Code Impact Beyond Disciplinary Boundaries

Constructing a Multidisciplinary Dependency Graph and Analyzing Cross-Boundary Impact

Gengyi Sun
Software REBELs
University of Waterloo, Canada
gengyi.sun@uwaterloo.ca

Mehran Meidani
Software REBELs
University of Waterloo, Canada
mehran.meidani@uwaterloo.ca

Sarra Habchi
La Forge
Ubisoft Montréal, Canada
sarra.habchi@ubisoft.com

Mathieu Nayrolles
La Forge
Ubisoft Montréal, Canada
mathieu.nayrolles@ubisoft.com

Shane McIntosh
Software REBELs
University of Waterloo, Canada
shane.mcintosh@uwaterloo.ca

ABSTRACT

To produce a video game, engineers and artists must iterate on the same project simultaneously. In such projects, a change to the work products of any of the teams can impact the work of other teams. As a result, any analytics tasks should consider intra- and inter-dependencies within and between artifacts produced by different teams. For instance, the focus of quality assurance teams on changes that are local to a team differs from one that impacts others. To extract and analyze such cross-disciplinary dependencies, we propose the multidisciplinary dependency graph. We instantiate our idea by developing tools that extract dependencies and construct the graph at Ubisoft—a multinational video game organization with more than 18,000 employees.

Our analysis of a recently launched video game project reveals that code files only make up 2.8% of the dependency graph, and code-to-code dependencies only make up 4.3% of all dependencies. We also observe that 44% of the studied source code changes impact the artifacts that are developed by other teams, 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 the same team (74% of the contributors are associated with that team).

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; **Software testing and debugging**; **Maintaining software**.

KEYWORDS

interdisciplinary dependencies, build systems, impact analysis

ICSE '24, April 14–20, 2024, Lisbon, Portugal
2024. ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3639477.3639726

1 INTRODUCTION

Various software artifacts need to be carefully developed in order to produce a software system. Naturally, source code describes system behaviour, but automated tests are needed to exercise the system under simulated conditions [8]. Moreover, containerization tools (*e.g.,* Docker) specify the execution environment in which the system should operate [12]. To weave these artifacts into a cohesive whole, software organizations rely on *build systems*, which specify and resolve internal and external dependencies, and recognize the conditions under which they should be traversed. In addition, build systems orchestrate the invocation of order-dependent commands that preprocess, compile, assemble, link, analyze, and package software artifacts into deliverables [16]. At their core, build systems specify and reason about build behaviour using a *dependency graph*, *i.e.,* a directed acyclical graph where nodes represent software modules (*e.g.,* source code files) and directed edges indicate dependencies between modules. While the dependency graph is at the heart of build execution, it can also be leveraged to perform software analyses, such as failure prediction [53], maintenance analysis [5, 6], quality improvement [22, 28, 40], and impact analysis [29, 43, 44].

In software projects that involve personnel from different disciplines, the breadth of software artifacts can be vast [45]. For example, producing high-budget (*a.k.a.,* 'AAA') video games requires the careful coordination of personnel with divergent expertise, such as technical software staff (*e.g.,* developers, QA, and operators), as well as creative staff (*e.g.,* graphic artists, composers and musicians, script writers, and level designers). AAA games are typically composed of millions of lines of code, as well as hundreds of thousands of other digital assets, such as textures and animations [32].

Multidisciplinary teams require a multidisciplinary dependency graph. Consider a change to a source code file that repositions an object in a game. This repositioning may have a transitive impact on other objects within the location in the game. To trace the impact of that change, we need a graph that captures *intra-dependencies* within each domain, as well as *inter-dependencies* among all of the other digital assets involved. Inaccurate analysis due to an incomplete dependency graph will lead to under- or over-estimating the impact of a change. While dependency graphs have been explored in the general development context [5, 22, 53], 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 paper, we show how such a multidisciplinary dependency graph can be extracted from a AAA video game project and study the properties of that graph (Section 2). Then, we leverage the graph to address five research questions spanning two dimensions:

1.1 Quantification (Section 3)

RQ1: What is missed by a 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. Thus, to better understand their potential impact, we first set out to study the prevalence of non-code artifacts and their dependencies in the multidisciplinary 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 confirm that 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 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 significant (Boschloo's test, $\alpha = 0.05$).

