Understanding the Implications of Changes to Build Systems

Mahtab Nejati
Software REBELs
University of Waterloo, Canada
mahtab.nejati@uwaterloo.ca

Mahmoud Alfadel
Department of Computer Science
University of Calgary, Canada
mahmoud.alfadel@ucalgary.ca

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

ABSTRACT

The maintenance of build systems imposes a considerable overhead on software development. Since automated quality assurance methods are rarely applied to build specifications, the importance of the role peer code review plays in the maintenance of build systems is amplified. Yet prior work shows that the review process for build systems suffers from a lack of build experts and effective tooling.

To support the understanding of changes to build specifications (a key stage in the review process), we propose BCIA—an approach to summarize the impact of changes to build specifications across the build configuration space. BCIA traverses the paths through which data and control flow in the prior and updated versions of the build system to generate an Impact Knowledge Graph (IKG), which describes the impact of the change across the build configuration space. We develop BuiScout—a prototype implementation of BCIA for CMake-based build systems. We use BuiScout to evaluate our approach through an empirical study of 10,000 change sets that we mine from ten large open-source projects that span a total of 28 development years. Our findings indicate that BuiScout can detect an impact that propagates to unmodified parts of the build system in 77.37% of the studied change sets. These changes impact a median of 14 unmodified commands, with a median of 95.55% of the impacted commands per change set appearing in unmodified files. Our study suggests that dedicated approaches, such as BCIA, have the potential to alleviate the challenges developers face when assessing the impact of changes to build systems.

KEYWORDS

Build systems, Build system maintenance, Impact analysis

ACM Reference Format:
Mahtab Nejati, Mahmoud Alfadel, and Shane McIntosh. 2024. Understanding the Implications of Changes to Build Systems. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695514

1 INTRODUCTION

Build systems orchestrate the transformation of software sources into executables. The maintenance of build systems is not trivial [1–3]—it introduces a substantial overhead on software development. Prior work [4] shows that up to 27% of source code changes and 44% of test code changes also update the build system. This overhead can impact a large proportion of team members, with up to 79%–89% of developers being affected [4]. Moreover, maintaining build systems for software with a high degree of compile-time variability requires a concerted effort [5].

As build systems continue to evolve, they become prone to quality decay [6], which can slow down builds [7, 8], cause build failures [7, 9, 10], or even lead to erroneous software behavior [11]. Despite these risks, systematic and automated quality assurance practices, such as automated testing, are rarely applied to build specifications, leaving peer code review as the primary method for sustaining the quality of build systems. Unfortunately, the changes to build systems are often not rigorously reviewed due to social and technical reasons [12], such as a pervasive lack of expertise in build systems and interest in their maintenance [12, 13].

In our prior work [12], practitioners lamented a perceived lack of tools to support the maintenance of build systems and the review of changes to build specifications. Understanding the implications of changes within the complex configuration space of build systems remains a challenging task [13]. Change Impact Analysis (CIA) has shown potential in improving the effectiveness and efficiency of code reviews in the context of production code by exposing the impact of changes across the system [14–17]. Yet, automated tools to facilitate CIA for build systems are scarce.

Inspired by the growing evidence of the benefits of adopting CIA in the code review cycle, we conjecture that an approach to conducting CIA on changes to build systems would help when creating and reviewing changes in build systems. Prior work [18, 19] proposes tools to illustrate the impact of changes to source code by tracing their propagation through the build system; however, to the best of our knowledge, a CIA approach does not yet exist for changes to build system itself. Such a technique would allow stakeholders to navigate the impact of the changes to the build system and potentially expose a missing or unintended impact of a change.

Therefore, we propose *BCIA*—an approach that uses data and control flow analysis to assess the impact of changes across the build configuration space. BCIA uses Conditional Definition-Use (CDU) chains [20]—an augmented *Definition-Use (DU) chain* that, along with the data flow information, implicitly captures control flow by storing the reachability condition of the definition and use points. BCIA then infers how change impact propagates within and across CDU chains based on value- and reachability condition-contamination patterns, storing this information in an Impact Knowledge Graph (IKG).

To evaluate the applicability of BCIA, we implement *BuiScout*, a prototype tool for BCIA that analyzes *CMake* build systems, which are renowned for the complexity of their maintenance [21]. We then use BuiScout to conduct an empirical study of 10,000 change

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695514

sets that we mine from 10 large open-source projects, spanning 28 development years, to address the following research questions:

RQ1. Impact Prevalence: ***How often do build-modifying change sets propagate their impact beyond the local scope?***

Motivation: The usefulness of BCIA will depend on how often the impact of the changes propagates in non-obvious ways. If changes frequently propagate an impact, then the value of BCIA is clear, but if an impact rarely propagates, then such a tool may be unnecessary. Therefore, we set out to understand how often BCIA detects propagation of an impact in real-world build maintenance activity.

Results: BuiScout detects impacts that propagate to unmodified parts of the build system in 77.37% of the 10,000 studied change sets, suggesting that such a tool would regularly provide useful data to stakeholders.

RQ2. Impact Characteristics: ***What are the characteristics of the propagating impact of build-modifying change sets?***

Motivation: The impact of changes that we detect in RQ1 may vary in terms of their magnitude (i.e., the number of commands that are affected) and their breadth (i.e., the location of the affected commands with respect to the change set). Both characteristics can influence the applicability of BCIA. For example, even if changes regularly have an impact that propagates, if that impact is of a small magnitude or does not propagate broadly, practitioners may not need tool-support to assess the impact. Therefore, we set out to understand the magnitude and breadth of the impact that BCIA detects in real-world build maintenance activity.

Results: Change sets with a propagating impact affect an overall median of 14 commands and that only 4.45% of these commands are local to the files that the change set modified directly. The remaining 95.55% of impacted commands require practitioners to recognize how impact propagates across build specifications, which prior work suggests is a skill that many lack [12, 13].

Contributions. The main contributions of this paper are as follows:

- BCIA—an approach that uses data and control flow analysis to infer how change impact propagates throughout build systems.
- BuiScout—a prototype implementation of BCIA for CMake.
- An empirical evaluation of the applicability of BuiScout (and BCIA) to real-world build maintenance efforts.
- A replication package containing the source code of BuiScout, the collected data set, and the analysis scripts that are required to reproduce our empirical study.[1]

2 THE BUILD PROCESS

In this paper, we describe how changes to the build system affect the build configuration space. We first define concepts related to the build process (Section 2.1) and then present a minimal build system example to outline the scope of the study (Section 2.2).

2.1 Key Build Concepts

The *build process* transforms software sources, such as source code, into executable software. This process is executed by *build tools*, which resolve dependencies that constrain the order in which build commands should be invoked. Dependency expressions and the commands should be invoked when outputs are out of sync and are written in *build specifications* that are organized within *build files*. Developers create a *build system*, consisting of a set of build specifications declared in one or more build files, to automate the software build process.

```
1  ..
2  file(GLOB SRC "src/app/*.c" "src/app/*.h")
3  ...
4  add_executable(server ${SRC} ${server_src})
5  ...
6  include(SetupFeatures.cmake)
7  ...
8  add_executable(client ${SRC} ${client_src})
9  ...
```

(a) The `CMakeLists.txt` file

```
1  ...
2  + list(APPEND SRC ${IRC_FILES})
3  ...
```

(b) The `SetupFeatures.cmake` file

Listing 1: A minimal example of a build system.

2.2 Example Build System

Listing 1 provides (a) a minimal example of a CMake-based build system with (b) a change set being applied. The example highlights portions of the build specifications from `CMakeLists.txt` and `SetupFeatures.cmake` files. The example build system is inspired by the ET-Legacy project[2] with simplifications such as moving and de-conditionalizing commands and renaming variables.

In CMake, by default, the entry point into the build system is the `CMakeLists.txt` file. Commands within the build files are evaluated sequentially to configure the build process. The `cmake` build tool processes the example build system a follows:

(1) The commands in the `CMakeLists.txt` file are evaluated until it encounters the `file` command (line 2). This command defines the `SRC` variable, which stores the list of files that match the specified patterns.
(2) The process reaches the `add_executable` command (line 4), where an executable target named `server` is defined, which depends on the dereferenced values of `SRC` (from Step 1).
(3) The `include` command (line 6) directs the build process to the `SetupFeatures.cmake` file, such that the commands in this file are processed within the same scope before continuing with the remaining commands in `CMakeLists.txt`.
(4) In the updated build system, the `SetupFeatures.cmake` file is modified by adding a `list` command (line 2). This command appends the values stored in `IRC_FILES` to `SRC` (from Step 1).
(5) After processing `SetupFeatures.cmake`, execution returns to `CMakeLists.txt` at line 7. The `add_executable` command (line 8), declares a new executable target named `client`, which depends on the dereferenced values of `SRC` (from Step 4 in the updated version and Step 1 in the prior version).

