Automated Dependency Optimization for Artifact-Based Build Systems

HONGXU XU, University of Waterloo, Canada
ZHENYANG XU, University of Waterloo, Canada
SHANE MCINTOSH, University of Waterloo, Canada
CHENGNIAN SUN, University of Waterloo, Canada

As projects grow, the maintenance of intra- and inter-project dependencies becomes increasingly complex. If dependency maintenance is lax, redundant dependencies may accrue, inflating incremental build and test latencies. The heterogeneity of language- and tool-specific dependency expressions and the complexity of the dependency graphs that they specify exacerbate the challenge of identifying and removing redundant dependencies. To address these challenges, this paper introduces DepReduce, an automated approach for optimizing declared dependencies in artifact-based build systems. DepReduce operates directly on the dependency graph managed by the underlying build system, and formalizes the optimization objective as minimizing the cumulative rebuild cost triggered by changes to individual targets. To achieve this, DepReduce performs the *dependency lifting* and *dependency flattening* operations on the dependency graph in topological order, which we prove is both correct and optimal under the defined optimization objective.

To empirically evaluate the approach, we implemented BazelDepReduce, an automated dependency optimization tool for Bazel. Bazel is an artifact-based build system with native support for multiple programming languages. We evaluated BazelDepReduce on 19 open-source Bazel projects written in seven programming languages. Among them, 16 projects across six languages achieved reductions in rebuild cost. In total, BazelDepReduce identified and removed 430 redundant dependencies, which we used to produce 16 Pull Requests (PRs). Twelve PRs have been merged by the target projects, including Angular and Apache RocketMQ, affecting up to 80.6% of subsequent commits, with a median of 26.3%. We also adapted BazelDepReduce to support Buck and Cargo, providing preliminary evidence that the implementation can be extended to other artifact-based build systems. Overall, these results show that our approach can effectively reduce rebuild cost on selected Bazel projects spanning multiple languages.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Maintaining software**.

Additional Key Words and Phrases: dependency optimization, dependency graph, build system

1 Introduction

Build systems serve as the backbone of software development, transforming source code into deliverables by interpreting configuration files and executing command sequences, such as order-dependent compilation and testing commands. The cadence of software delivery is constrained by the pace at which builds can be processed. Incrementality and parallelism are two critical capabilities that span most build systems and directly impact build performance. Incremental builds ensure that only artifacts affected by changes are rebuilt, while parallelism enables independent tasks to execute concurrently [37]. These capabilities are particularly vital in large-scale projects [16], especially during iterative development where frequent rebuilds are the norm [24, 32]. Effective dependency management is a prerequisite to leveraging these capabilities. A major challenge is avoiding redundant dependencies (declared but unused), which hinder both incrementality and parallelism [21, 38].

Accepted at ISSTA 2026.
Authors' Contact Information: Hongxu Xu, University of Waterloo, Waterloo, Canada, hongxu.xu@uwaterloo.ca; Zhenyang Xu, University of Waterloo, Waterloo, Canada, zhenyang.xu@uwaterloo.ca; Shane McIntosh, University of Waterloo, Waterloo, Canada, shane.mcintosh@uwaterloo.ca; Chengnian Sun, University of Waterloo, Waterloo, Canada, cnsun@uwaterloo.ca.

Despite a broad recognition of the importance of preventing redundancies, build systems tend to degrade over time. Build system specifications tend to grow unless intentional refactoring effort is invested [31], consistent with Lehman's laws of software evolution, which state that software must be continually adapted to remain useful and that complexity increases unless work is done to maintain or reduce it [26, 27]. Indeed, preventing redundancies is non-trivial, especially as projects grow in complexity [20]. In practice, developers often introduce redundancies inadvertently by copying build configurations [34].

Although prior work has addressed the related problem of detecting dependency issues including redundancies [13, 21, 28, 29, 39, 42], they exhibit several limitations. First, most prior work does not support automatic repair. Repair of redundancies is non-trivial, as naïvely removing redundancies does not guarantee optimal builds. Second, many existing tools are tailored to specific languages or are incompatible with artifact-based build systems. For example, DepClean [39] provides initial automated repair support, but it is specific to build systems specified in Maven.