1.2 Characterization (Section 4)

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 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: It is clear that cross-boundary changes are a common and risky type of change in multidisciplinary development. Next, 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 one team or dispersed across multiple teams. Identifying these contributor tendencies can help management to better account for the 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 the same team. Indeed, cross-boundary changes within the studied game tend to be concentrated rather than dispersed.

Our findings highlight the prevalence of cross-boundary changes—a special type of change that occurs during the development of multidisciplinary projects. Despite being a regular occurrence, cross-boundary changes are highly impactful and risky, and they are primarily produced by the same team. The multidisciplinary dependency graph that we extract and analyze in this paper lays the foundation for improvements to reviewer recommendation and change prioritization for continuous integration at Ubisoft.

2 CONSTRUCTING THE MULTIDISCIPLINARY GRAPH

The studied project is the most recent installment of a popular video game in a franchise that was released years ago by Ubisoft. The project contains millions of production files, including source code, graphical assets, and audio samples. Throughout its development history, more than 100,000 change sets have been committed since 2022, providing a rich source of dependency graph data.

In this section, we present our approach to constructing the multidisciplinary dependency graph of the studied project.

2.1 Graph Construction Approach

We first group artifacts into code and non-code categories. Source code, header files, and libraries are categorized as code artifacts, while the rest of the files are categorized as non-code artifacts. In our context, non-code artifacts include machine learning models, animations, sound, 3D models, and textures.

We construct the dependency graph in separate steps. First, we construct the non-code portion of the graph (Section 2.2). Second, we construct the code portion of the graph (Section 2.3). Finally, we connect the non-code and code portions into the multidisciplinary dependency graph (Section 2.4).

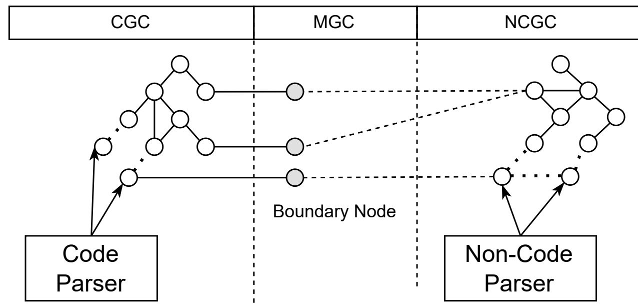

Figure 1: An overview of the graph extraction approach.

2.2 Non-Code Graph Construction

Figure 1 provides an overview of the non-code graph construction stage, which is composed of change set extraction (NCGC1) and graph construction (NCGC2) phases. Below, we describe each phase.

2.2.1 Extract Non-Code Changes (NCGC1). Nowadays, software projects may contain hundreds of thousands of non-code artifacts. Similar to the concept of code reusability, non-code artifacts are also created in a modular way such that they intra-depend on each other to form composite objects. For instance, the 3D model of a table in a game contains a surface, four legs, and a texture. The texture describes the characteristics of the surface like its density, weight, and smoothness. The surface, legs, and the texture are modules that are stored in separate files on which the table model depends.

Snowdrop [2] is a game engine developed by Ubisoft, which combines the artifacts into a cohesive whole. Although Snowdrop is capable of producing a non-code dependency graph for each change set, re-extracting this graph for every change set in the studied period would be prohibitively expensive because the number of non-code files in our studied project is on the order of millions.

Our approach starts with an initial non-code graph generated by Snowdrop and incrementally updates it by interpreting the changed files and performing corresponding graph updates when necessary. Thus, we extract all changed files and determine the corresponding graph updates from change sets that we extract from the version control system.

