency analytics in such contexts will likely also increase the value of the results.

(2) Cross-boundary changes should receive additional scrutiny during the code review process. Observation 4 shows that changes that cross boundaries are more likely to break builds than changes that do not. Furthermore, Observation

5 shows that changes that cross boundaries are more likely to be implicated in future defect-inducing commits than changes that do not. Therefore, to mitigate these potential risks, we believe that additional QA rigour should be applied during the integration process of cross-boundary changes. The natural first step in the software process at Ubisoft is to flag these changes during the code review process.

5.5 Threats to Validity

Below, we present the threats to the validity of our study.

5.5.1 Construct Validity

The process of extracting code dependencies at the file level involves reading the content of source files and identifying import statements. However, the import statements may be located within conditional statements, *e.g.,* when platform-dependent libraries are loaded to accelerate graphics on PC or console platforms. As a result, our liberal approach may introduce additional nodes and edges to the graph. while we are developing improvements to address these overestimates, we find that they are relatively rare in our practical setting. Moreover, the overestimates do not impact the fundamental concepts that we present in this paper.

5.5.2 Internal Validity

Since the graphs generated for each revision are large enough to present practical traversal challenges, we have restricted our study to revisions generated within an 11-week period. Also, the graph example that we use to count nodes and edges is only one slice of the development process. While there could be alternative explanations for the observations presented in this paper, such as the occurrence of a maintenance period or frozen development branches, our communication with project managers indicates that our studied period aligns with regular and steady development.

In addition, due to privacy concerns, we omit cross-boundary contributors who no longer have commit access from our analysis. While this omission may affect the observed percentages in Section 5.3, it is important to note that the majority of the cross-boundary changes originate from a team that shares a common manager six levels up in the organization chart. Therefore, we suspect the impact of this omission on our conclusions is minimal.

5.5.3 External Validity

The results for the research questions are derived from a AAA game at Ubisoft. This limited scope may impact the generalizability of the study's findings to a broader context. It does not affect the concept of the multidisciplinary dependency graphs and its implications for future research and practical tools. Nonetheless, replication studies may strengthen the generalizations that can be drawn.

5.6 Chapter Summary

In Chapter 4 and 5, we demonstrate the importance of multidisciplinary dependency graphs. While our observations are drawn from the context of the development of a AAA game at Ubisoft, we conjecture that similar observations may hold in other domains that involve multidisciplinary teams. We observe that changes having an impact that crosses disciplinary boundaries are a regular occurrence and tend to be associated with a greater risk of build breakage and defect introduction than changes that do not cross boundaries. Moreover, we show that cross-boundary changes are more commonly associated with gameplay functionality, feature additions, and UI-related development than changes that do not cross boundaries, which are more commonly associated with tool implementation and bug fixes than cross-boundary changes. Finally, the vast majority of cross-boundary changes are contributed by a single team rather than dispersed across the organization.

These findings have laid the foundation for improvements to dependency analytics at Ubisoft. We are actively developing solutions that leverage the multidisciplinary dependency graph to improve code reviewer recommendations and accelerate CI processes.

Chapter 6

Context, Relevance, and Dependency-Aware Build Outcome Prediction: RavenBuild

An earlier version of the work in this chapter has been submitted to FSE 2024.

6.1 Introduction

Continuous Integration (CI) [19] enables rapid feedback by invoking build and test jobs for the change sets that development teams produce. For large organizations, adopting CI can accelerate development [72], but it is not without costs. The execution of a slow CI job can delay time-to-feedback [25], forcing developers to perform a mentally taxing context switch [85] to another task or remain effectively idle. Moreover, CI execution consumes a large quantity of computational resources [32, 34], which can quickly accrue annual costs on the order of millions of dollars [35].

Ubisoft heavily invests in the execution of builds for its games. For example, for Project X (the latest installment in a ten-year video game franchise), thousands of build jobs are submitted daily, consuming an average of 49 build hours on the main branch. The build duration for Project X ranges between 10 to 30 minutes, delaying developers from receiving immediate feedback and incurring expensive computational costs.

Although early results show plenty of promise, the distinct characteristics of Project X present new challenges for build outcome prediction. Prior work on build outcome prediction has largely focused on open-source projects that are code-intensive. As such, many features adopted by these studies are code-specific, *e.g.,* the number of lines changed in source code. In the Project X setting, non-code artifacts and changes therein are more prevalent than source code and code changes. These non-code artifacts play a crucial part in the game experience. The game engine compiles the non-code artifacts with source code in an order-sensitive manner that respects the specified dependencies. Therefore, if non-code artifacts are corrupted, or dependencies are not respected, non-code changes will incur build failures.

6.2 RavenBuild On Project X

In this section, we present a set of new features designed for build outcome prediction on Project X, and an evaluation of the performance of the state-of-the-art BuildFast [13] model on Project X before and after adopting our features.

6.2.1 Approach

Table 6.1 shows the build outcome prediction features adopted by BuildFast. In addition to adopting BuildFast features, we engineer features that are file type-agnostic to better accommodate the context of video game development into build outcome prediction. Table 6.2 provides an overview of RavenBuild features, which characterize three aspects of a change set.

Context-Aware Features Change sets are typically submitted with the intention of integrating new features or bug fixes. This context transitively applies to the builds that are invoked to test a change set. We infer the context of a build from the metadata of its change set, such as commit messages, file types, and prior build activity. We classify change sets according to five intention categories using the keyword list that was originally proposed by Hindle *et al.* [36] and updated by Nayrolles and Hamou-Lhadj [55]. In addition to the intention of changes, we classify the studied changes based on the types of files being modified. Change sets can be code-only, data-only, or a mix of both. Finally, we implement a feature to detect if any of the commits in the change set being built have been built before.

Table 6.1: Build outcome features adopted by BuildFast.