In this example, changes made to `SetupFeatures.cmake` that update the `SRC` variable will not impact the `server` executable, but will impact the `client` executable. This example illustrates how

[1] https://zenodo.org/doi/10.5281/zenodo.11505222

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

changes to build specifications may impact the build process in non-trivial ways. For instance, if the goal is to add a feature to both the `server` and `client` executables, the incomplete impact of the change might not be immediately apparent to the developer.

3 RELATED WORK

In this section, we position our work with respect to the literature on build system maintenance and change impact analysis.

Maintenance of Build Systems. Recent work has focused on supporting the maintenance of build systems through automated defect detection [22–27] and repair [28–30]. For example, Macho et al. [27] proposed an approach that incorporates method calls within the source code into the dependency graph to detect external dependency conflicts in a Java project. Hirebuild [29] was proposed to fix build breakages by extracting fixing patterns from historical data and successfully repaired 45% of their studied breakages. While the prior work proposes promising directions, they have not yet been implemented into practice-ready tools due to their limited effectiveness [30] and performance [31, 32]. As a result, quality assurance of build systems is still predominantly a manual process, imposing a substantial cognitive load on build maintainers [13].

Another recent line of work assists build maintenance in other ways. For example, approaches have been proposed to visualize the dependency graph to clarify interdependencies between components [33, 34], automate common refactoring tasks, such as renaming and removing targets [22, 35], detect the addition and removal of dependencies [35, 36], and summarize reports of build failures for debugging purposes [34, 37]. While these approaches offer valuable support, they are mostly concerned with guiding or automating the application of build system changes, whereas in this paper, we propose an approach that supports developers and reviewers to understand the implications of maintenance activities on build systems.

Macho et al. [38] stated that not only assisting the application of the change is necessary, but it is also essential to understand the details of the changes in build systems after they are applied. They proposed BuildDiff, a tool that computes the differences between pairs of Maven build specifications and detects the operation type, e.g., adding or deleting a dependency or updating the version of an artifact, to provide detailed information about what has been modified. However, their approach is not concerned with the implications the changes may have on other portions of the build system. In the build systems of projects with compile-time variability, the logic that establishes dependencies among internal and external sources can propagate in complex ways. In our study, we propose an approach that detects the impact of changes across the build system, facilitating change comprehension and review.

Change Impact Analysis. Change Impact Analysis (CIA) is the process of identifying and assessing the consequences of a change to the software to reveal their broader implications [39]. While CIA can be applied to a broad range of software artifacts, studies predominantly focus on code-based CIA techniques [40]. These studies have effectively implemented CIA techniques in change propagation [41–43], change effort estimation [44, 45], integration testing [46], defect detection [17, 47], and code review [14, 15]. Given its wide-ranging applications and benefits, studies have proposed approaches to automated CIA [40, 48–50]. Among these approaches, Dai et al. [51] highlighted that methods that analyze variable operations and trace their data flow paths yield substantial improvements in the effectiveness of the CIA.

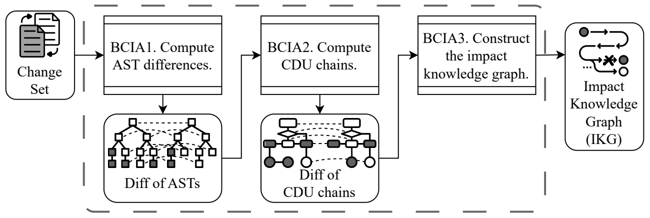

Change Set
BCIA1. Compute AST differences.
Diff of ASTs
BCIA2. Compute CDU chains.
Diff of CDU chains
BCIA3. Construct the impact knowledge graph.
Impact Knowledge Graph (IKG)

Figure 1: Overview of BCIA.

Studies on CIA in build systems have been limited. While prior research has explored the effects of changes to source or test code on build systems [52–54] or their output, such as targets [18, 19], the impact of modifications within the build system itself on the build configuration remains unexplored. We propose a fine-grained approach to Build Change Impact Analysis (BCIA), which aims to estimate the impact of such changes to support understanding the implications of changes to build systems.

Perhaps the most similar work to ours is that of Al-Kofahi et al. [55] where their tool, MkDiff, detects semantic changes in Makefiles. Our work differs from theirs mainly in two ways. First, MkDiff abstracts the build specifications into a Symbolic Dependency Graph (SDG) [22], which may obscure the logical pathways through which changes impact build rules. In contrast, our approach, BCIA, provides a clear view of how changes transitively affect the entire build system. Second, MKDiff lacks global analysis capabilities. It analyzes semantic changes within a selected file, incorporating other files only if the selected file depends on them. This approach requires developers to either have extensive knowledge of the build system to select the files that will be impacted for the analysis or to manually check multiple files to understand the full impact of changes. In contrast, BCIA automatically performs a global analysis on the build system, which helps to uncover a more comprehensive impact of changes and alleviates the need for such manual effort.

4 PROPOSED APPROACH

To understand how changes in build systems impact the build configuration space, we propose BCIA, an approach that extracts the semantic differences between snapshots of a build system (i.e., before and after a change set has been applied). Figure 1 provides an overview of BCIA, which comprises three steps. Below, we describe the general behaviour of each step. Section 5 provides the details that are specific to the implementation of BuiScout, the prototype that we produce to conduct our empirical study.

BCIA1. Compute AST differences

Changes to build files may impact the build system in ways that developers do not expect or intend. We set out to understand how changes to build files can impact unmodified parts of build systems. This involves comparing two snapshots of the build system from before and after a change set that modifies build files has been applied.

An analysis that focuses on textual differences cannot detect features of the change set that developers often value, such as syntactic and semantic changes [56, 57]. To detect these features of change sets, as prior work suggests [56–61], we compute the differences between Abstract Syntax Trees (ASTs).

To do so, we first extract two snapshots of the state of the build system before and after the change set has been applied. We create a first snapshot by selecting all the build files within the codebase in its state prior to applying the change set. We then create a second snapshot by extracting the parts of the change set that modify, add, delete, or rename build files and apply the subset of changes to a copy of the first snapshot. We pair the ASTs representing two versions of each build file in the snapshots. Added and deleted files are paired for comparison with an empty AST, whereas modified build files are paired across the snapshots.

After pairing ASTs, differences are computed to highlight four types of AST nodes. *Added nodes* appear in the updated AST, but not in the prior one. *Deleted nodes* appear in the prior AST, but not in the updated one. *Moved nodes* appear in both updated and prior ASTs under parent nodes that are not matched with each other and are linked by matching edges across the ASTs. *Updated nodes* appear in both updated and prior ASTs under parent nodes that are matched with each other and are linked by matching edges across the ASTs. Other nodes are assumed to be unchanged and are not highlighted, but are matched across the ASTs.

BCIA2. Compute CDU chains

While differences in ASTs offer a more detailed perspective than textual differences, they are still limited to the change location. For instance, it is relatively easy to automatically determine if a change impacts the value of a variable when the assignment statement for that variable is altered. However, comprehending the broader impact of the change on distant parts of the code requires inference capabilities that are not directly supported by sets of AST differences.

Definition-Use (DU) chains extract data flow connections between the locations where identifiers have their values set or updated, and the locations where identifiers are dereferenced. *Conditional Definition-Use (CDU) chains* [20] enhance traditional DU chains by incorporating reachability conditions of data flow paths. In simple terms, a DU chain links each definition to all its direct usages along the data flow path and a CDU chain does the same while preserving the reachability condition of the definition and usage locations.

Assume that S is the set of statements in a snapshot where:

$$\forall i, j; s_{[i]}, s_{[j]} \in S \land i < j \Rightarrow s_{[i]} \textit{ precedes } s_{[j]} \textit{ on the execution path.}$$

A definition point for an identifier id, in statement $s_{[i]} \in S$ is denoted as $d^{id}_{s_{[i]}} \in D$, where D is the set of all definition points. The use point of $d^{id}_{s_{[i]}}$ in statement $s_{[j]} \in S$ is denoted as $u^{id}_{s_{[j]}} \in U$, where U is the set of all use points.