2.2.2 Build Non-Code Dependency Graph (NCGC2). There are plenty of file types among the non-code artifacts. For example, animation files are different from 3D model files in terms of content and format. An `OBJ` 3D object file depends on `MTL` material files (expressed using the `mtllib` statement), and `MTL` files may use `.MPC` color texture files (imported using the `map_Ka` statement) [1].

To account for this variety, we build a specific parser for each file extension that appears in our studied project. Each parser analyzes non-code files with the aim of identifying dependent files. At higher levels, files depend on multiple artifacts to build composite objects. For example, the `.uasset` file format from the Unreal game engine allows game creators to reference animations, sounds, and textures to draw an object within the game.

Since the parsing process is time-consuming and there are plenty of non-code files to process in a typical game project, we build the non-code graph incrementally. We start with the onerous creation of the initial graph, but can then update it based on changed files. More specifically, for each changed file, we perform one of the following actions:

- **Add**: For each added file, we add a new node to the graph and parse it to identify its children.
- **Remove**: When a file is removed, the corresponding node and its edges are removed from the graph.
- **Scan**: For each updated file, we remove the node and its edges from the graph. Then, we re-parse the file from scratch using Algorithm 1: $G.scan()$. By updating the scanned node and its edges, we prevent the duplication of dependencies from the updated node.

The output of this step is a directed graph representing dependencies among the non-code files.

Algorithm 1: Build Non-Code Dependency Graph

```
Function G.scan(File):
    File.children = binary_parser(File)
    for child : File.children do
        if not child.vistied then
            mark child visited
            G.addNode(child)
            // Node type: non-code
        File.addChildren(child) // Connect nodes
    G.File = File // Initialize or Update File node
Function BuildNonCodeGraph(changes):
    for changeID : changes do
        getChangedFileActions(changeID)
        for each (File, Action) do
            switch Action do
                case add, update do
                    G.scan(File)
                case delete do
                    G.remove(File)
        return Save G
```

2.3 Code Graph Construction

Figure 1 provides an overview of the code graph construction stage, which is composed of compilation database extraction (CGC1) and graph construction (CGC2) phases. Below, we describe each phase.

2.3.1 Extract Code Compilation Database (CGC1). We rely on the `import` statements in each source file to find other dependent source code files. For example, in C++, the `#include` preprocessor directive imports a referenced file within the context of the file in which the statement appears. The preprocessor resolves imported files by searching a list of directories in the search path. While there are default search locations, the search path is often updated within build specifications. In this study, we extract search path entries from the build system.

Algorithm 2: Build Code Dependency Graph

```
Function AddChildren(File, SearchPaths):
    if not File.visited then
        mark File visited
        imports = import_parser(File)
        for import : imports do
            for path : SearchPaths do
                if path+import exists then
                    // Node type: code or boundary
                       node
                    G.addNode(import)
                    File.addChild(import)
                    // Traverse up dependencies until
                       exhausted
                    AddChildren(import, SearchPaths)
                    break
                    // Terminate when all visited
Function BuildCodeGraph():
    for changeID : changes do
        Files, Paths = read(compile_command.json)
        for File : Files do
            G.addNode(File)
            AddChildren(File, Paths)
        return Save G
```

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

- **file**: the full path of the file in the compilation process.
- **directory**: The root directory from which relative paths are being resolved.
- **command**: The list of arguments that have been passed to the compiler for this specific compilation unit.

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

2.3.2 Build Code Dependency Graph (CGC2). In this step, we iterate over the compilation database file (`compile_commands.json`) generated in step CGC1 to identify source code files. We also analyze the '-I' search path arguments in the command section of each file and store them in a list that preserves the order of precedence from the compilation database.

Next, we resolve each file name passed to the `#include` preprocessor directive by checking the list of search path locations in the same order of precedence as the preprocessor. If a node with the exact path matching a referenced file does not exist in the graph, we create the node and recursively invoke Algorithm 2: *AddChilren*() with this new file as input. Finally, we connect the source node to the dependency under analysis to connect it to the rest of the graph. Note that it is guaranteed that the algorithm will terminate, since we do not analyze existing (visited) nodes. The output of this step is an adjacency list, which represents the file-level dependency graph for code files.