BuildFast Features						
Current Build	C_1	src_churn	C_2	test_churn	C_3	src_ast_diff
	C_4	test_ast_diff	C_5	line_added	C_6	line_deleted
	C_7	files_added	C_8	files_deleted	C_9	files_modified
	C_{10}	src_files	C_{11}	test_files	C_{12}	config_files
	C_{13}	doc_files	C_{14}	class_changed	C_{15}	met_sig_modified
	C_{16}	met_body_modified	C_{17}	met_changed	C_{18}	field_changed
	C_{19}	import_changed	C_{20}	class_modified	C_{21}	class_added
	C_{22}	class_deleted	C_{23}	met_added	C_{24}	met_deleted
	C_{25}	field_modified	C_{26}	field_added	C_{27}	field_deleted
	C_{28}	import_added	C_{29}	import_deleted	C_{30}	commits
	C_{31}	fix_commits	C_{32}	merger_commits	C_{33}	committers
	C_{34}	by_core_member	C_{35}	is_master	C_{36}	time_interval
	C_{37}	day_of_week	C_{38}	time_of_day		
Previous Build	P_1	pr_state	P_2	pr_compile_error	P_3	pr_test_exception
	P_4	pr_tests_ok	P_5	pr_tests_fail	P_6	pr_duration
	P_7	pr_src_churn	P_8	pr_test_churn		
Historical Builds	H_1	fail_ratio_pr	H_2	fail_ratio_pr_inc	H_3	fail_ratio_re
	H_4	fail_ratio_com_pr	H_5	fail_ratio_com_re	H_6	last_fail_gap
	H_7	consec_fail_max	H_8	consec_fail_avg	H_9	consec_fail_sum
	H_{10}	commits_on_files	H_{11}	file_fail_prob_max	H_{12}	file_fail_prob_avg
	H_{13}	file_fail_prob_sum	H_{14}	pr_src_files	H_{15}	pr_src_files_in
	H_{16}	pr_test_files	H_{17}	pr_test_files_in	H_{18}	pr_config_files
	H_{19}	pr_config_files_in	H_{20}	pr_doc_files	H_{21}	pr_doc_files_in
	H_{22}	log_src_files	H_{23}	log_src_files_in	H_{24}	log_test_files
	H_{25}	log_test_files_in	H_{26}	team_size		

For example, in the Project X setting, builds can go through a preflight build before it is merged. In open-source settings, it is also possible for a commit to be built during the review of a pull request, and that same commit will be built again when the PR is merged.

Relevance-Aware Features In the development process, a set of builds may relate to a single task. For example, when an initial build error occurs and subsequent failing builds are attempts to fix it; or when a large task is decomposed into a series of incremental change sets. Therefore, the features that exploit the status or context of a previous build should be constrained to the context when the prior and current builds are part of the same task. Thus, we propose features to suggest to the model when the previous build features are relevant for the current prediction, and when they should be ignored.

We compare the current build to its immediate predecessor to extract the relevance-aware features that are shown in Table 6.2. R_1, R_2, and R_4 are binary features, while R_3 is computed as $len(set(current\ actions) - set(previous\ action))$. ($R_5$): Path relevance measures the similarity between the modified directories of the previous and current builds. A pair of builds that modify files in the same location is more likely to build on inter-dependent targets than a pair of builds changing files in completely different locations. The path length is measured by counting the sub-directories in that common path. We then compute the relevance score as the length of the longest common path between the file paths of the previous (LCM_{prev}) and current builds (LCM_{cur}) divided by the sum of $len(LCM_{prev})$ and $len(LCM_{prev})$.

$$Path\ Relevance\ Score = \frac{len(LCM_{prev,cur})}{len(LCM_{prev}) + len(LCM_{cur})}$$

We compute the *sum* instead of the *max* because the longest common path of the build also provides a perspective on the dispersion across the changed files. A shorter *LCM* suggests that changed files are spread across multiple sub-directories, while a longer *LCM* suggests that the changed files are located in the same sub-directory. We want to detect whether a low path relevance score is due to the broad dispersion of changed files, or due to changed files being from completely different sub-directories, *e.g.,* consider two cases where in the first case $len(LCM_{prev}) = 3$, $len(LCM_{cur}) = 4$, and $len(LCM_{prev,cur}) = 1$, whereas in the second case $len(LCM_{prev}) = 3$, $len(LCM_{cur}) = 1$, and $len(LCM_{prev,cur}) = 1$. We can differentiate the two cases using *sum*, while using *max*, the path relevance score for both cases would be the same.

Dependency-Aware Features Modifying files that are depended upon by a large number of files is riskier than modifying a file that has no dependents. For example, changing a popular library function will propagate that change to its many use points, and is inherently more risky than changing a short script that calls a function from the library.

Table 6.2: RavenBuild Features

	ID	Name	Description
Context	T_1	Is Corrective	If the commit message has 'fix', 'bug', 'wrong', 'fail', and 'problem'
	T_2	Is Additive	If the commit message has 'new', 'add', 'requirement', 'initial', and 'create'
	T_3	Is Non-Functional	If the commit message has 'doc' and 'merge'
	T_4	Is Perfective	If the commit message has 'clean' and 'better'
	T_5	Is Preventive	If the commit message has 'test', 'junit', 'coverage', and 'assert'
	T_6	Built-Before	If any of the commits in the current build have been built before (preflight)
	T_7	Change Type	The modified files are source files only, non-code files only, or a mix of both.
Relevance	R_1	Committer Changed	The pair of builds are submitted by different developers.
	R_2	Type Changed	The pair of builds have different change types (*i.e.,* T_7).
	R_3	Action Changed	The pair of builds have different actions.
	R_4	Is Same Intention	The pair of builds are of the same intention (*i.e.,* $T_1 - T_5$)
	R_5	Path Relevance	The similarity between the modified directories of the pair of builds
Dependency	D_1	Is Cross Boundary	Is a cross-boundary change
	D_2	#Impacted	Number of nodes impacted by the changed nodes
	D_3	#Impacted Non-Code	Number of non-code nodes impacted by the changed nodes
	D_4	#Impacted Code	Number of code nodes impacted by the changed nodes
	D_5	#Impacted Boundary	Number of boundary nodes impacted by the changed nodes

Therefore, we propose a set of features that we extract from build dependency graphs as shown in Table 6.2. Our multidisciplinary approach to dependency graphs enables us to compute dependency-aware features on all types of files. We detect if the impact of a change set crosses the code-data boundary (*i.e.,* D_1), the total number of nodes impacted (*i.e.,* D_2), and the number of data/code/boundary nodes impacted (*i.e.,* $D_3 - D_4$). In the context of Project X, since non-code files also have dependencies, examining the set of impacted nodes provides an additional perspective on the riskiness of the change sets, which is not captured by any of the prior features.