CDU chains are constructed based on definitions and usages of identifiers, e.g., variables or functions, and each snapshot of the build system yields a set of CDU chains, C. Formally, a CDU chain for the identifier id is defined as:

$$c^{id}_{s_{[i]}} = (d^{id}_{s_{[i]}}, \{u^{id}_{s_{[i+j_1]}}, ..., u^{id}_{s_{[i+j_m]}}\}) \in C$$

where $d^{id}_{s_{[i]}}$ is considered the head of the chain and the sequence of its use points is the tail of the chain. For each use point $u^{id}_{s_{[i+k]}}; k \in \{j_1, j_2, ..., j_m\}$ in $c^{id}_{s_{[i]}}$, there is a data flow path from $d^{id}_{s_{[i]}}$ that reaches $u^{id}_{s_{[i+k]}}$ where no other definition points re-define id on that path. Note that an identifier may have multiple CDU chains, one per defining statement.

When constructing the CDU chains for a snapshot, we store the following data as references to their corresponding nodes in ASTs:

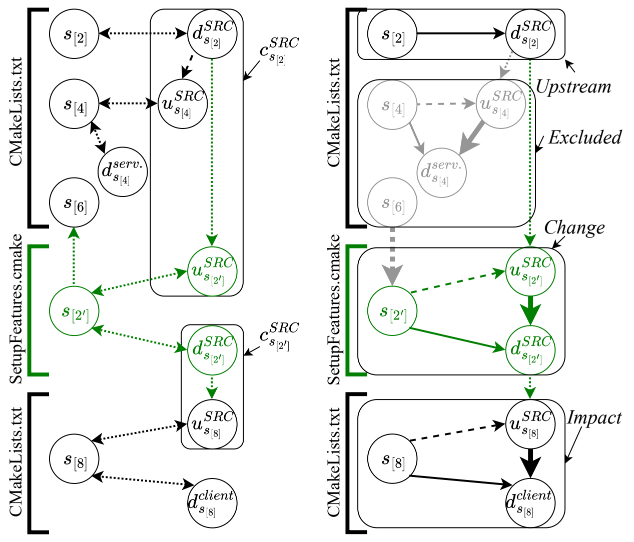
- The *statements* ($s_{[i]}$). For each $s_{[i]}$, we also store references to the identifiers that $s_{[i]}$ defines and dereferences, as well as its reachability conditions along the data flow path.
- The *definition points* ($d^{id}_{s_{[i]}}$). For each $d^{id}_{s_{[i]}}$, we store id, which is used to match $d^{id}_{s_{[i]}}$ with its use points along the data flow path, and the type of the value that is stored within id, e.g., variable or function. We also store references to both the defining statement and all of its direct use points.
- The *use points* ($u^{id}_{s_{[i]}}$). For each $u^{id}_{s_{[i]}}$, we store id, the type of the value that is stored within id, and a reference to the statement that dereferences id.

Figure 2a provides examples of CDU chains in our storage format. The CDU chains in Figure 2a are computed from the build system in Section 2.2 and do not include the use points with hidden definition points, i.e., the `server_src` and `client_src` variables. Note that statements are not formally an element of the CDU chains [20], however, they play an important role in detecting the impact of changes across CDU chains in BCIA3.

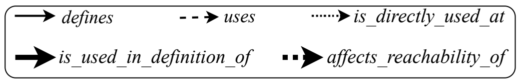
In Figure 2a, $S_{[i]}$ and $S_{[i']}$ represent the statements on the i^{th} line of the *CMakeLists.txt* and *SetupFeatures.cmake* files respectively. Moreover, in this figure:

- A bidirectional reference between a statement and a definition/use point indicates that the statement is directly defining/dereferencing the value of the identifier.
- A reference from a definition point to a use point represents a direct dereference of the defined value.
- A reference from a statement to a use point in one direction indicates that the reachability condition for the referencing statement relies on the referenced identifier.
- A reference from a statement to another statement in one direction indicates that the reachability condition for the referencing statement relies on the referenced statement. More specifically, this is when the referenced statement imports the referencing statement onto the data flow path, e.g., an import statement.

The construction of CDU chains for a snapshot of the build system relies on the correct detection of data flow paths such that definition points are linked to their actual use points. This involves extracting data flow paths that span the entire build system. To do so, we must (1) process ASTs in the order in which the build tool would process their corresponding build files, and (2) traverse each AST in the order that the build tool would traverse the corresponding statements. We construct the CDU chains for the two snapshots of each change set independently. Instances of the stored data are associated with their corresponding AST nodes. Therefore, the differences between the CDU chains are obtained based on the differences between the ASTs.



(a) Computed CDU chains **(b) Constructed IKG**



(c) Notations representing the IKG relationship types.

Figure 2: The resulting CDU chains and IKG from the analysis of the example in Listing 1.

Table 1: The types of relationships in the IKG.

Relationship	$(s\ ,\ \mathit{defines}\ ,\ d_s^{id})$
Description	When a statement defines an identifier.
Relationship	$(s\ ,\ \mathit{uses}\ ,\ u_s^{id})$
Description	When a statement dereferences an identifier.
Relationship	$(u_s^{id}\ ,\ \mathit{is\_used\_in\_definition\_of}\ ,\ d_s^{id'})$
Description	When a use point is dereferenced by a statement to define a definition point (*value contamination*).
Relationship	$(d_s^{id}\ ,\ \mathit{is\_directly\_used\_at}\ ,\ u_{s'}^{id})$
Description	When a definition point is dereferenced in a use point within a CDU chain (*value contamination*).
Relationship	$(d_s^{id}\ ,\ \mathit{is\_passed\_as\_argument}\ ,\ u_{s'}^{id})$
Description	When an argument is passed to a user-defined callable structure, e.g., a function, and its content is dereferenced within the body of the callable (*value contamination*).
Relationship	$(u_s^{id}\ /\ s\ ,\ \mathit{affects\_reachability\_of}\ ,\ s')$
Description	When a use point (in a conditional statement) or a statement (representing commands that import other build files or invoke user-defined processes) imposes a reachability condition on another statement (*reachability condition contamination*).

BCIA3. Construct the IKG

The constructed CDU chains link use points to their definition point and capture the basic impact of the change within a CDU chain, i.e., from the definition to its uses. However, changes can transitively affect disjoint CDU chains, which in turn, can propagate the impact of the change in complex ways throughout the build system. CDU chains alone do not explicitly store such transitive impacts.

Figure 2b provides examples of how the impact of changes can propagate across CDU chains. Below, we described the two common patterns of cross-chain impact propagation:

- *Value contamination:* If use point $u^{id}_{s_{[i]}}$ is either modified or affected by the change and there exists a definition point $d^{id'}_{s_{[i]}}$, statement $s_{[i]}$ uses the value of identifier id to set the value of the identifier id'. In this case, the impact of the change propagates from $c^{id}_{s_{[k]}}$, which contains $u^{id}_{s_{[i]}}$ in its tail, to $c^{id'}_{s_{[i]}}$.
- *Reachability condition contamination:* There are two types of this pattern. First, if a modified or affected use point $u^{id}_{s_{[i]}}$ occurs within a conditional statement $s_{[i]}$, then $u^{id}_{s_{[i]}}$ is recorded as the reachability condition for $s_{[j]}$ that appear within the body of the conditional statement $s_{[i]}$. Second, if a modified or affected statement $s_{[i]}$ imports other statements $s_{[j]}$ onto the data flow path, then $s_{[i]}$ is recorded as the reachability condition for $s_{[j]}$. In both cases, the impact of the change propagates to $s_{[j]}$ through its reachability condition, and will in turn, propagate to any definitions made by $s_{[j]}$, affecting their CDU chain.

We capture how changes impact code entities within and across CDU chains in a knowledge graph [62], which we refer to as the *Impact Knowledge Graph (IKG). IKG* $= (N, R)$ is a heterogeneous Directed Acyclic Graph (DAG) where:

- The nodes of the graph are $N \subset \{n | n \in S \cup D \cup U\}$ where n participates in the change impact propagation. Each node is labelled with its participation role, i.e., *modified* or *contaminated through value/reachability condition*. A *modified* node is not considered *contaminated*, even though the impact of changes to other entities propagates to it.
- The edges of the graph represent the propagation relationships R of the types listed in Table 1 among the nodes in the format (*subject*, *relation*, *object*), capturing relationships within and across modified or affected CDU chains along the data flow path.

Figure 2b provides an example of an IKG that is constructed for the build system in Section 2.2 using the CDU chains from Figure 2a. Note that Figure 2c presents a guide for the notations representing different relationship types in Figure 2b. To construct the IKG, we apply Algorithm 1 on the set of CDU chains computed for each snapshot. Lines 9–28 describe an iterative process through which the IKG expands to cover all the CDU elements that participate in the change impact propagation and the propagation relationships among them, starting from the modified statements, definition points, and use points. The iterative process is repeated until the set of nodes stops expanding. Then, in lines 29–34, the upstream definition points and the statements that define or dereference a node in the IKG are added to the IKG, along with the relationships among them, to provide a complete overview of the change impact throughout the build system. Once we construct the IKGs for each snapshot, we unify them into a single IKG in which the differencing information is made available through references to the AST nodes.