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

2.4 Multidisciplinary Graph Construction

Figure 1 provides an overview of the multidisciplinary graph construction stage (MGC), which is composed of boundary node extraction (MGC1) and graph merging (MGC2) phases. Below, we describe each phase.

2.4.1 Extract Boundary Nodes (MGC1). Eventually, the artifacts produced by different teams must be integrated to build the final software [31]. The game engine often enables a data-driven development approach [19]. Thus, developers provide generic and game-specific boundary nodes to the engine, which can be later used by artists. These boundary nodes are the bridge between code and non-code nodes [35]. In our study, boundary nodes are defined as classes that inherit from the 'NodeGraph' class. The exact class name also appears in the non-code files, which are inputs to these nodes. We process each source file to detect classes that are defined in header files and implemented in `.cpp` files. Unlike code and non-code nodes that represent files, boundary nodes are not actual files, but hyper-nodes that connect the code and non-code nodes. While this step is implemented in a project-specific manner, we conjecture that the concepts will generalize to other multidisciplinary settings. The output of this step is a map holding the one-to-many relationships between classes (*i.e.,* boundary nodes) and code files.

2.4.2 Merge Non-Code and Code Graphs (MGC2). The intersection between the non-code and code graphs are the boundary nodes (*i.e.,* edges from non-code to boundary node as inputs, and from boundary node to the code files where the node is implemented). Using the map generated by MGC1, we identify the corresponding files for those boundary nodes and add edges from the non-code graph (generated in NCGC2) to the code graph (generated in CGC2).

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

While our approach relies on the build information produced by Snowdrop for constructing the dependency graph, our main contribution lies in the efficiency of incremental updates for the non-code graph, as well as the creation of a novel multidisciplinary dependency graph.

Table 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%

3 QUANTITATIVE ANALYSES

In this section, we present our quantitative analyses with respect to three research questions. For each research question, we present our approach to addressing it and the results that we observe.

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

Approach: Following the process described in Section 2, we construct 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.

Results: Table 1 presents the numbers of the three types of nodes and the four types of edges in the graph. Figure 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 the studied project, the vast majority of nodes (97.2%) and edges (95.7%) will be missed if non-code artifacts are excluded from the dependency graph.

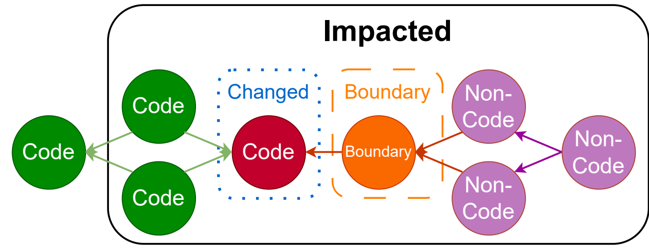

Figure 2: A cross-boundary change impacts nodes from another discipline that are depending on the changed nodes.

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

Approach: Considering each graph contains millions of nodes, we extract the dependency graph of 4,640 changes on the main branch that span 11 weeks of development, occupying more than 1.8 TB of memory space. Then, we 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. First, if a change adds, updates, or deletes any code files, we call it a code change. Otherwise, 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 2.

Results: Table 2 compares the numbers of the dependent nodes associated with each change. Figure 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 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

Table 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.

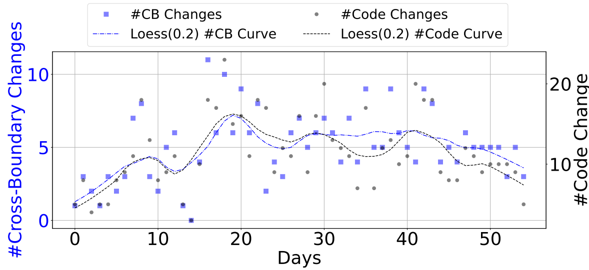

Figure 3: The daily occurrence of cross-boundary changes (blue y-axis on the left) and code changes that do not cross boundaries (black y-axis on the right).