Model Description To isolate the contribution of the new features, we adopt the original BuildFast model architecture and its hyper-parameters. BuildFast [13] uses an adaptive prediction mechanism that consists of two XGBoost [14] models: (1) one is trained on the previously passed builds and currently failing builds, and (2) the other is trained on the previously failing builds and currently failing builds. During the training phase, the previous-passing model adopts Information Gain [44] as the feature selection method to select the top 25 features, whereas the previous-failing model adopts Chi-Square Testing [28] to select the top 30 features. At inference time, BuildFast predicts the build outcomes based on their immediate predecessor's outcome, *i.e.,* if the previous build failed, BuildFast predicts the current build outcome using the previous-failing model, while if the previous build passed, BuildFast predicts the current build outcome using the previous-passing model.

Table 6.3: Performances of BuildFast, RavenBuild, and the improvements on Project X

	BuildFast	RavenBuild	Improvement
F1-fail	.430	.628	+.198
F1-pass	.961	.968	+.007
Recall-fail	.319	.563	+.244
Recall-pass	.984	.979	-.005
Precision-fail	.662	.710	+.048
Precision-pass	.938	.959	+.021
AUC	.687	.881	+.194
Benefit	298.4	286.6	-11.8
Cost	81.9	46.5	+35.4
Gain	216.5	240.2	+23.7

Model Evaluation To respect the chronological order of builds, the most recent one-third of our dataset is held out as the testing set, with the prior two-thirds being used for training. Table 6.3 shows the performance of BuildFast and RavenBuild, and the difference between the two in F1-fail, F1-pass, recall-fail, recall-pass, precision-fail, precision-pass, and AUC when applied to Project X. We also compute the benefit, cost, and gain metrics that were proposed by Chen *et al.* [13]. Benefit is the number of build hours that would have been saved by skipping the correctly predicted passing builds. Cost is the number of unnecessary build hours that would have been spent due to incorrectly predicted failing builds. Gain is the difference between benefit and cost, *i.e., Benefit − Cost*.

6.2.2 Results

Observation 1: RavenBuild improves BuildFast on Project X in F1-score, recall, precision, and AUC, with a negligible decrease in recall-pass. Table 6.3 shows that after adopting the context-aware, relevance-aware, and dependency-aware features, RavenBuild improves BuildFast in all failing classes, by 19.8 percentage points in F1-fail, 24.4 percentage points in recall-fail, and 4.8 percentage points in precision-fail. Though RavenBuild also incurs a 0.5 percentage point penalty in recall-pass, RavenBuild improves BuildFast by 0.7 percentage points in F1-pass, 2.1 percentage points in precision-pass, and 19.4 percentage points in AUC.

Observation 2: Although RavenBuild does not save as much as BuildFast, it makes fewer mistakes when recommending builds, leading to a better over-

all gain score. Table 6.3 shows that RavenBuild accrues 11.8 fewer build hours of savings when compared to BuildFast. We suspect that this is due to the fact that RavenBuild has a slightly lower recall-pass score. On the other hand, RavenBuild reduces costs by 35.4 build hours by more accurately predicting the failing class. The overall perspective shows that RavenBuild improves the net gain of build outcome prediction in the Project X setting by 23.7 build hours.

RavenBuild outperforms the state-of-the-art approach to build outcome prediction in the Project X setting by a substantial amount.

6.3 RavenBuild-CR on Open-Source and Project X

While dependency-aware features are promising a new direction for build outcome prediction, the dependency graphs from which they are extracted are expensive to construct and require project-specific knowledge to be reliably produced. Therefore, we present an alternative RavenBuild-CR that employs the context-aware and relevance-aware features and excludes dependency-aware features. In this section, we evaluate the performance of the simplified RavenBuild-CR model on Project X. In addition, since the need for deep knowledge of dependency structures has been relaxed, we also evaluate RavenBuild-CR on a sample of active open-source projects.

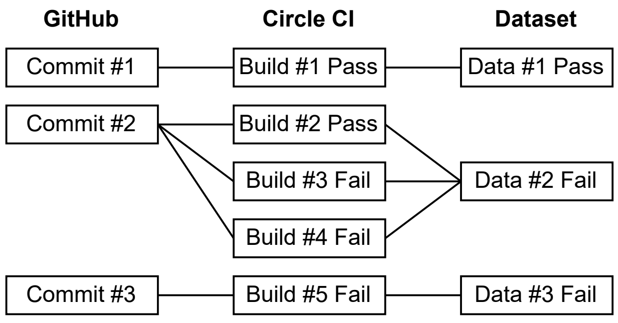

Figure 6.1: Builds triggered on the same commit ID are merged into one data point.

6.3.1 Approach

Studied Open-Source Projects Gallaba *et al.* [25] provide a large dataset that contains 23,330,690 builds that were performed using Circle CI [16], spanning 7,795 projects that are implemented in 40 different languages. To be consistent with BuildFast, we exclude all repositories for which Java is not the primary language (*i.e.,* GitHub uses the Linguist library [1] to determine the primary language). Then, for each such project, we collect its build data from Circle CI. Since Circle CI does not retain data about all of the past builds, we filter out the repositories that have fewer than 100 builds available at the time of our analysis, resulting in a dataset with 45,247 builds from 22 projects.