Algorithm 1: IKG construction algorithm

```
1  $IKG = (N_{IKG}, R_{IKG})$;
2  Procedure ExtendIKG($subject, relation, object$)
3      $N_{IKG} \leftarrow N_{IKG} \cup \{subject, object\}$
4      $R_{IKG} \leftarrow R_{IKG} \cup \{(subject, relation, object)\}$
5  Algorithm ConstructIKG()
6      $S_{current} \leftarrow$ {All modified $s_{[c]}$ }
7      $U_{current} \leftarrow$ {All modified $u^{idc}_{s_{[c]}}$ }
8      $D_{current} \leftarrow$ {All modified $d^{idc}_{s_{[c]}}$ }
9      while ($S_{current} \cup U_{current} \cup D_{current} \neq \emptyset$) do
10         foreach ($s_{[c]} \in S_{current}$) do
11             foreach $d^{idc}_{s_{[c]}}$ that $s_{[c]}$ defines do
12                 ExtendIKG($s_{[c]}$, defines, $d^{idc}_{s_{[c]}}$)
13             foreach $s_{[k]}$ for which $s_{[c]}$ is a reachability condition do
14                 ExtendIKG($s_{[c]}$, affects_reachability_of, $s_{[k]}$)
15         foreach ($u^{idc}_{s_{[c]}} \in U_{current}$) do
16             foreach $d^{idk}_{s_{[c]}}$ that $s_{[c]}$ defines using $u^{idc}_{s_{[c]}}$ do
17                 ExtendIKG($u^{idc}_{s_{[c]}}$, is_used_in_definition_of, $d^{idk}_{s_{[c]}}$)
18             foreach $s_{[k]}$ for which $u^{idc}_{s_{[c]}}$ is a reachability condition do
19                 ExtendIKG($u^{idc}_{s_{[c]}}$, affects_reachability_of, $s_{[k]}$)
20         foreach ($d^{idc}_{s_{[c]}} \in D_{current}$) do
21             foreach $u^{idc}_{s_{[k]}}$ that directly dereferences $d^{idc}_{s_{[c]}}$ do
22                 ExtendIKG($d^{idc}_{s_{[c]}}$, is_directly_used_at, $u^{idc}_{s_{[k]}}$)
23             if $d^{idc}_{s_{[c]}}$ is an argument passed to a callable at call site then
24                 foreach $u^{idc}_{s_{[k]}}$ in callable that dereferences $d^{idc}_{s_{[c]}}$ do
25                     ExtendIKG($d^{idc}_{s_{[c]}}$, is_passed_as_argument, $u^{idc}_{s_{[k]}}$)
26         $S_{current} \leftarrow$ {All newly added $s_{[c]}$ in $N_{IKG}$ }
27         $U_{current} \leftarrow$ {All newly added $u^{idc}_{s_{[c]}}$ in $N_{IKG}$ }
28         $D_{current} \leftarrow$ {All newly added $d^{idc}_{s_{[c]}}$ in $N_{IKG}$ }
29     foreach ($u^{idc}_{s_{[c]}} \in N_{IKG} \cap U$) do
30         ExtendIKG($s_{[c]}$, uses, $u^{idc}_{s_{[c]}}$)
31         foreach $d^{idc}_{s_{[k]}}$ that $u^{idc}_{s_{[c]}}$ directly derefernces do
32             ExtendIKG($d^{idc}_{s_{[k]}}$, is_directly_used_at, $u^{idc}_{s_{[c]}}$)
33     foreach ($d^{idc}_{s_{[c]}} \in N_{IKG} \cap D$) do
34         ExtendIKG($s_{[c]}$, defines, $d^{idc}_{s_{[c]}}$)
35     return ($N_{IKG}$, $R_{IKG}$)
```

5 IMPLEMENTATION DETAILS

To evaluate the applicability of BCIA, we develop BuiScout as its prototype. The specific details of BuiScout are influenced by the features of the build technology and the syntax of the language in which build systems are described. We choose to develop BuiScout for *CMake* build systems because they have proven to demand greater maintenance effort [21]. Below, we describe the details that are specific to the implementation of BuiScout. Our online appendix[1] contains an evaluation of BuiScout, where we assess the correctness of its output.

BCIA1. Compute AST Differences

CMake build files are conventionally named `CMakeLists.txt` or have the `.cmake` extension. Influenced by previous studies [12, 63], we rely on these naming conventions to identify build files and their corresponding modifications within change sets.

We compute the AST differences of build files using GumTree [57]. GumTree offers a syntax-aware algorithm designed to compute differences between two ASTs. It provides an interface for parsers to facilitate support for new languages. To parse CMake build files, we implement a CMake parser using Tree-Sitter.[3] Tree-Sitter is a parser generator that consumes a grammar definition and generates a compliant parser. Our CMake grammar is derived from its syntax documentation.[4] We produce GumTree outputs in *DOT* format,[5] which is a text-based markup for graphs. We apply minor changes to GumTree to avoid truncation of node labels.

We post-process the DOT output using NetworkX.[6] This step ensures that changes to a subtree are reflected in its ancestor nodes. For example, when nodes representing a statement within the body of a function are modified, the parent node representing the statement that defines the function is also highlighted as an "*updated*" node to reflect the change.

BCIA2. Compute CDU chains

By computing the CDU chains, we aim to establish connections between the definition and use points. This involves the detection of the correct data flow path on the (1) *system level*, by processing ASTs in the order in which the `CMake` tool would process their corresponding build files, and (2) *file level*, by traversing each AST in the order that the `CMake` tool would traverse the corresponding statements. The `CMake` tool initiates the build process from the main build file, conventionally a `CMakeLists.txt` file in the root directory of the project, and incorporates specifications from other build files in the project as specified, i.e., when they are referenced by the `include`, `add_subdirectory`, and `find_package` commands.

We extend the NetworkX library to support AST-specific operations, such as the traversal of nodes in the order of execution to reproduce the correct data flow paths. We process the ASTs of prior- and post-change versions of the build system independently, beginning with the main build file of the project. As we encounter statements referring to other build files, we resolve them by following the documented behaviour of the referencing statement.[7] We then incorporate the specified build file into the analysis.

The construction of the CDU chains also relies on the characteristics of the CMake language. In CMake, values are assigned to identifiers in a command-sensitive fashion. For example, commands rely on the position of the arguments and keyword argument settings to determine if an argument is a definition or use point. For instance, the `list` command with the `LENGTH` keyword argument specified has the signature: `list(LENGTH <list> <out-var>)` and stores the output (i.e., the length of `<list>`) in the `<out-var>` identifier. The same command with the `APPEND` keyword argument specified follows the signature: `list(APPEND <list> [<element>...])` and stores the output (i.e., the updated list) in its second argument. Referring to the documentation of the latest CMake version,[7] we recognize definition and use points based on the command signatures.

[3]https://tree-sitter.github.io/tree-sitter/
[4]https://cmake.org/cmake/help/v3.0/manual/cmake-language.7.html#syntax
[5]https://graphviz.org/doc/info/lang.html
[6]https://networkx.org/
[7]https://cmake.org/cmake/help/v3.27/index.html

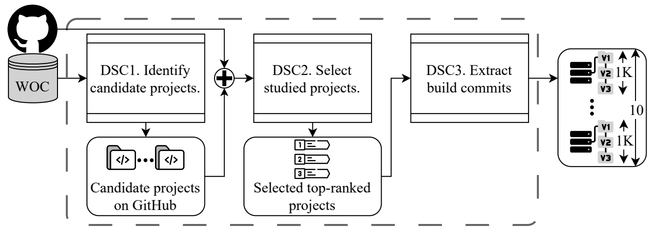

WOC
DSC1. Identify candidate projects.
DSC2. Select studied projects.
DSC3. Extract build commits
Candidate projects on GitHub
Selected top-ranked projects
V1
V2
V3
1K
10

Figure 3: Overview of our process for data set curation.

BCIA3. Construct the IKG

To construct the IKG, we implement Algorithm 1. As described in Section 4, the algorithm is applied to the CDU chains constructed for each snapshot and the two resultant IKGs are unified into a global one, which includes the set of differences. The IKG covers the elements that participate in the change impact propagation as nodes and explicitly models the propagation relationships along the data flow path in the form of directed edges.

Although it is possible to construct the IKG such that it provides an overview of the entire build system, we choose to only include elements that participate in the change impact propagation. This does not harm the inference capabilities of the IKG for its intended purposes and reduces the disk space required to preserve the outputs for further analyses. Our observations show our approach reduces the disk footprint from over 2 gigabytes per commit to an average of 10 megabytes.