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 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.**

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 [30] at Ubisoft, which identifies the defect-inducing commits with the SZZ algorithm presented by Kim *et al.* [26]. 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 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 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 (51% of the time) and induce defects (67% of the time) with significantly greater rates than changes that do not cross boundaries (44% and 37% of the time, respectively).**

4 CHARACTERIZATION STUDY

In this section, we present our characterization of cross-boundary changes with respect to two research questions. For each research question, we present our approach to addressing it and the results that we observe.

Table 3: The percentage of changes that are 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:
$^{*}p < 0.05;^{**}p < 0.01;^{***}p < 0.001; otherwise\ p \geq 0.05$

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 questions, 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.

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. We apply Boschloo's Exact test to assess the significance of the observed differences between cross-boundary changes and others.

Results: Table 2 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 3 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 3 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 3 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 3 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 [3] tags are more prevalent among code changes that do not cross boundaries. Table 2 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 3 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 3 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 3 shows that 36% and 16% of non-code changes include file additions and deletions, while 26% and 8% of

[3] The defect-fix pairs are many-to-many relationships [13, 26, 39].

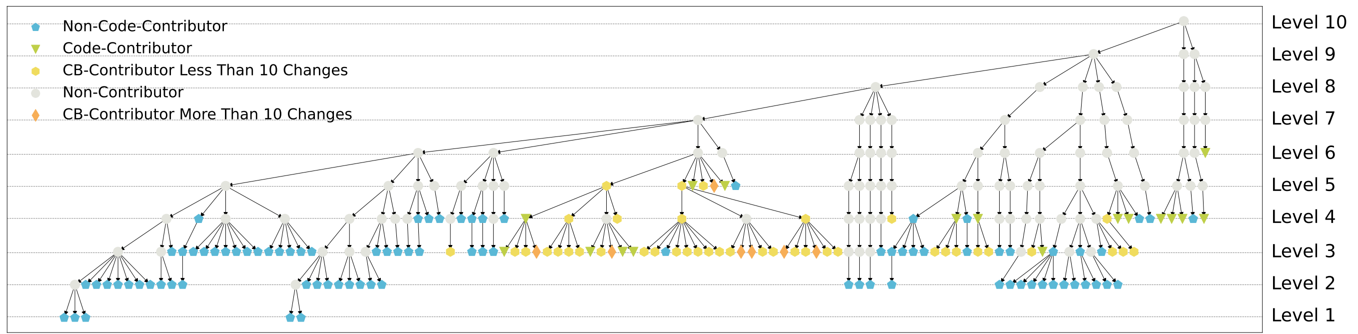

Figure 4: Organization graph of the contributors that committed at least one change during the studied period.

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.

RQ5: Who produces cross-boundary changes?

Approach: In collaboration with the human resources team, we extract the organizational graph of contributors who committed during the study period. Starting from the contributors, we traverse the graph upwards to their direct managers and continue transitively until all contributors share the same manager. The levels are defined from the bottom of the tree structure to the top, and 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) are considered to be members of the same team.

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, their data are omitted. Details regarding the impact of this omission on the validity of our conclusions will be discussed in Section 7.

Results: Figure 4 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 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 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 4. 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 4.

The contributors who have committed cross-boundary changes, namely cross-boundary contributors (*i.e.,* CB-contributor in Figure 4), 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 4 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 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 [53], 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 code review.** 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.

6 RELATED WORK

Dependency graphs have long been at the heart of software build systems. For example, Feldman [16] 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 [29, 41]. Inspired by this work, we incorporate static code analysis to extract the nodes that connect the non-code and code graphs to each other.

Research has leveraged dependency graphs to tackle other software engineering challenges. For example, Zimmermann *et al.* [53] used complexity metrics extracted from dependency graphs to predict subsystem failures. Cao *et al.* [9] 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.

6.1 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 [7, 11, 33, 42, 50]. Others focus on file paths [15, 46] or social network graphs [33, 36, 49]. Hirao *et al.* [23] demonstrated the performance of CRRs tend to improve if the linkage between reviews is considered. Zhang *et al.* [51] 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.* [27] 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.