We address these limitations with DepReduce, an automated approach for optimizing declared dependencies in artifact-based build systems. Our method does not infer dependencies from language semantics or file-access traces like prior work [21, 28, 29]. Instead, DepReduce operates on the declared dependency graph. We first formalize the objective of dependency optimization as minimizing the cumulative rebuild cost triggered by changes to individual targets, subsequently deriving an optimal strategy based on topological ordering. To realize this strategy, DepReduce systematically prunes redundant dependencies while applying its *dependency lifting* and *dependency flattening* operations. These operations strategically introduce dependencies to unlock opportunities for removing others that incur higher rebuild costs. For instance, to remove a dependency from target A to B, DepReduce may lift B as a dependency of all targets that depend on A. Additionally, DepReduce may flatten target B in A (i.e., add B's dependencies directly to A). Throughout this process, DepReduce verifies correctness by incrementally building and testing the project after each modification. In practice, this serves as a pragmatic validation oracle rather than a complete guarantee of semantic soundness.

We implemented DepReduce as BazelDepReduce, an automated dependency optimization tool for Bazel. Bazel is an artifact-based build system with multi-language support [2]. Since DepReduce operates on the declared dependency graph, BazelDepReduce can in principle be applied across Bazel-managed targets written in different languages without requiring language-specific customization.

During implementation, we found that overly aggressive pruning can introduce unintended architectural changes, such as violations of strict dependency principles [4, 18, 36] and flattening of alias targets that support build comprehension. Thus, to counteract overly aggressive pruning in BazelDepReduce, we retain transitively accessible dependencies, ignore targets that have no source file, and provide a filtering mechanism to exclude specific targets from optimization. To explore how the proposed technique generalizes to other build systems, we also adapted BazelDepReduce to support Buck, another multi-language build system, and Cargo, the Rust-specific build system.

Our evaluation of DepReduce addresses the following two research questions (RQs):

RQ1 *How **effectively** does DepReduce reduce rebuild costs in real-world projects?*
We first applied BazelDepReduce to 19 open-source Bazel projects, where 16 achieved measurable reductions in rebuild costs. In total, 430 redundant dependencies were removed. Notably, 74 of these removals were enabled by 70 dependency lifting and 34 dependency flattening operations. At the time of writing, twelve out of 16 optimization Pull Requests (PRs) have been accepted, including contributions to prominent open-source projects, such as Angular [7] and Apache RocketMQ [11]. Beyond optimization, our case studies reveal

that BazelDepReduce aids in detecting latent build script errors, such as duplicated main functions and incorrect globbing patterns.

RQ2 *How **efficiently** does DepReduce reduce rebuild costs in real-world projects?*
The execution times of BazelDepReduce across the evaluated projects ranged from minutes to hours. Since we envision that BazelDepReduce would be applied on a scheduled rather than continuous basis (e.g., weekly rather than for every submitted PR), we believe that these observed latencies would be tolerable in a practical setting.

Contributions. In our estimation, the following are the main contributions of this paper:

- The formulation of DepReduce: to the best of our knowledge, the first automated approach for optimizing declared dependencies in artifact-based build systems (§ 3). We formally defined the optimization problem, introduced *dependency lifting* and *dependency flattening* operations (§ 3.2), and proved the correctness and optimality of our core strategy (§ 3.3).
- The implementation of BazelDepReduce: a practical automated dependency optimization tool for Bazel (§ 4). We also adapted BazelDepReduce to support Buck and Cargo, showing that the implementation can be extended to other artifact-based build systems. The implementation is open-source and freely available at https://github.com/xuhongxu96/depreduce.
- An empirical evaluation of BazelDepReduce spanning 19 open-source projects (§ 5): the results show rebuild cost reductions in 16 projects, with 430 removed dependencies and twelve merged pull requests.

2 Background

In this section, we first briefly describe build system paradigms. Next, we discuss common dependency errors, and existing techniques for detecting them. Lastly, we provide a formal definition of the dependency graph model used in this paper.