6 DATA SET CURATION

To conduct our empirical evaluation of BuiScout (Section 5), we curate a data set of 10,000 change sets. Figure 3 provides an overview of our curation process, which is composed of steps that select (**DSC1**) and rank (**DSC2**) candidate projects, and then extract commits from the top-ranked projects (**DSC3**). Below, we describe each step.

DSC1. Identify candidate projects

We obtain a set of candidate projects from the World Of Code (WOC) data set—a frequently updated corpus of the open-source software ecosystem [64]. Since our prototype supports CMake, candidate projects must use CMake as their build technology. We identify these projects using file naming conventions, i.e., candidate projects must contain a `CMakeLists.txt` file in their root directory. Additionally, we filter out projects that: (1) are forks (because their content is often largely redundant); (2) are archived (because those projects are inactive); and (3) have a build system that is solely authored by one contributor (because we suspect that this sole author has a deep understanding of the build system, and is less likely to need tool support than a build system with multiple authors). 20,143 projects survive this filtering process.

DSC2. Select studied projects

We set out to analyze changes to build systems where their maintenance is a concern for contributors. Therefore, we select subject projects that actively maintain the CMake build specifications. To do so, we rank the candidate projects based on a set of study relevance heuristics that estimate:

Table 2: Overview of selected subject projects.

Project	# BC[a]	# BF[b]	# BA[c]	% RBF[d]
P1: Spectre	3,023	719	49	99.54
P2: Paddle	7,063	467	574	96.54
P3: AliceO2	3,612	500	241	98.19
P4: Krita	8,941	410	256	83.03
P5: MySQL-Server	6,011	443	349	84.03
P6: Qt-Creator	2,336	419	107	87.62
P7: Serenity	4,353	318	297	86.01
P8: Calligra	9,000	388	231	84.38
P9: VXL	6,284	669	124	96.40
P10: Swift	6,344	260	311	94.86

[a] Number of Build Commits
[b] Number of Build Files; in the latest analyzed build commit.
[c] Number of Build Authors.
[d] Percentage of Reached Build Files; average among analyzed build commits in the project.

- ***Activity:*** Projects that actively maintain their build systems have frequent build commits, i.e., change sets that modify build files. However, the frequency of build commits might vary over time. Therefore, instead of the average frequency of build commits, we define the *Recent Build Commit (RBC)* heuristic to characterize build activity in the candidate projects. This heuristic characterizes build maintenance activity based on the prevalence and recency of build commits. More recent commits are assigned a greater weight to give a higher priority to projects with more recent build maintenance activity. Inspired by Mockus and Weiss' [65] formulation of commit recency, we define *RBC* as:
$$RBC = \sum_{c \in C_B} \frac{1}{\Delta_c + 1}$$
where C_B is the set of build commits and Δ_c is the length of the time interval between the build commit date and the date our data collection process began. We measure Δ_c in months, where extra days after the last complete month are discarded.
- ***Size:*** To avoid selecting projects with build systems that are not complex, we consider the number of build files in the latest version of each candidate project to prioritize larger build systems for our analysis. While size is not a direct measure of complexity, prior work [2, 3] shows that the complexity of a build system tends to be strongly correlated with its size.

We use Activity and Size values to rank the candidate projects. To do so, we first rank candidates by Activity and Size independently. When two or more candidates have identical heuristic values, they are assigned the same rank (i.e., ranks are not mutually exclusive). The final rank order is produced by ranking candidates according to the sum of their Activity and Size rank values. From the final ranking, we select the top 10 projects for further analysis.

When selecting the top-ranked projects, we ensure that the order of build file invocations can be automatically inferred. For example, the order of build file invocations cannot be inferred for the *zephyr* project[8] because the paths to the invoked files are dynamically determined based on the value of variables that store information about the machine architecture or the system environment. In such

[8]https://github.com/zephyrproject-rtos/zephyr

a setting, a single build run leaves many of the build files in the project unused as they are specified for a different build setup. Hence, to avoid projects that have build systems with numerous unused build files and to mitigate the impact of technical limitations in resolving file references, we remove the projects in which fewer than 80% of their build files can be automatically inferred and integrated into the build system in their latest 10 build commits. We replace these projects with the next highest-ranked options. A threshold set at 50% would retain 11 of the 47 removed projects. However, we opt for the more rigorous 80% threshold to enhance the accuracy of our measurements. Table 2 shows an overview of the 10 studied projects.

DSC3. Extract build commits

We extract the latest 1,000 build commits from each of the studied projects—a total of 10,000 (1,000×10) studied commits. When selecting the build commits, we ensure that the automatic inference of the order of build file invocations remains consistently over the 80% threshold. Fluctuations in this value are expected as the build system evolves and some build files may become unused but persist until a housekeeping commit deletes them. For example, commit 8785069 from the project *Krita* removes references among build files resulting in instability and a drop in the file reach level.

In our dataset, we also identified six distinct build commits that introduce syntax errors that cause our parser to fail. For example, commit c56085a from the project Swift adds a statement with a syntax error (missing an enclosing quotation mark). The parser inserts error nodes into the AST when it encounters invalid syntax. To ensure our parser is not defective, we inspect each case where an error node is produced. We find that all cases correspond to valid syntax issues, which were addressed in subsequent build commits. Consequently, build commits with syntactically invalid build files are removed from the dataset.

Finally, we observe a growing tendency to use deprecated commands and signatures from older versions of CMake as we process older build commits. For example, commit 039330f from the *VXL* project is when they refactor their build specifications to eliminate the use of the deprecated command `subdirs` and start using the `add_subdirectory` command instead. Since BuiScout does not support commands from CMake versions earlier than `v3.27`, more pronounced use of unsupported commands can lead to inaccurate measurements in our empirical study. Therefore, we restrict our analyses to the latest 1,000 build commits in the selected projects. Thus, from each of the top-10 projects, we extract the latest 1,000 build commits that meet the other specified criteria and date before March 1st, 2024. This eliminates the media-driver project as it has a total of 723 build commits but is ranked highly because its large size (781 build files) compensates for its low RBC measure. We replace this project with the next highest-ranked candidate project.

7 EMPIRICAL STUDY

In this section, we present an empirical study of the impact of changes to build systems. Our goal is to evaluate the applicability of BCIA. We analyze the prevalence of build change sets where their impact propagates transitively (**RQ1**), as well as the characteristics of the incurred impact that capture their extent (**RQ2**). To conduct our study, we apply BuiScout to the 10,000 change sets in our data set. Below, we present our results with respect to each RQ.

RQ1: Impact Prevalence

In this RQ, we explore the prevalence of build change sets where BuiScout uncovers transitive impact propagation. Below, we describe our approach and then present our findings.

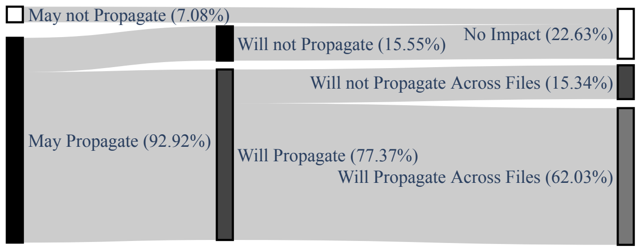
Approach: BuiScout identifies the *impact set* of a given change set—a collection of unmodified CDU chain elements and statements that are affected by the change. We adopt a strict definition to determine what constitutes impact when retrieving an impact set: *an entity is deemed impacted if it is either value- or reachability condition-contaminated and no modifications have been made to the statement containing the entity or its components.* We apply this conservative criterion because when a modification is applied to any component of a statement performing an atomic operation, the statement, as a whole, is expected to produce a different output, and any impact on unmodified locations within the statement is likely expected and intentional.

To identify the impact set, BCIA traces modifications to CDU chain elements and entities that control the reachability of build specifications. As such, for a change set to have a potential impact, its modifications must either involve direct changes to definition or use points (*Condition 1*) or changes to commands that influence the execution path, such as by incorporating build files into the process (*Condition 2*). We label change sets that meet either of these conditions as "*change sets that may propagate*", indicating that they have the necessary characteristics to propagate impact beyond the modified build configurations. Although such change sets can be identified based solely on their modified locations, conducting an impact analysis is required to determine the extent to which their impact propagates.