6.2 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 [47] 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 [4, 25, 37] or trained using statistical or machine learning approaches [3, 10, 20, 21, 24, 34, 38, 48]. Similar rules and features can be crafted from to exploit the multidisciplinary dependency graph. For example, as shown by Observation 3, 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 [14], 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.* [18] 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 [52], 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.* [17] 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 re-generating 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.

7 THREATS TO VALIDITY

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

7.1 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.

The compilation database that we generate (see Section 2) is for the executable version of the game as it is deployed on the Windows operating system. It is not uncommon for video games to target multiple platforms, which would each produce a different compilation database. However, during the studied period, all the developments related to the game that we studied occurred in platform-independent areas. Thus, the specific platform used for generating the compilation database does not affect the results that we present in this paper.

7.2 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 4, 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.

7.3 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.

8 CONCLUDING REMARKS

In this paper, 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 one team rather than dispersed across the organization.

8.1 Future Work

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. Currently, research on accelerating the CI pipeline in the studied project using a build outcome prediction model has adopted features extracted directly from the dependency graphs. Our ongoing work is also applying our learnings to another video game project at Ubisoft. Indeed, analysis of their multidisciplinary dependency graph (*i.e.,* Observation 3, 4 and 5) is being used by a build outcome prediction model to decide whether pre-running a change set in a staging environment would be likely to prevent a breakage from being promoted to the active development branch.

Our findings may also generalize to other systems with heterogeneous development artifacts (e.g., some components implemented in one programming language and other components implemented in others). For example, in such a system, a false input that exists in a script that coordinates components of different programming languages will not cause a compilation error but will propagate its impact through dependencies, and cause a test failure (or worse!) in one of the downstream components. We encourage future researchers to replicate the study on such systems to evaluate the generalizability of our findings.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of MITACS Canada.

REFERENCES

[1] [n. d.]. MTL Material Format (lightwave, OBJ). http://paulbourke.net/dataformats/mtl/

[2] [n. d.]. *Snowdrop Game Engine.* https://www.massive.se/project/snowdrop-engine/ Accessed: August 10, 2023.

[3] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2021. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2740–2754. https://doi.org/10.1109/TSE.2020.2967380

[4] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2021. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* 47, 3 (2021), 448–463. https://doi.org/10.1109/TSE.2019.2897300

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

[6] Jafar M. Al-Kofahi, Hung Viet Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2012. Detecting semantic changes in Makefile build code. In *2012 28th IEEE International Conference on Software Maintenance (ICSM).* 150–159. https://doi.org/10.1109/ICSM.2012.6405266

[7] Vipin Balachandran. 2013. 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).* 931–940. https://doi.org/10.1109/ICSE.2013.6606642

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

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

[10] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2021. BuildFast: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20).* Association for Computing Machinery, New York, NY, USA, 42–53. https://doi.org/10.1145/3324884.3416616

[11] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. 2021. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing* 100 (2021), 106908. https://doi.org/10.1016/j.asoc.2020.106908

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

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

[14] Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. 2022. 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).* 87–96. https://doi.org/10.1145/3510457.3513078

[15] Mikołaj Fejzer, Piotr Przymus, and Krzysztof Stencel. 2018. Profile based recommendation of code reviewers. *Journal of Intelligent Information Systems* 50 (06 2018). https://doi.org/10.1007/s10844-017-0484-1

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

[17] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. 2022. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* 48, 6 (2022), 2040–2052. https://doi.org/10.1109/TSE.2020.3048335

[18] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015).* Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/2771783.2771784

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

[20] Ahmed E. Hassan and Ken Zhang. 2006. 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.* IEEE Computer Society, 189–198. https://doi.org/10.1109/ASE.2006.72

[21] Foyzul Hassan and Xiaoyin Wang. 2017. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 157–162. https://doi.org/10.1109/ESEM.2017.23