2.1 Build System Paradigms

Build systems encompass diverse design paradigms and admit multiple classification criteria [33, 44]. In this work, we characterize them along dependency granularity and language support dimensions. First, regarding granularity, traditional tools like Make [22] operate at the file level, whereas artifact-based build systems such as Bazel [2] function at the target[1] (or artifact) level. Regarding language support, build systems may be ecosystem-centric, i.e., tailored for specific languages like Maven [25] and Cargo [8], or designed to accommodate multiple languages within a single project. We focus our discussion and optimization techniques on artifact-based, multi-language build systems which have proven effective for large-scale software development [35], with a particular focus on Bazel [2].

```
cc_library(
    name = "foo",
    srcs = ["foo.cpp"],
    hdrs = ["foo.h"],
    includes = ["."],
)
```

```
cc_binary(
    name = "main",
    srcs = ["main.cpp"],
    deps = [":foo"],
)
```

Fig. 1. Example Bazel build script. Target `main` declares a dependency on target `foo` (highlighted in blue).

[1]Since artifact-based build systems abstract the build artifacts as build targets, we use the terms *artifact* and *target* interchangeably in this paper.

2.1.1 Artifact-Based Build Systems. Artifact-based build systems operate by managing dependencies at the granularity of abstract targets that correspond to build artifacts. This paradigm decouples the build definition from platform-specific details, allowing developers to focus on the logical architecture of the software rather than low-level file operations or imperative build tasks [33, 44]. In this model, each build artifact (e.g., a library or binary) is encapsulated as a target, and dependencies are explicitly declared between them. Prominent examples of this category include Bazel [2] and Buck [3]. Figure 1 provides an example of how dependencies are defined between a binary and a library target in a Bazel script.

2.1.2 Multi-Language Build Systems. Unlike ecosystem-centric build systems tailored to a single programming language (e.g., Maven [25] for Java and Cargo [8] for Rust), multi-language build systems are engineered to manage repositories containing source code in multiple programming languages. Build systems such as Bazel [2] and Buck [3] achieve this by decoupling the build execution from language-specific logic. In this architecture, the build system manages the dependency graph and artifacts, and performs build execution in a language-agnostic manner. Language-specific build instructions are encapsulated in rules (e.g., `java_library`, `cc_library`) [1, 14]. Crucially, these build systems construct a unified, heterogeneous dependency graph where targets written in one language can directly depend on targets of another. For example, a Python library target can explicitly declare a dependency on a Rust library target. This unified view enables the build system to enforce correctness and maintain incrementality across language boundaries.

2.2 Types of Dependencies

Accurate dependency declaration is critical for the correctness and efficiency of build systems. Deviations from the actual dependency requirements can lead to missing dependencies (under-declaration) and redundant dependencies (over-declaration). These are among the most common types of errors in build systems [21].

In this section, we first define declared and actual dependencies, then characterize missing and redundant dependencies. While these concepts are universally applicable to artifact-based build systems, here we focus on their specific effects on build correctness and incrementality in Bazel.

2.2.1 Declared Dependencies. A *declared dependency* is a dependency edge explicitly specified in the build configuration, such as a `deps` entry in a Bazel `BUILD` file. For example, in Figure 1, the `deps` entry for `main` declares a dependency on `foo`.

2.2.2 Actual Dependencies. An *actual dependency* is a dependency that is required for the build to succeed. Notably, unused include or import statements in source code can constitute actual dependencies, and eliminating them is out of scope of this paper, as it is a code-level issue rather than a build-level one. For example, if `main.cpp` includes `foo.h`, then `main` has an actual dependency on `foo`, as the build will fail if the dependency containing `foo.h` is removed even though no types or functions from `foo.h` are used in `main.cpp`.

2.2.3 Missing Dependencies. A missing dependency occurs when a target requires an input artifact that is not explicitly declared in its build configuration, which can lead to unsound incremental builds. This issue is prevalent in file-level build systems, where implicit file access allows builds to succeed coincidentally, often leading to reproducibility problems [21].

In contrast, Bazel effectively precludes potential missing dependencies through its sandboxing mechanism [12]. By executing build actions in isolated environments (a hermetic sandbox), Bazel ensures that only explicitly declared dependencies are accessible during the build process. If a dependency is not declared, it is physically absent from the sandbox, resulting in a deterministic build failure. Consequently, Bazel ensures that all successful builds have fully specified dependencies.