Circle CI is a versatile platform. As such, the studied projects configure the platform in various ways to meet their needs. For example, we observed that some projects trigger multiple builds with different parameter settings on the same change set to, *e.g.,* execute test cases in parallel. Since many features are extracted at the change set level, we avoid duplicate entries in our dataset by merging builds that are triggered on the same commit ID as shown in Figure 6.1. We consider that the merged entry has a passing build outcome if and only if all of the merged entries have passing outcomes. The duration of the merged build is calculated by summing up the durations of each of the merged entries. Finally, to ensure that previous build features can be computed, we exclude the first build of each project, since it will not have a predecessor from which the features can be extracted. Table 6.4 provides an overview of the 22 studied open-source projects and Project X. To foster future replication efforts, we make our data collection scripts and raw data available online (see Section 6.6).

6.3.2 Results

Observation 3: With only the context-aware and relevance-aware features, RavenBuild-CR still outperforms BuildFast on Project X. Without the dependency-aware features, Table 6.5 shows that RavenBuild-CR still outperforms BuildFast in the Project X setting by 11.2 percentage points in F1-fail, 11.1 percentage points in recall-fail, 7.2 percentage points in precision-fail, and 12.9 percentage points in AUC. RavenBuild-CR also improves slightly on the performance in the passing class as well.

In terms of time-to-feedback, RavenBuild-CR leads to 20.9 hours less in Cost, with a negligible 0.2 hours reduction in Benefit, resulting in 20.7 more hours of Gain than BuildFast. Though RavenBuild-CR improves both precision and recall on Project X, the

[1]https://github.com/github-linguist/linguist/

Table 6.4: Open-Source projects and Project X Statistics

ID	Project	Duration[a]	Pass	Fail	ID	Project Name	Duration[a]	Pass	Fail
1	atlasdb	37	2,135	1,021	12	interlok	7	120	29
2	auth0-java	2.3	328	29	13	java-jwt	3	274	55
3	cassandra	117	313	180	14	micrometer	18	322	76
4	cbioportal	29	58	217	15	opennms [b]	1	244	37
5	client_java	4	87	18	16	react-native-share	12	24	41
6	conjure-java-runtime	7	354	83	17	rskj	18	844	198
7	docker-compose-rule	4	105	131	18	spring-cloud-aws	5	419	265
8	fresco	11	79	559	19	spring-cloud-security	2	113	366
9	gradle-baseline	8	378	111	20	Strata	16	136	27
10	gradle-git-version	6	96	37	21	styx	9	109	81
11	grakn	20	106	13	22	testcontainers-java	2	1018	16
X	Project X	21	3,898	743					

[a]in minutes. [b]opennms-provisioning-integration-server.

improvements in the passing class are relatively small. Since the benefit is calculated by summing up the execution time of the correctly predicted passing builds, we suspect the minor decrease in benefit is due to the build duration of the correctly predicted passing builds of RavenBuild-CR being slightly shorter than the passing builds that are correctly predicted by BuildFast.

Observation 4: RavenBuild-CR outperforms BuildFast on all metrics in the open-source setting in all metrics. Table 6.5 also shows that RavenBuild-CR outperforms BuildFast in the open-source setting by 2.1 percentage points in F1-fail, 1.8 percentage points in F1-pass, 1.9 percentage points in recall-fail, 0.3 percentage points in recall-pass, 2.9 percentage points in precision-fail, 5.8 percentage points in precision-pass, and 1.9 percentage points in AUC. In terms of time-to-feedback, RavenBuild-CR saves a total of 13.9 hours more than BuildFast. As RavenBuild-CR also incurs 19.2 hours less in Cost than BuildFast, the net gain is improved by 33.1 hours.

Without dependency-aware features, although a negligible reduction in Benefit was observed, RavenBuild-CR still outperforms the state of the art in the Project X setting. Moreover, RavenBuild-CR is general enough to apply to the open-source setting, where across-the-board improvements with respect to the state of the art were observed.

Table 6.5: Performances of BuildFast and RavenBuild-CR on Open-Source projects and Project X.

	Open-Source			Project X		
	BuildFast	RavenBuild-CR	Improvement	BuildFast	RavenBuild-CR	Improvement
F1 fail	.447	.468	+.021	.43	.542	+.112
F1 pass	.753	.771	+.018	.961	.966	+.005
Recall fail	.552	.571	+.019	.319	.43	+.111
Recall pass	.745	.748	+.003	.984	.985	+.001
Precision fail	.439	.468	+.029	.662	.734	+.072
Precision pass	.793	.851	+.058	.938	.948	+.01
AUC	.717	.736	+.019	.687	.816	+.129
Benefit	229.4	243.3	+13.9	298.4	298.2	-.2
Cost	138.1	118.9	+19.2	81.9	61	+20.9
Gain	91.3	124.4	+33.1	216.5	237.2	+20.7

6.4 An Introspective Look at the State of Build Outcome Prediction

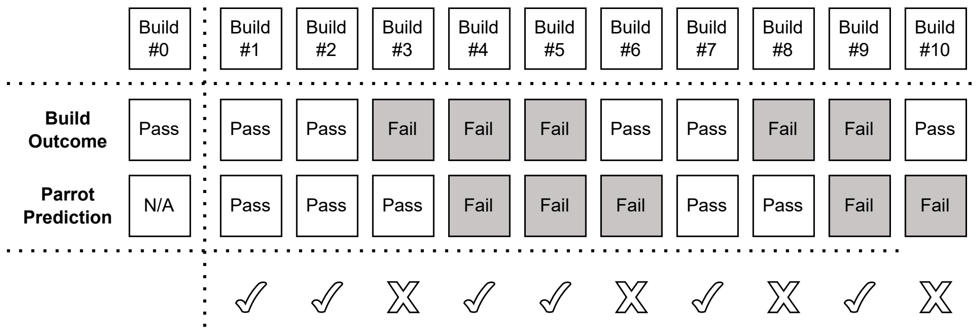

Figure 6.2: The parrot approach repeats the previous build outcome as its prediction, achieving accuracy, precision, and recall of .60.