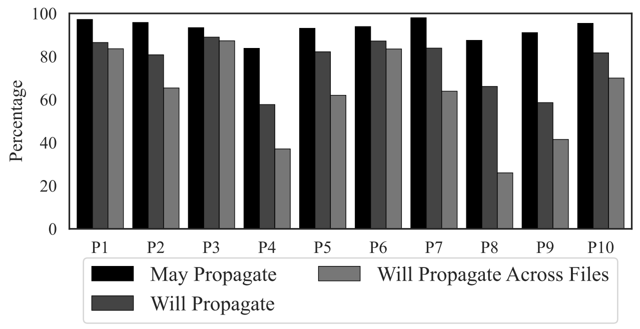
If the analysis of a change set that may propagate an impact yields an impact set, we label it as a *change sets that will propagate.* In such cases, BCIA can reveal the impact of the change. The benefits of using BCIA become more prominent in *change sets that will propagate across files*, where changes in one build file influence another. Tracing the impact of such change sets is challenging as it requires a deep understanding of how build code is interpreted, which our prior work suggests is not common knowledge among developers [12].

To measure the prevalence of these change sets, we identify them using queries that we execute on their IKGs:

- *Change sets that may propagate:* We query for nodes that represent modified definition/use points or modified statements that import blocks of build code into another location, e.g., the `include` command. An IKG containing such nodes indicates that the change set may propagate.
- *Change sets that will propagate:* We extract the impact sets from the IKGs of the change sets that may propagate, adhering to our definition of what constitutes impact. A non-empty impact set indicates that the change affects unmodified build configurations.
- *Change sets that will propagate across files:* We check if both the subject and the object of all relationships in a given IKG reside within the same build file. If they do not, the change set is considered to have an impact that crosses files.


May not Propagate (7.08%)
Will not Propagate (15.55%)
No Impact (22.63%)
Will not Propagate Across Files (15.34%)
May Propagate (92.92%)
Will Propagate (77.37%)
Will Propagate Across Files (62.03%)

(a) General perspective.



(b) Project-specific perspective.

Figure 4: Prevalence of change sets that may propagate, will propagate, and will propagate across files.

Results: Figure 4 provides a summary of our results. We structure the rest of this section around our key observations.

Observation 1. The majority of build change sets propagate their impact, often across files. Figure 4a provides an overview of the studied change sets. We identify 9,292 change sets that may propagate in our data set of the 10,000 studied build commits. Of these, 9,226 directly modify CDU chain elements (condition 1) and 2,934 modify commands that influence the execution path (condition 2). Among the change sets that may propagate, 83.27% have an impact that does propagate, which corresponds to 77.37% of the studied change sets. Furthermore, 80.17% of the change sets with an impact that propagates (62.03% of the studied change sets) have an impact that crosses files. We also find that in 97.79% of the change sets with an impact that crosses files, the modifications affect locations in unmodified build files, accounting for 60.66% of the studied change sets.

Figure 4b shows the prevalence of change sets in each category per project. Overall, a large proportion of change sets (83.80%–98.00%, σ = 4.42) may propagate an impact; however, the proportions of change sets that do propagate their impact to unmodified build configurations vary between projects (σ = 11.92). We observe an even broader range of the proportions of change sets that propagate their impact across files (σ = 21.04). For example, in the AliceO2 project (P3), 95.29% of the change sets with an impact that may propagate do indeed propagate to unmodified build configurations, of which 98.09% propagate their impact across files. In contrast, for the Calligra project (P8), these values drop to 75.54% and 39.33%, respectively. Despite being lower, these ratios remain non-negligible, with more than half of the studied change sets in every project (minimum of 57.70%) affecting unmodified build configurations. These quantities highlight the pervasiveness of change sets with an impact that propagates, demonstrating that BuiScout will frequently expose results beyond the visible change set.

Next, we strive to understand why in some cases BuiScout does not detect any impact that propagates from the change sets with the potential for it. To do so, we randomly inspect and label such change sets until we satisfy our saturation criterion, i.e., when no new reasons have emerged for 50 consecutive change sets. This criterion was satisfied after inspecting 73 change sets.

Observation 2. When a change set that may propagate an impact yields no effect, it is often because the impact cannot propagate beyond the modified locations. In 76.71% of the inspected change sets, the modifications span all involved definitions and uses of the changed identifiers, and hence, cannot propagate their impact beyond the modification location.[9] In 21.92% of the inspected change sets, the modifications affect call sites to external APIs.[10] These API calls influence the build processes in ways that are not immediately apparent because their definition does not reside within the studied build system. As such, assessing the impact of modifications to such call sites would require a cross-project analysis, which is currently unsupported in our prototype implementation. Additionally, we only observed one other change set where an impact did propagate; however, it was not detected due to limitations in our prototype implementation, specifically, resolving the execution order of build files.[11] This error occurs when a build file is not explicitly loaded into the build process. Such a case is uncommon in our data set because BuiScout successfully incorporates over 80% of the build files in our studied build systems into its analysis, with an average of 91.04%. Regardless, to prevent such issues in practice, our prototype supports mechanisms where build maintainers can configure specific build files to be loaded into the build process at selected locations in the build system.

Moreover, our analysis reveals implementation and modification patterns specific to projects, which elucidate our divergent project-specific results. For example, in the VXL project (P9), developers frequently use the `AUX_SOURCE_DIRECTORY` command. According to its documentation,[12] developers must modify the CMake file containing this command to activate its automatic source file detection feature. To trigger this behaviour, developers in the VXL project modify a variable that is immediately overwritten by the `AUX_SOURCE_DIRECTORY` command.[13] Therefore, such change sets do not technically propagate their impact. This helps to explain why the impact of change sets in the VXL project does not propagate as often as the other studied projects.

Conclusion of RQ1. *Our prototype can detect an impact propagating to unmodified parts of the build specification in 77.37% of the* 10,000 *studied change sets.*

[9]See commit 83e13c1 in the Calligra project for an example.
[10]See commit 90a8496 in the Qt-Creator project for an example.
[11]See commit 48b6cd5 in the Swift project for an example.
[12]https://cmake.org/cmake/help/v3.27/command/aux_source_directory.html
[13]See commit 76a4374 in the VXL project for an example.

RQ2: Impact Characteristics

In this RQ, we characterize the impact of the studied build change sets. Below, we describe our approach, followed by our results.
Approach: We characterize the impact of changes to build systems in terms of the magnitude and breadth of the impact.

Magnitude. To estimate the magnitude of the impact of a change set, we measure the cardinality of the impact set in terms of *concrete commands*, i.e., commands that are identifiable by unique AST nodes that represent statements. Concrete commands must include the statement identifier and the arguments passed to it, and may span over more than one line of code. More specifically, the cardinality of the impact set is the number of unique AST nodes that are associated with at least one statement that the set contains.

We measure the magnitude of impact in this fashion because it consolidates multiple statement instances that are associated with the same concrete command. For example, a command within the body of a function is executed every time the function is called. This results in several instances of the same command with varying reachability conditions and argument values. For maintenance and code review purposes, these instances are all viewed at once. Thus, we count them once to avoid inflating their perceived magnitude.

Breadth. To estimate the breadth of the impact, we first label the concrete commands in the impact set as either (a) localized, i.e., those that appear within the build files that the change set modifies; or (b) dispersed, i.e., those that appear in build files that are not included in the studied change set. We suspect that the detection of the incurred impact on build configuration is more challenging for dispersed concrete commands because it requires the reader to recognize that unmodified build files are being impacted. Note that a subset of the change sets that will propagate across files (RQ1) have dispersed concrete commands in their impact set, whereas any change set may have localized concrete commands.

We estimate the breadth of the incurred impact of a change set using $B = \frac{N_D}{N_D+N_L}$ where N_L and N_D are the number of affected localized and dispersed concrete commands, respectively. A breadth of zero indicates that all affected concrete commands appear in modified build files, whereas a breadth of one indicates that all affected concrete commands appear in unmodified build files. The latter may happen, for example, when the modifications are made to a statement that defines a variable for use in other build files.[14]

Interplay of Magnitude and Breadth. The independent analyses of magnitude and breadth do not capture their potential interaction. Our breadth measurement normalizes against the magnitude of impact sets to control for its potentially confounding impact. To study the extent to which magnitude and breadth interact, we measure the Spearman Rank correlation, which detects the non-linear (monotonic) relationships between two variables [66].
Results: Figures 5 and 6 and Table 3 summarize our results. We structure the rest of this section around our key observations.

Observation 3. The studied impact sets consist of a non-negligible number of affected concrete commands. Figure 5 shows the distribution of the magnitude of the impact sets that we measure from each project. We observe a median overall magnitude of 14 concrete commands, with project-specific medians ranging between 4–27 concrete commands. In extreme cases, the magnitude of impact sets can reach 4,245 concrete commands. These observed magnitude tendencies suggest that common practitioner perceptions about the simplicity of changes to build systems [12] may be a misconception. In our online appendix,[1] we present an investigation of the change types that result in the largest impact sets.