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

[23] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019).* Association for Computing Machinery, New York, NY, USA, 578–589. https://doi.org/10.1145/3338906.3338949

[24] Xianhao Jin and Francisco Servant. 2020. A Cost-Efficient Approach to Building in Continuous Integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20).* Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/3377811.3380437

[25] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (2022), 111292. https://doi.org/10.1016/j.jss.2022.111292

[26] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06).* IEEE Computer Society, USA, 81–90. https://doi.org/10.1109/ASE.2006.23

[27] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. 2020. Does Reviewer Recommendation Help Developers? *IEEE Transactions on Software Engineering* 46, 7 (2020), 710–731. https://doi.org/10.1109/TSE.2018.2868367

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

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

[30] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. 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).* 153–164.

[31] 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. 2018. Application of the Dependency Inversion Principle to Multidisciplinary Software Development. In *2018 Fluid Dynamics Conference.* 3856.

[32] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. 2022. Towards Language-Independent Brown Build Detection. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 2177–2188. https://doi.org/10.1145/3510003.3510122

[33] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-Based Peer Reviewers Recommendation in Modern Code Review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 367–377. https://doi.org/10.1109/ICSME.2016.65

[34] Cong Pan and Michael Pradel. 2021. Continuous Test Suite Failure Prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021).* Association for Computing Machinery, New York, NY, USA, 553–565. https://doi.org/10.1145/3460319.3464840

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

[36] Guoping Rong, Yifan Zhang, Lanxin Yang, Fuli Zhang, Hongyu Kuang, and He Zhang. 2022. Modeling Review History for Reviewer Recommendation: A Hypergraph Approach. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 1381–1392. https://doi.org/10.1145/3510003.3510213

[37] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021).* Association for Computing Machinery, New York, NY, USA, 1530–1534. https://doi.org/10.1145/3468264.3473115

[38] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the Prediction of Continuous Integration Build Failures Using Deep Learning. *Automated Software Engg.* 29, 1 (may 2022), 61 pages. https://doi.org/10.1007/s10515-021-00319-5

[39] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *SIGSOFT Softw. Eng. Notes* 30, 4 (may 2005), 1–5. https://doi.org/10.1145/1082983.1083147

- [40] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. Build code analysis with symbolic evaluation. In *2012 34th International Conference on Software Engineering (ICSE)*. 650–660. https://doi.org/10.1109/ICSE.2012.6227152
- [41] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. 2012. SYMake: a build code analysis and refactoring tool for makefiles. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 366–369.
- [42] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. 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)*. 141–150. https://doi.org/10.1109/SANER.2015.7081824
- [43] Shuying Wang and Miriam AM Capretz. 2009. A dependency impact analysis model for web services evolution. In *2009 IEEE International Conference on Web Services*. IEEE, 359–365.
- [44] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. 2018. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. 685–694.
- [45] Mark Werner. 1996. Barriers to a collaborative, multidisciplinary pedagogy [software development teams]. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*. IEEE, 203–210.
- [46] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. 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)*. 261–270. https://doi.org/10.1109/ICSM.2015.7332472
- [47] Francisco Servant Xianhao Jin. 2021. What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration. *CoRR* abs/2102.06666 (2021). arXiv:2102.06666 https://arxiv.org/abs/2102.06666
- [48] Zheng Xie and Ming Li. 2018. Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization. In *International Joint Conference on Artificial Intelligence*.
- [49] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer Recommendation for Pull-Requests in GitHub. *Inf. Softw. Technol.* 74, C (jun 2016), 204–218. https://doi.org/10.1016/j.infsof.2016.01.004
- [50] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543. https://doi.org/10.1109/TSE.2015.2500238
- [51] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen. 2023. 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)*. 162–172. https://doi.org/10.1109/ICSE-SEIP58684.2023.00020
- [52] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2012. 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* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 40, 4 pages. https://doi.org/10.1145/2393596.2393642
- [53] Thomas Zimmermann and Nachiappan Nagappan. 2007. Predicting subsystem failures using dependency graph complexities. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 227–236.