Although RavenBuild and RavenBuild-CR improve on the state-of-the-art BuildFast model, we find that a naïve "parrot" approach to build outcome prediction that simply echoes the previous build as a prediction for the current one can perform as well as these state-of-the-art approaches.

In this section, we evaluate the performance of Parrot in comparison to BuildFast, RavenBuild, and RavenBuild-CR.

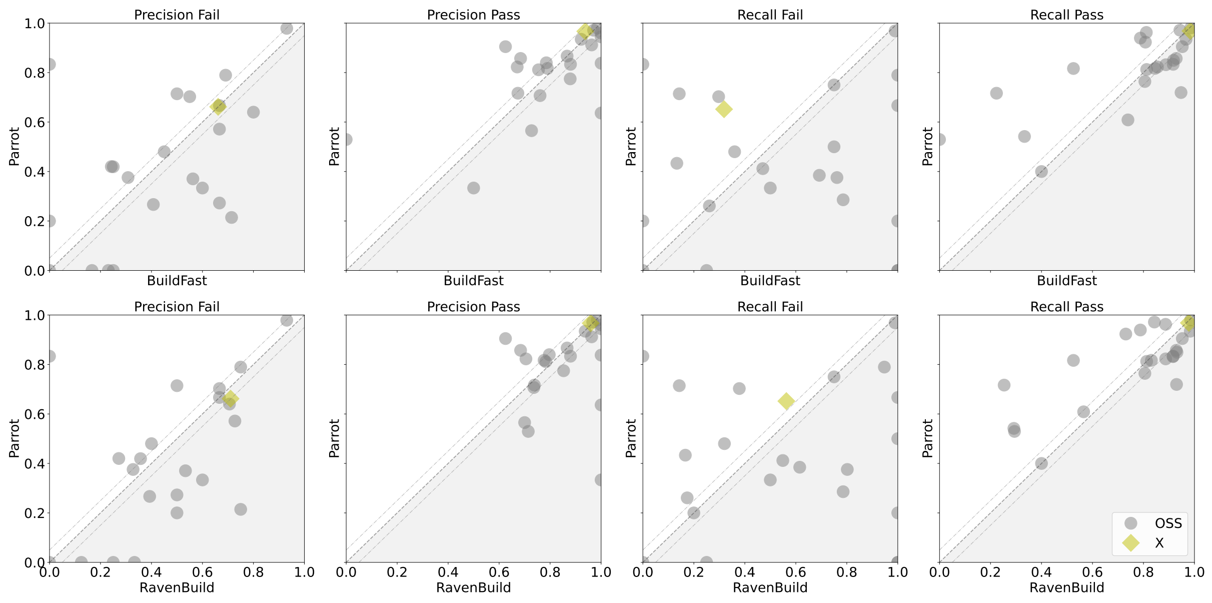

Figure 6.3: RavenBuild (Project X), RavenBuild-CR, and BuildFast cannot consistently outperform the parrot approach.

6.4.1 Approach

Figure 6.2 provides an example application of Parrot. We use Parrot as a benchmark to which we compare the performance of BuildFast, and RavenBuild-CR on the studied open-source projects, and BuildFast and RavenBuild on Project X.

Figure 6.3 plots the performance of the parrot approach (y-axis) against that of BuildFast and RavenBuild/RavenBuild-CR (x-axis) in terms of precision-fail, precision-pass, recall-fail, and recall-pass, respectively. Each open-source project is represented by a point, and Project X is denoted by a diamond-shaped point. To visually compare the performance of BuildFast and RavenBuild with the Parrot approach, we divide the coordinate space with a diagonal line into unshaded and shaded regions. Points falling in the unshaded region indicate that the Parrot approach yields better results, whereas points falling in the shaded region indicate that the other approach is performing better.

To further refine the comparison, we include two lines adjacent to the diagonal line, which denote a 5% confidence interval. We consider points that fall within the confidence interval to have a performance within the margin of error, and the difference between the model and Parrot is likely indistinguishable. Conversely, points that fall outside of the confidence interval are considered to have performance that is distinguishable between the state-of-the-art models and the parrot approach. Therefore, by counting the number of points falling in the unshaded region, we can infer how often Parrot outperforms state-of-the-art build outcome prediction models.

To study why naïve predictions that simply echo the previous build outcome achieve comparable performance to the state of the art, we count the number of builds that have the same outcome as their immediate predecessors. Table 6.6 shows the total number of passing and failing builds, the number of passing and failing builds that have the same outcome as their immediate predecessors, and the relative percentages in each category for the studied open-source projects and Project X.

6.4.2 Results

Observation 5: The naïve Parrot outperforms both BuildFast and RavenBuild in terms of time-to-feedback. Table 6.7 shows the performances of Parrot, BuildFast, and RavenBuild(-CR). While both BuildFast and RavenBuild(-CR) outperform Parrot in F1-fail, recall-fail, precision-fail, and AUC in the open-source context, and in recall-pass, precision-fail, and AUC in the Project X context, Parrot outperforms both approaches in terms of Gain. More specifically, Parrot achieves better Gain scores by improving the Benefit score in the open-source context, and by reducing the Cost score in the Project X context.