2.2.4 Missing Direct Dependencies. A missing direct dependency (a.k.a., a strict dependency violation) occurs when a target consumes artifacts from a transitive dependency without explicitly declaring it as a direct dependency [4, 18, 36]. Unlike standard missing dependencies, these often allow the build to succeed initially, but introduce fragility into the build process. Consider a scenario where target A depends on B, and B depends on C. If A consumes artifacts from C without declaring C as a direct dependency, it implicitly relies on C through B. If B is later refactored to remove its dependency on C, the build for A will fail, despite no changes being made to A itself. This issue is particularly problematic when relying on third-party libraries, where upgrading to a new version may inadvertently remove dependencies that were previously accessible transitively.

Bazel projects are prone to this issue because the sandboxing mechanism does not prevent access to transitive dependencies. While Bazel enforces strict dependencies for Java [13], this enforcement is not universally enabled or available for other languages, leaving them vulnerable to missing direct dependencies.

2.2.5 Redundant Dependencies. A redundant dependency is a declared dependency that is not used during the build process. Unlike missing dependencies, which jeopardize correctness, redundant dependencies silently degrade performance by introducing false constraints into the build graph [21]. The impact is twofold. First, they limit parallelism by enforcing artificial synchronization points. For example, declared dependencies $A \rightarrow B$ and $A \rightarrow C$ force B and C to be built before A, even when A does not actually depend on them. Removing these edges decouples A, allowing it to be built in parallel with (or even before) B and C. Second, they trigger spurious rebuilds. If either B or C is modified, A is invalidated and rebuilt, even though the change has no effect on A.

Although Bazel provides an *unused_deps* analyzer to detect redundant dependencies, it applies only to Java artifacts [13]. Moreover, by only removing unused dependencies, it does not guarantee optimal dependency graphs.

2.3 Dependency Error Detection

In prior work, detecting dependency errors typically involves a differential analysis: comparing the *actual dependency* graph against the *declared dependency* graph, as defined in the § 2.2. This section first reviews how prior techniques extract declared and actual dependencies, then describes limitations in various interpretations of actual dependencies, and finally discusses why these limitations make prior approaches difficult to apply directly to artifact-based build systems.

2.3.1 Extracting Declared Dependencies. Declared dependency graphs are typically easier to obtain than actual dependency graphs, as they are explicitly specified in build configurations. Tools for specific build systems often parse configuration files directly [17, 21, 29, 40]; MkCheck [28] infers declared dependencies by tracing IO-related system calls, while BuildFS [37] instruments build scripts. Build systems like Bazel and CMake further simplify this step through dedicated APIs, such as the Bazel Query API [10], which provide structured access to declared dependencies.

2.3.2 Extracting Actual Dependencies. Inferring the actual dependency graph is more difficult than extracting the declared dependency graph. To do so, prior work has used dynamic tracing, mutation-based, and ecosystem-specific strategies. We describe each strategy below.

Dynamic tracing approaches observe file-system behavior during a build. For example, BuildFS, VeriBuild, and BuildChecker trace IO-related system calls and interpret observed accesses as dependencies [21, 29, 37]. The mutation-based strategy, applied by approaches such as MkCheck [28], uses *build fuzzing*, i.e., they systematically touch files and observe whether the build output changes. Ecosystem-specific analyzers such as DepClean [39] and *unused_deps* [13] inspect bytecode or

compiled artifacts to identify dependencies that are used; however, such analyzers are limited to specific ecosystems such as the JVM.

2.3.3 Different Interpretations of Actual Dependencies. These strategies differ not only in implementation, but also in their notion of what constitutes an actual dependency. Dynamic tracing captures concrete file accesses, whereas build fuzzing captures output sensitivity to file mutations, and ecosystem-specific analyses of bytecode and artifact capture references encoded in artifacts. Although all aim to distinguish declared dependencies from actually needed ones [21, 29], the granularities at which they operate can lead to disagreements.