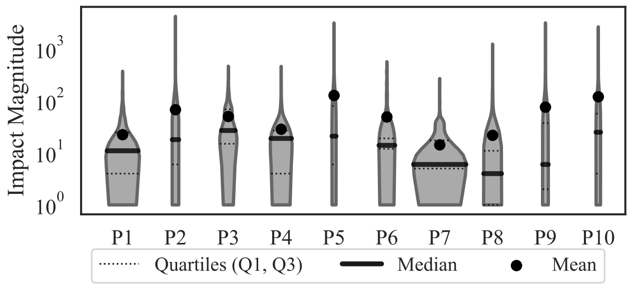


Figure 5: The distribution of the magnitude of impact sets.

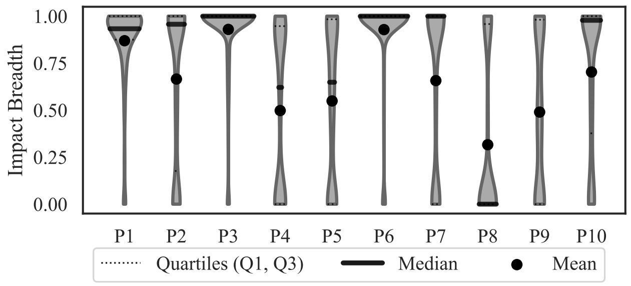


Figure 6: The distribution of the breadth of impact sets.

Observation 4. The impact sets are predominantly characterized by large breadths. Specifically, the median measured breadth is 0.955, indicating that in 50% of impact sets, 95.55% of the concrete commands affected by the change are not local, and are thereby not immediately visible to developers. Extreme cases where the breadth is zero or one are not uncommon. In 21.60% (45.97%) of the impact sets, concrete commands affected by the change are entirely local (non-local), having a breadth of zero[15] (one[16]).

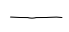
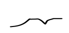
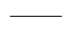
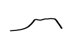
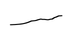
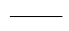
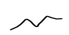
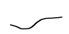
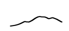
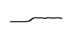
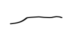
Figure 6 provides an overview of the distribution of the breadth of impact sets. We observe substantial variations in the median breadth across projects, ranging between 0.5–1. The Calligra project (P8) stands out with a distinctly lower density of impact sets with higher breadth, such that the median breadth drops to zero. This is likely due to the especially low percentage of change sets (26.00%) that propagate their impact across files. As a result, in 60.67% of the non-empty impact sets in P8, the incurred impact is entirely localized.

Observation 5. Magnitude and breadth are only weakly correlated, indicating that a small impact does not imply that the concrete commands are localized. Table 3 provides an overview

[14]See commit fc24909 in project Calligra for an example.

[15]See commit f9c5c45 in the Swift project for an example.
[16]See commit 338c104 in the Krita project for an example.

Table 3: The correlations between magnitude and breadth.

Project	**Median**		**Spearman ρ ($\alpha = 0.05$)**		**Trend**[a]
	Mgn.	**Brd.**			
P1	11	0.933	+0.054	($p = 0.11 > \alpha$)	
P2	18	0.957	+0.313	($p << \alpha$)	
P3	27	1.000	+0.092	($p < \alpha$)	
P4	19	0.621	+0.492	($p << \alpha$)	
P5	21	0.649	+0.343	($p << \alpha$)	
P6	14	1.000	+0.099	($p < \alpha$)	
P7	6	1.000	+0.038	($p = 0.28 > \alpha$)	
P8	4	0.000	-0.002	($p = 0.96 > \alpha$)	
P9	6	0.500	+0.463	($p << \alpha$)	
P10	25	0.978	+0.177	($p < \alpha$)	
Overall	14	0.955	+0.272	($p << \alpha$)	

[a] LOWESS (Locally Weighted Scatterplot Smoothing) [67], the vertical axis represents the breadth and the horizontal axis represents the magnitude (log scale).

of the correlations between magnitude and breadth across the studied projects. Although a statistically significant positive correlation exists overall, with a weak magnitude ($\rho = 0.272$), this relationship is not uniform across the studied projects. Specifically, the Spearman rank correlation coefficient shows no statistical significance in three of the ten studied projects (P1, P7, and P8). In contrast, the remaining seven projects show a statistically significant positive correlation between magnitude and breadth, indicating that impact sets with a larger magnitude tend to also have a larger breadth. The strength of these correlations varies from very weak (P3, P6, and P10) or weak (P2 and P5) to moderate (P4 and P9).

The observed magnitudes and breadths highlight the complexity of the impact that propagates from changes in build systems. For example, in commit 75f0e3c6 from the Swift project, a seemingly small change in the build specifications transitively propagates its impact to 167 concrete commands and reaches 4 concrete commands that define or update deliverables. Out of the 167 affected concrete commands, only one is local to the change, and the remaining 166 are scattered across five unmodified build files.

Conclusion of RQ2. *The studied impact sets affect a median of 14 concrete commands where that impact propagates non-locally in a median of 95.55% of cases. The magnitude and breadth of the impact of changes share a weak positive correlation, suggesting that changes with a small magnitude of impact can still propagate in a broad fashion.*

8 THREATS TO VALIDITY

Below, we present the threats to the validity of our study.
Internal Validity: BCIA is a static approach to detecting the impact of a change set. As such, it does not capture the impact that propagates due to dynamic programming features. For example, in CMake, commands within the build system may produce side effects by executing shell commands [68]. Such side effects may propagate an impact that is only detectable at runtime. As a result, change sets may induce a dynamic impact that substantially affects the build process. While BCIA lacks the ability to identify such an impact, it can help avoid a postmortem detection of unintended build behaviour due to a statically detectable impact only after a potentially costly build run.
Construct Validity: We define impact such that any command that is affected by the change and participates in propagating its impact is treated equally. However, impact may not be equally distributed, e.g., an impacted `message` command may not be as great of a concern as an impacted `add_executable` command. Therefore, our uniform measurement of magnitude and breadth may not accurately represent the complexity of the incurred impact. However, assigning arbitrary weights to the affected commands could introduce a subjective bias, reflecting our own perception of the importance of a command within the build process.
External validity: We evaluate BCIA using BuiScout on a high-quality dataset since we select 10 large open-source projects with large actively maintained build systems across 10,000 build changes. Our data set represents projects with complex build systems where understanding changes in the build system may be challenging.

9 CONCLUSION AND IMPLICATIONS

In this paper, we propose BCIA—an approach to uncover the impact of change sets on the build system. We also develop BuiScout—a prototype of BCIA for CMake build systems. Using BuiScout, we conduct an empirical study on 10,000 change sets that modify CMake build systems, which focuses on the frequency and extent of the impact that propagates from these change sets.

Below, we distill the key takeaway messages of our study.
Recognizing how the impact of build changes propagates is key, but requires a deep understanding of build systems. Our results show that in 77.37% of change sets, an impact propagates to unmodified build configurations (Observation 1), with a considerable median magnitude of 14 commands being affected (Observation 3), and with 95.55% of these commands appearing in unmodified files (Observation 4). Without a deep understanding of the design and implementation of the build system, it would be challenging for developers to recognize these implications and understand their interactions. As practitioners often lack detailed insight into the intricacies of build systems, raising awareness about the impact of build changes is important.
Researchers and tool developers have various opportunities to further support the understanding of changes to build systems. While Observations 1, 3, and 4 highlight the applicability and usefulness of techniques that detect the impact of changes to build systems, Observation 2 points out challenges that tool developers should consider when designing such techniques. For example, detecting how impact propagates across API calls in build systems requires cross-project analysis. Furthermore, the variability that we observe in implementation patterns across projects opens further opportunities for future work to study the impact of these patterns on the maintainability of the build system.

REFERENCES

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The Evolution of the Linux Build System. *Electronic Communications of the EASST (ECEASST)*, 8(0), 2008.

[2] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The Evolution of ANT Build Systems. In *the Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 42–51, 2010.

[3] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The Evolution of Java Build Systems. *Empirical Software Engineering (EMSE)*, 17(4):578–608, 2012.

[4] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An Empirical Study of Build Maintenance Effort. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 141–150, 2011.

[5] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering (EMSE)*, 20(6):1587–1633, 2015.

[6] Nachiappan Nagappan and Thomas Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *the Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 364–373, 2007.

[7] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for Build Debt: Experiences Managing Technical Debt at Google. In *the Proceedings of the International Workshop on Managing Technical Debt (MTD)*, pages 1–6, 2012.

[8] Ying Zhang, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Ping Yu. ABC: Accelerated Building of C/C++ Projects. In *the Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 182–189, 2015.