Observation 6: BuildFast, RavenBuild, and RavenBuild-CR often underperform with respect to Parrot in terms of precision and recall. Figure 6.3 shows 11, 11, 7, and 8 points in the four sub-graphs associated with BuildFast performance, and 9, 8, 7, and 9 points in the four sub-graphs associated with RavenBuild performance, fall in the unshaded regions, indicating that the parrot approach can outperform BuildFast and RavenBuild in at least one of the evaluation metrics (*i.e.,* precision-fail, precision-pass, recall-fail, and recall-pass) in at least seven of the studied projects. Overall, BuildFast and RavenBuild underperform with respect to the parrot approach in 39% and 35% of the 92 evaluation cases (*i.e.,* 23 projects $\times$ 4 evaluation metrics).

Indeed, a project-level inspection reveals that though Parrot does not outperform BuildFast and RavenBuild in all evaluation metrics, Parrot outperforms BuildFast and RavenBuild in, respectively, 10 and 6 of the 23 studied projects in 3 of the 4 evaluation metrics.

In the cases where build outcome prediction models outperform the Parrot approach, the difference in performance is often within the margin of error. Figure 6.3 shows BuildFast and RavenBuild only outperform the Parrot approach by more than the margin of error in

6–12 projects, of the 23 projects in terms of precision-fail, precision-pass, recall-fail, and recall-pass. In 62% and 60% of the evaluated cases, BuildFast and RavenBuild do not significantly outperform the Parrot approach.

We suspect that Parrot takes advantage of the repeated build outcomes and provides predictions that offer little practical value. For example, if the testing set contains *True* labels only, a naïve model can achieve perfect performance in all evaluation metrics by simply predicting *True*, whereas it would be much harder for any model to score 100% in all evaluation metrics. Though Parrot outperforms both BuildFast and RavenBuild, Parrot fails to capture the scenarios when a build outcome flips from passing to failing or failing to passing, which are the scenarios that provide developers with the most insight. Therefore, Parrot outperforming BuildFast and RavenBuild does not imply the build outcome prediction work is meaningless. Instead, Parrot provides a baseline of the natural tendency of the build outcome prediction dataset.

Table 6.6: The number of builds that have the same outcome as their immediate predecessor in the passing and failing class, the total number of builds in the passing and failing class, and the percentage of the builds that have the same build outcome as their immediate predecessor in all builds in the passing and failing class.

	Open-Source			Project X		
	Passing	Failing	All	Passing	Failing	All
Same[a]	6,302	2,229	8,531	3,587	744	4,031
Total	7,662	3,590	11,252	3,898	743	4,640
%	82%	62%	76%	92%	59%	82%

[a] Builds that have same outcomes as their immediate predecessor.

Observation 7: Build outcomes tend to repeat their previous outcome. The fact that the Parrot approach achieves performance that is comparable to or better than the state-of-the-art models reveals build outcomes tend to repeat their previous outcome. Indeed, Table 6.6 shows that 76% of builds from open-source projects have the same outcome as their previous builds. More specifically, 82% of the passing builds and 62% of the failing builds have the same outcome as their immediately preceding build. In terms of Project X, 87% of all builds, 92% of the passing builds, and 59% of the failing builds have the same outcome as their immediately preceding build. Our finding aligns with the observation of Jin and Servant [39], and demonstrates that a bias towards the previous build outcome exists not only in open-source contexts but also in AAA video game projects like Project X.

Table 6.7: The performance of Parrot, BuildFast, and RavenBuild. Values in boldface fonts indicate the best performance among the three.

	Open-Source			Project X		
	Parrot	BuildFast	RavenBuild-CR	Parrot	BuildFast	RavenBuild
F1 fail	.421	.447	**.468**	**.657**	.43	.628
F1 pass	**.797**	.753	.771	.967	.961	**.968**
Recall fail	.422	.552	**.571**	**.652**	.319	.563
Recall pass	**.797**	.745	.748	.968	**.984**	.978
Precision fail	.420	.439	**.468**	.662	.662	**.71**
Precision pass	.798	.793	**.851**	**.967**	.938	.959
AUC	.610	.717	**.736**	.810	.687	**.881**
Benefit	**336.3**	229.4	243.3	292.7	**298.4**	286.6
Cost	158.3	138.1	**118.9**	**39.1**	81.91	46.5
Gain	**178**	91.3	124.4	**253.6**	216.5	240.2

Despite the improvements achieved in build outcome prediction, both RavenBuild and BuildFast are often outperformed by the naïve Parrot approach. However, Parrot cannot capture the flip cases that developers need to be notified of. Future build outcome prediction work should consider Parrot performance as a tendency indicator when evaluating.

6.5 Threats to Validity

Below, we describe the threats to the validity of our study.

6.5.1 Construct Validity

Construct validity concerns may creep into our study if our measurements are misaligned with the phenomena we set out to study. In our work, these threats could be related to the selection of evaluation metrics. We rely on cost metrics from state-of-the-art studies [13] to ensure a fair comparison between RavenBuild and BuildFast [13], and other baseline approaches. Nevertheless, the cost measured by these metrics might still not reflect the real cost incurred by build outcome prediction in practice. In particular, deployments that aggressively pursue savings, negative side effects may emerge when these approaches are actually adopted. To capture such costs, we would need to deploy the studied approaches in a live CI service and measure costs that are actually impacting practitioners and stakeholders. Unfortunately, this evaluation is not feasible at this stage given the critical role that CI services play in production pipelines at Ubisoft.

6.5.2 Internal Validity

One potential threat to our internal validity is the existence of flaky builds in the studied datasets. Flaky builds are builds that fail intermittently due to non-deterministic factors in the system under test, the CI service, or the deployment environment. At Ubisoft, failing build steps are automatically retried. Therefore, to mitigate the effect of flaky builds on our study and datasets, we consider a build step to be passing as long as it has one passing execution.

Another threat to internal validity has to do with the use of machine learning in the studied approaches, especially the steps of feature selection in the BuildFast model architecture. Some features might be left out in the feature selection process. Through project-level inspection, we confirm that the RavenBuild features are selected to train the models in the feature selection step.