For our optimization problem, the relevant unit is a *declared target-level* edge in an artifact-based build graph. We treat such an edge as actual if removing it causes the project to stop building successfully. This operational notion should not be conflated with file-access-based notions: the two often agree, but neither strictly subsumes the other. For instance, a compiler might speculatively read a declared header file during execution (an actual dependency under file-access metrics), but the build may still succeed if that edge is removed (not an actual dependency under our operational metric). Conversely, a target might propagate essential build arguments such as compiler flags without passing any files to the downstream target. Since no files are opened, a system-call trace would classify the edge as unused, yet removing the edge would alter the behavior of the build, and could trigger a failure.

2.3.4 Applicability to Artifact-Based Build Systems. Prior approaches are not well aligned with artifact-level dependency semantics. *Build fuzzing* struggles with packaged artifacts such as JAR files and static libraries, since a build may update such artifacts, even when effective contents remain unchanged. *System-call tracing* cannot reliably distinguish actual dependencies from declared ones, since language toolchains often access all declared inputs regardless of actual, functional use. *Language- or ecosystem-specific analyzers* avoid some of these issues, but do not generalize to multi-language artifact-based build systems.

Moreover, prior approaches tend to focus on the detection of (potential) dependency errors rather than the optimization of build dependencies. Even when a redundant edge is detected, naïve removal can yield suboptimal builds (see § 3.1). DepClean [39] provides initial repair support, but is restricted to Java and Maven. In contrast, our approach adopts a graph-based iterative optimization strategy that circumvents these limitations.

2.4 Formalization of Dependency Graphs

Below, we establish the mathematical notation and formal definitions that characterize the dependency graphs in artifact-based build systems. This formalization serves as the basis for our subsequent problem definition and algorithm derivation.

Definition 2.1 (Dependency Graph). A *dependency graph* is represented as a Directed Acyclic Graph (DAG), denoted by $DG = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes representing build targets, and $\mathcal{E}$ is the set of directed edges representing dependencies between these targets.

Given a specific dependency graph DG, we use $\mathcal{N}_{DG}$ and $\mathcal{E}_{DG}$ to explicitly refer to the nodes and edges of DG. Each node $n_i \in \mathcal{N}$ corresponds to a build target, and each directed edge $e_{i \to j} \in \mathcal{E}$ indicates that target n_i directly depends on target n_j.

Definition 2.2 (Dependencies). For a given target $n_i \in \mathcal{N}$, its *dependencies* are defined as the set of targets that n_i directly depends on, denoted as $deps(n_i) = \{n_j \in \mathcal{N} \mid e_{i \to j} \in \mathcal{E}\}$.

Definition 2.3 (Dependents). For a given target $n_i \in \mathcal{N}$, its *dependents* are defined as the set of targets that directly depend on n_i, denoted as $\mathcal{D}(n_i) = \{n_j \in \mathcal{N} \mid e_{j \to i} \in \mathcal{E}\}$.

```
package <PKG>.sync.libs
...
import <PKG>.config.*
import <PKG>.coroutines.*
...
```

(a) A Source File of `sync/libs`

```
package <PKG>.go.sync
...
import <PKG>.sync.libs.*
import <PKG>.sync.proj.*
...
```

(b) A Source File of `go/sync`

```
kt_jvm_library(
  name = "flow",
  deps = [
    "<PKG>/config",
    "<PKG>/coroutines",
    "<PKG>/sync/proj",
  ])
```

(c) Build Script of `flow`

```
kt_jvm_library(
  name = "sync/libs",
  deps = [
    "<PKG>/flow",
  ])
kt_jvm_library(
  name = "go/sync",
  deps = [
    "<PKG>/sync/libs",
  ])
```

(d) Original Build Script

```
kt_jvm_library(
  name = "sync/libs",
  deps = [
    "<PKG>/config",
    "<PKG>/coroutines",
    "<PKG>/sync/proj",
  ])
kt_jvm_library(
  name = "go/sync",
  deps = [
    "<PKG>/sync/libs",
    "<PKG>/flow",
  ])
```

(e) Step 1

```
kt_jvm_library(
  name = "sync/libs",
  deps = [
    "<PKG>/config",
    "<PKG>/coroutines",
  ])
kt_jvm_library(
  name = "go/sync",
  deps = [
    "<PKG>/sync/libs",
    "<PKG>/config",
    "<PKG>/coroutines",
    "<PKG>/sync/proj",
  ])
```