[9] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' Build Errors: A Case Study (at Google). In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 724–734, 2014.

[10] Lorin Hochstein and Yang Jiao. The Cost of the Build Tax in Scientific Software. In *the Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 384–387, 2011.

[11] Sarah Nadi and Ric Holt. Make It or Break It: Mining Anomalies from Linux Kbuild. In *the Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 315–324, 2011.

[12] Mahtab Nejati, Mahmoud Alfadel, and Shane McIntosh. Code Review of Build System Specifications: Prevalence, Purposes, Patterns, and Perceptions. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1213–1224, 2023.

[13] Shaun Phillips, Thomas Zimmermann, and Christian Bird. Understanding and Improving Software Build Teams. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 735–744, 2014.

[14] Yuan Huang, Nan Jia, Xiangping Chen, Kai Hong, and Zibin Zheng. Salient-Class Location: Help Developers Understand Code Change in Code Review. In *the Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 770–774, 2018.

[15] Quinn Hanam, Ali Mesbah, and Reid Holmes. Aiding Code Change Understanding with Semantic Change Impact Analysis. In *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 202–212, 2019.

[16] Fernanda M. Delfim, Lilian P. Scatalon, Jorge M. Prates, and Rogério E. Garcia. Visual Approach for Change Impact Analysis: A Controlled Experiment. In *the Proceedings of the International Conference on Information Technology - New Generations (ITNG)*, pages 391–396, 2015.

[17] Siyuan Jiang, Collin McMillan, and Raul Santelices. Do Programmers Do Change Impact Analysis in Debugging? *Empirical Software Engineering (EMSE)*, (2): 631–669, 2017.

[18] Mehran Meidani, Maxime Lamothe, and Shane McIntosh. Assessing the Exposure of Software Changes: The DiPiDi Approach. *Empirical Software Engineering (EMSE)*, 28(2):41, 2023.

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

[20] Nikhil Parasaram, Earl T. Barr, and Sergey Mechtaev. Rete: Learning Namespace Representation for Program Repair. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1264–1276, 2023.

[21] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering (EMSE)*, 20(6):1587–1633, 2015.

[22] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Build Code Analysis with Symbolic Evaluation. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 650–660, 2012.

[23] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. Fault Localization for Build Code Errors in Makefiles. In *the Companion Proceedings of the International Conference on Software Engineering (ICSE-Companion)*, pages 600–601, 2014.

[24] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. An Empirical Study of Unspecified Dependencies in Make-based Build Systems. *Empirical Software Engineering (EMSE)*, 22(6):3117–3148, 2017.

[25] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the Dependency Conflicts in My Project Matter? In *the Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 319–330, 2018.

[26] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. Could I Have a Stack Trace to Examine the Dependency Conflict Issue? In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 572–583, 2019.

[27] Christian Macho, Fabian Oraze, and Martin Pinzger. DValidator: An Approach for Validating Dependencies in Build Configurations. *Journal of Systems and Software (JSS)*, 209:111916, 2024.

[28] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically Repairing Dependency-related Build Breakage. In *the Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 106–117, 2018.

[29] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1078–1089, 2018.

[30] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. History-Driven Build Failure Fixing: How Far Are We? In *the Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 43–54, 2019.

[31] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. A Model for Detecting Faults in Build Specifications. *the Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[32] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph. In *the Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 463–474, 2020.

[33] Bram Adams, Herman Tromp, Kris de Schutter, and Wolfgang de Meuter. Design Recovery and Maintenance of Build Systems. In *the Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 114–123, 2007.

[34] Carlene Lebeuf, Elena Voyloshnikova, Kim Herzig, and Margaret-Anne Storey. Understanding, Debugging, and Optimizing Distributed Software Builds: A Design Study. In *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 496–507, 2018.

[35] Ryan Hardt and Ethan V. Munson. An Empirical Evaluation of Ant Build Maintenance Using Formiga. In *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 201–210, 2015.

[36] Jim Buffenbarger. Adding Automatic Dependency Processing to Makefile-Based Build Systems with Amake. In *the Proceedings of the International Workshop on Release Engineering (RELENG)*, pages 1–4, 2013.

[37] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution. *Empirical Software Engineering (EMSE)*, 25(3):2218–2257, 2020.

[38] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. The Nature of Build Changes: An Empirical Study of Maven-Based Build Systems. *Empirical Software Engineering (EMSE)*, 26(3):32, 2021.

[39] Bohner. Impact analysis in the software change process: a year 2000 perspective. In *the Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 42–51, 1996.

[40] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A Survey of Code-Based Change Impact Analysis Techniques. *Software Testing, Verification and Reliability (STVR)*, 23(8):613–646, 2013.

[41] Zijian Jiang, Ye Wang, Hao Zhong, and Na Meng. Automatic Method Change Suggestion to Complement Multi-Entity Edits. *Journal of Systems and Software (JSS)*, 159:110441, 2020.

[42] Zijian Jiang, Hao Zhong, and Na Meng. Investigating and Recommending Co-Changed Entities for JavaScript Programs. *Journal of Systems and Software (JSS)*, 180:111027, 2021.

[43] Daihong Zhou, Yijian Wu, Xin Peng, Jiyue Zhang, and Ziliang Li. Revealing Code Change Propagation Channels by Evolution History Mining. *Journal of Systems and Software (JSS)*, 208:111912, 2024.

[44] Binish Tanveer, Anna Maria Vollmer, and Ulf Martin Engel. Utilizing Change Impact Analysis for Effort Estimation in Agile Development. In *the Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 430–434, 2017.

[45] Binish Tanveer, Anna Maria Vollmer, Stefan Braun, and Nauman bin Ali. An Evaluation of Effort Estimation Supported by Change Impact Analysis in Agile Software Development. *Journal of Software: Evolution and Process (JSEP)*, 31(5): e2165, 2019.

[46] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *the Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 432–448, 2004.

[47] Ye Wang, Na Meng, and Hao Zhong. An Empirical Study of Multi-entity Changes in Real Bug Fixes. In *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 287–298, 2018.

[48] Maria Kretsou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Ignatios Deligiannis, and Vassilis C. Gerogiannis. Change impact analysis: A systematic mapping study. *Journal of Systems and Software (JSS)*, 174:110892, 2021.

[49] Leon Moonen, David Binkley, and Sydney Pugh. On Adaptive Change Recommendation. *Journal of Systems and Software (JSS)*, 164:110550, 2020.

[50] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Associating Code Clones with Association Rules for Change Impact Analysis. In *the Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 93–103, 2020.

[51] Peng Dai, Yawen Wang, Dahai Jin, Yunzhan Gong, and Wenjin Yang. An improving approach to analyzing change impact of C programs. *Journal of Computer Communications (ComputCommun)*, 182:60–71, 2022.

[52] Shane Mcintosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. Mining Co-Change Information to Understand When Build Changes Are Necessary. In *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 241–250, 2014.

[53] Christian Macho, Shane McIntosh, and Martin Pinzger. Predicting Build Co-changes with Source Code Change and Commit Categories. In *the Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 541–551, 2016.

[54] Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E. Hassan. Cross-Project Build Co-Change Prediction. In *the Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 311–320, 2015.

[55] 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 *The Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 150–159, 2012.

[56] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. In *the Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 188–197, 2004.

[57] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-Grained and Accurate Source Code Differencing. In *the Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 313–324, 2014.

[58] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering (TSE)*, 33(11):725–743, 2007.

[59] Georg Dotzler and Michael Philippsen. Move-Optimized Source Code Tree Differencing. In *the Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 660–671, 2016.

[60] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. Generating Accurate and Compact Edit Scripts Using Tree Differencing. In *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 264–274, 2018.

[61] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. ClDiff: generating concise linked code differences. In *the Proceedings of the International Conference on Automated Software Engineering (ASE)*, page 679–690, 2018.

[62] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge Graphs. *ACM Computing Surveys*, 54(4):1–37, 2021.

[63] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When Testing Meets Code Review: Why and How Developers Review Tests. In *the Proceedings of the International Conference on Software Engineering (ICSE)*, pages 677–687, 2018.

[64] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering (EMSE)*, 26(22):1–42, 2021.

[65] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[66] Jerrold H. Zar. Significance Testing of the Spearman Rank Correlation Coefficient. *Journal of the American Statistical Association (JASA)*, 67(339):578–580, 1972.

[67] William S. Cleveland. Robust Locally Weighted Regression and Smoothing Scatterplots. *Journal of the American Statistical Association (JASA)*, 74(368): 829–836, 1979.

[68] KimHao Nguyen, ThanhVu Nguyen, and Quoc-Sang Phan. Analyzing the CMake Build System. In *the Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 27–28, 2022.