In addition, the dependency-aware features are collected from the multidisciplinary dependency graphs, which take time to be fully extracted. To address the issue, we propose to use the previous version of the dependency graph. Indeed, our analysis shows that only 34% of the studied changes alter edges within the graph, but those changes only affect one side of the dependency (*i.e.,* the files that the added files are depending on), and thus would not invalidate dependency analyses performed on the prior graph state. The changes that do alter the graph structure in analysis-invalidating ways are easily identified by scanning for differences in the `import` section of the edited files, which can be used to prevent incorrect dependency analysis results from being adopted by the model. However, future work needs to address the cases where the prior dependency graph is invalidated for analysis.

6.5.3 External Validity

Threats to external validity have to do with the generalizability of our results to other systems. To ensure consistency with BuildFast, we sample Java projects from GitHub to construct the open-source dataset. The performances of RavenBuild might differ when applied to other projects, but the key insights of the context-ware, relevance-aware, and dependency-aware features remain the same, as these features are extracted from change information that does not depend on the platform or programming languages.

6.5.4 Reliability Validity

Reliability validity threats may impact the replicability of this study. Due to legal constraints, we cannot share our dataset from Project X, since datasets collected on a proprietary project may leak confidential information. We conduct our evaluation of state-of-the-art approaches on 22 open-source projects as well as Project X. This selection of the same language of subject systems as prior work [13] ensures consistency when comparing results.

6.6 Data Availability

The presented dataset and results, as well as data collection scripts for open-source projects, are available in our replication package [2].

6.7 Chapter Summary

In this chapter, we describe the unique characteristics of AAA video game project development that make build outcome prediction particularly challenging. In the context of Project X, we observe that non-code changes that do not modify source code also incur build failures, as non-code artifacts should be compiled in an order-sensitive manner that respects the specified dependencies. Moreover, code changes that have an impact that crosses the source-data boundary are more prone to build failures than code changes that do not impact non-code files. The method by which we construct the multidisciplinary dependency graph forges a new direction for the extraction of features for build outcome prediction. To better accommodate the non-code changes into build outcome prediction, and to exploit the abundant and rich dependency data, we propose features that are context, relevance, and dependency aware. We incorporate those features into RavenBuild—a novel solution for build outcome prediction. We find that RavenBuild outperforms BuildFast by

[2]https://zenodo.org/record/8388161

a substantial amount on Project X. While extracting the dependency graphs requires extensive project-specific understanding, we provided an alternative to RavenBuild that only adopts the context-aware and relevance-aware features, namely RavenBuild-CR. We find that RavenBuild-CR outperforms BuildFast in both the open-source and Project X settings. Surprisingly, however, we find that neither BuildFast nor RavenBuild can consistently outperform a naïve Parrot approach that simply echoes the previous build outcome as predictions. Though Parrot fails to capture the important flip cases, the comparable performance of Parrot reveals that build outcomes tend to repeat their immediate predecessor. Therefore, future build outcome prediction work should consider Parrot performance as a tendency indicator when evaluating.

Chapter 7

Conclusion

Dependency graphs are at the core of build systems—a key component of the CI process. While nodes representing source code are essential components of dependency graphs, nodes representing non-code files also exist, and can even dominate the graph in projects that require contributions from personnel with different expertise. To weave the artifacts into a cohesive system and generate rapid feedback, build systems are adopted by modern software organizations to execute build and test jobs. In addition, since the code and non-code nodes are interconnected through boundary nodes in dependency graphs, changes to one type of node may have an impact on other types of nodes.

7.1 Contribution and Findings

In this thesis, we conduct an empirical analysis of the cross-boundary changes in a video game project, which (1) quantifies the frequency and riskiness of cross-boundary changes, and (2) characterizes the context of cross-boundary changes being committed. Finally, we propose RavenBuild, which (3) leverages dependency analytics to enhance build outcome prediction. We claim that:

Thesis Statement: The presence and magnitude of cross-boundary change can be leveraged to enhance Continuous Integration.

1. *Quantifying Cross-Boundary Changes (Chapter 4)*
   Through quantitative analysis, we reveal that non-code nodes and edges connecting to non-code nodes dominate in the dependency graph, and code

changes frequently have a cross-boundary impact. Moreover, cross-boundary changes are more prone to build breakages and more frequently implicated in defect-inducing commits.

2. *Characterizing Cross-Boundary Changes (Chapter 5)*
   To understand the circumstance in which changes cross disciplinary boundaries, we analyze the commit messages and the contributors of cross-boundary changes. We discover that cross-boundary changes are more commonly associated with gameplay functionalities and feature addition, and they are produced disproportionally by a single team of developers.

3. *Context, Relevance, and Dependency Aware Build Outcome Prediction (Chapter 6)*
   We complemented the prior build outcome prediction work and proposed RavenBuild, which leverages context-aware, relevance-aware features, and dependency analytics in build outcome prediction, and RavenBuild-CR, which only uses context-aware and relevance-aware features to ease the adoption process.

7.2 Opportunities for Future Research

Our research demonstrates that dependency analytics can be leveraged to enhance build outcome prediction. To further exploit the dependency analytics, we present two future research paths:

7.2.1 Measuring the Riskiness of Files with Dependency Analytics

We believe that when assessing the riskiness of code changes, it is crucial to consider the transitive dependencies of the changed files. These dependencies can create a domino effect where a change in one file can impact multiple other files. Failure to account for transitive impact could lead to an incomplete assessment of risks. For instance, without considering these dependencies, a change initially accessed to be low-risk might transitively impact high-risk files, leading to an inaccurate estimation of the overall risk of the change. Therefore, we can leverage the dependency data to enhance change risk estimation.