(f) Step 2

```
kt_jvm_library(
  name = "sync/libs",
  deps = [
    "<PKG>/config",
    "<PKG>/coroutines",
  ])
kt_jvm_library(
  name = "go/sync",
  deps = [
    "<PKG>/sync/libs",
    "<PKG>/sync/proj",
  ])
```

(g) Step 3 (Result)

Fig. 2. Illustrative example adapted from JetBrains/hirschgarten (#303). Removed dependencies are enclosed in a Dotted Box. Lifted and flattened dependencies are highlighted in blue and orange, respectively.

Definition 2.4 (Transitive Dependencies and Dependents). For a given target $n_i \in \mathcal{N}$, its *transitive dependencies* $deps^+(n_i)$ and *transitive dependents* $\mathcal{D}^+(n_i)$ represent the transitive closures of its direct dependencies and dependents, respectively. Formally, these are defined as:

$$deps^+(n_i) = \bigcup_{n_j \in deps(n_i)} \left[n_j \cup deps^+(n_j) \right], \quad \mathcal{D}^+(n_i) = \bigcup_{n_j \in \mathcal{D}(n_i)} \left[n_j \cup \mathcal{D}^+(n_j) \right].$$

Definition 2.5 (Actual Direct Dependencies). For a given target $n_i \in \mathcal{N}$, its *actual direct dependencies* refer to the subset of its declared dependencies that are actually used during the build process, denoted as $deps_{actual}(n_i)$.

Given Definitions 2.1–2.5, we can now define the concepts of rebuild set and (cumulative) rebuild cost, which are central to our optimization goal.

Definition 2.6 (Rebuild Set). The *rebuild set* of a target n_i, denoted as $\mathcal{R}_i$, is the set of all targets that need to be rebuilt if n_i is modified, excluding n_i itself. This can be expressed as exactly the transitive dependents of n_i: $\mathcal{R}_i = \mathcal{D}^+(n_i) = \bigcup_{n_j \in \mathcal{D}(n_i)} \left[n_j \cup \mathcal{R}_j \right]$.

Definition 2.7 (Rebuild Cost). The *rebuild cost* of a target n_i is the size of its rebuild set $|\mathcal{R}_i|$. This metric quantifies the number of targets that must be rebuilt when n_i is modified.

Definition 2.8 (Cumulative Rebuild Cost). The cumulative rebuild cost of the dependency graph DG is the sum of the rebuild costs of all targets: $\sum_{n_i \in \mathcal{N}} |\mathcal{R}_i|$.

3 Approach

This section presents the design of DepReduce. We begin with an example to illustrate the intuition behind our approach. Next, we formalize the dependency optimization objective and derive our solution. We provide a rigorous analysis of the correctness and optimality of DepReduce.

3.1 Illustrative Example

We illustrate the operation of DepReduce using a real-world optimization scenario[2] in Figure 2. The objective is to optimize the build dependencies for the `sync/libs` and `go/sync` targets. For simplicity of presentation, this example focuses on the optimization operations rather than the exact attempt order used by the algorithm described in § 3.2, and the source code is simplified.

Setup. As shown in Figures 2c and 2d, `sync/libs` declares a single dependency on `flow`, and `flow` further depends on `config`, `coroutines`, and `sync/proj`. Consequently, `go/sync`, by relying on `sync/libs`, transitively inherits all dependencies of `sync/libs` and `flow`.

Problem. `sync/libs` imports `config` and `coroutines` (Figure 2a), but declares `flow` as its dependency; `go/sync` imports both `sync/libs` and `sync/proj` (Figure 2b), but only declares `sync/libs` as a dependency. The resulting dependency graph is inefficient: `sync/libs` retains redundant edges to `flow` and `sync/proj` (via `flow`), while `go/sync` relies on an implicit transitive path to access `sync/proj`.

The Naïve Approach. A naïve optimization strategy might attempt to simply remove `flow` from `sync/libs`. This would result in a build failure for two reasons: ① *missing direct dependency*, as `sync/libs` would lose access to `config` and `coroutines`; and ② *missing transitive access*, as `go/sync` would lose its path to `sync/proj`.