7.2.2 Investigating Contributor Awareness and Communication of Impact of Cross-Boundary Changes

A video game project consists of code files contributed by developers and non-code files contributed by experts of other domains. Our analysis of the video game project reveals that cross-boundary changes carry higher risks compared to changes that do not impact non-code files. However, it is unclear whether the developers are aware of the risk. We will conduct a survey to understand the developers' level of awareness regarding cross-boundary changes. We believe that cross-team communication may lower the risk of cross-boundary changes. We will also investigate the level of communication between developers and experts of other domains regarding cross-boundary changes.

References

[1] Json compilation database format specification.

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

[3] Snowdrop game engine. Accessed: August 10, 2023.

[4] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering*, 47(12):2740–2754, 2021.

[5] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, 47(3):448–463, 2021.

[6] 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.

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

[8] Jafar M. Al-Kofahi, Hung Viet Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Detecting semantic changes in makefile build code. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 150–159, 2012.

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

[10] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940, 2013.

[11] 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.

[12] 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.

[13] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53, 2020.

[14] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[16] CircleCI. Circleci, 2023. Accessed on Date,September 26, 2023.

[17] 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.

[18] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[19] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[20] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 235–245, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. Build system aware multi-language regression test selection in continuous integration. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 87–96, 2022.

[22] Mikołaj Fejzer, Piotr Przymus, and Krzysztof Stencel. Profile based recommendation of code reviewers. *Journal of Intelligent Information Systems*, 50, 06 2018.

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

[24] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering*, 48(6):2040–2052, 2022.

[25] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. *IEEE Transactions on Software Engineering*, 46(1):33–50, 2020.

[26] GitHub. Github rest api documentation, 2023. Accessed on Date, September 26, 2023.

[27] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 211–222, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Priscilla E Greenwood and Michael S Nikulin. *A guide to chi-squared testing*, volume 280. John Wiley & Sons, 1996.

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

[30] Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 189–198. IEEE Computer Society, 2006.

[31] Foyzul Hassan and Xiaoyin Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 157–162, 2017.

[32] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 483–493. IEEE Press, 2015.

[33] 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.

[34] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 197–207, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us? a taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, page 99–108, New York, NY, USA, 2008. Association for Computing Machinery.

[37] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. The review linkage graph for code review analytics: A recovery approach and empirical study. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 578–589, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Xianhao Jin. Reducing cost in continuous integration with a collection of build selection approaches. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

ESEC/FSE 2021, page 1650–1654, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 13–25, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Xianhao Jin and Francisco Servant. Hybridcisave: A combined build and test selection approach in continuous integration. *ACM Trans. Softw. Eng. Methodol.*, dec 2022. Just Accepted.

[41] Xianhao Jin and Francisco Servant. Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. *J. Syst. Softw.*, 188(C), jun 2022.

[42] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, page 81–90, USA, 2006. IEEE Computer Society.

[43] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, 46(7):710–731, 2020.

[44] Changki Lee and Gary Geunbae Lee. Information gain and divergence-based feature selection for machine learning-based text categorization. *Information Processing & Management*, 42(1):155–165, 2006. Formal Methods for Information Retrieval.

[45] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015.

[46] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018.

[47] 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.

[48] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. Identifying and Understanding Header File Hotspots in C/C++ Build Processes. *Automated Software Engineering*, 23(4):619–647, 2016.

[49] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 141–150, New York, NY, USA, 2011. Association for Computing Machinery.

[50] Mehran Meidani. Towards an Enhanced Dependency Graph. Master's thesis, University of Waterloo, 200 University Ave. W., Waterloo, ON, Canada, December 2022.

[51] Mehran Meidani, Maxime Lamothe, and Shane McIntosh. Assessing the Exposure of Software Changes: The DiPiDi Approach. *Empirical Software Engineering*, page To appear, 2023.

[52] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242, 2017.

[53] Andre N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178 – 1193, 2017.

[54] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1–11, New York, NY, USA, 2014. Association for Computing Machinery.

[55] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 153–164, 2018.

[56] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 455–458, 2017.

[57] 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 multidisciplinary software development. In *2018 Fluid Dynamics Conference*, page 3856, 2018.

[58] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. Towards language-independent brown build detection. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2177–2188, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. Search-based peer reviewers recommendation in modern code review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 367–377, 2016.

[60] Cong Pan and Michael Pradel. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 553–565, New York, NY, USA, 2021. Association for Computing Machinery.

[61] 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.

[62] 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.

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

[64] Guoping Rong, Yifan Zhang, Lanxin Yang, Fuli Zhang, Hongyu Kuang, and He Zhang. Modeling review history for reviewer recommendation: A hypergraph approach. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1381–1392, New York, NY, USA, 2022. Association for Computing Machinery.

[65] 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.

[66] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: An automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1530–1534, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engg.*, 29(1), may 2022.

[68] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, may 2005.

[69] 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, 2012.

[70] 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.

[71] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150, 2015.

[72] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 805–816, New York, NY, USA, 2015. Association for Computing Machinery.

[73] 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.

[74] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, page 685–694, 2018.

[75] 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.

[76] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–270, 2015.

[77] Francisco Servant Xianhao Jin. What helped, and what did not? an evaluation of the strategies to improve continuous integration. *CoRR*, abs/2102.06666, 2021.

[78] Zheng Xie and Ming Li. Cutting the software building efforts in continuous integration by semi-supervised online auc optimization. In *International Joint Conference on Artificial Intelligence*, 2018.

[79] Zhao Xu, Yang Zhang, and Xiayan Xu. 3d visualization for building information models based upon ifc and webgl integration. *Multimedia Tools and applications*, 75(24):17421–17441, 2016.

[80] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github. *Inf. Softw. Technol.*, 74(C):204–218, jun 2016.

[81] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2016.

[82] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen. Using large-scale heterogeneous graph representation learning for code review recommendations at microsoft. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 162–172, 2023.

[83] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[84] 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.

[85] Manuela Züger and Thomas Fritz. Interruptibility of software developers and its prediction using psycho-physiological sensors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, page 2981–2990, New York, NY, USA, 2015. Association for Computing Machinery.