The DepReduce Approach. DepReduce optimizes dependencies by introducing *dependency lifting* and *dependency flattening* operations. The optimization comprises three steps:

Step 1: DepReduce first applies *dependency flattening* to `sync/libs` by promoting `flow`'s dependencies to direct dependencies. This enables the safe removal of `flow` from `sync/libs` without breaking compilation. Meanwhile, to preserve the transitive path for downstream consumers, DepReduce performs *dependency lifting*, explicitly adding `flow` as a dependency of `go/sync`. The updated build configuration is shown in Figure 2e, and the corresponding dependency graph transformations are illustrated in Figures 3c and 4c.

Step 2: Next, DepReduce re-examines the newly added dependencies. For `sync/libs`, `sync/proj` can be removed directly. For `go/sync`, DepReduce applies *dependency flattening* again by replacing `flow` with `config` and `coroutines`. The resulting state is presented in Figure 2f.

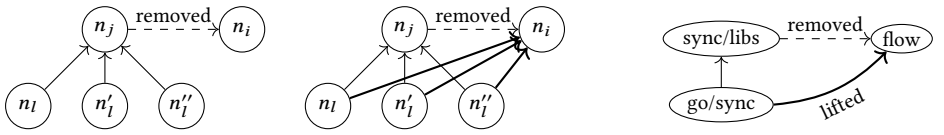
Step 3: Finally, DepReduce checks the dependencies added to `go/sync`. Both `config` and `coroutines` can be removed directly, resulting in the final optimized build configuration shown in Figure 2g.

Analysis. To quantify the impact of this optimization, we compare the rebuild sets for the underlying library targets. Initially, a change to any of the three targets triggers a rebuild of the entire chain. After optimization, the rebuild sets are reduced as follows:

$$\mathcal{R}_{\texttt{config}} \to \mathcal{R}'_{\texttt{config}} : \{\texttt{flow, sync/libs, go/sync}\} \to \{\texttt{flow, sync/libs}\}$$
$$\mathcal{R}_{\texttt{coroutines}} \to \mathcal{R}'_{\texttt{coroutines}} : \{\texttt{flow, sync/libs, go/sync}\} \to \{\texttt{flow, sync/libs}\}$$
$$\mathcal{R}_{\texttt{sync/proj}} \to \mathcal{R}'_{\texttt{sync/proj}} : \{\texttt{flow, sync/libs, go/sync}\} \to \{\texttt{flow, go/sync}\}$$

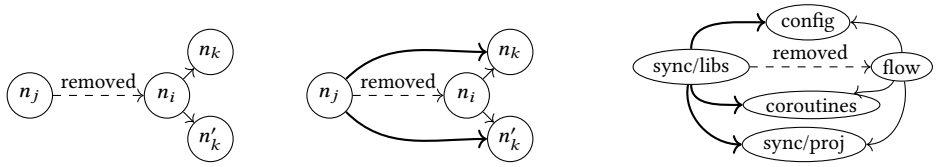
In practice, library targets often have more dependents, so eliminating redundant edges yields substantial reductions in the cumulative rebuild cost. For hirschgarten, from which this example is adapted, the cumulative rebuild cost was reduced from 45,098 to 43,771 (number of targets that need to be rebuilt).

[2]This example is adapted from JetBrains/hirschgarten (#303), where a redundant dependency was successfully removed by DepReduce with the help of dependency lifting and flattening.



(a) Before Lifting **(b)** After Lifting **(c)** Lifting in Illustrative Example

Fig. 3. Illustration of dependency lifting. From (a) to (b), n_i will be lifted as a dependency of all dependents of n_j. In (c), we visualize the lifting operation in the illustrative example (Figures 2d–2e). Added edges are shown in bold.



(a) Before Flattening **(b)** After Flattening **(c)** Flattening in Illustrative Example

Fig. 4. Illustration of dependency flattening. From (a) to (b), n_i's dependencies are flattened to n_j. In (c), we visualize the flattening operation in the illustrative example (Figures 2d–2e). Added edges are shown in bold.

3.2 Dependency Optimization Objective and Strategy

The primary objective of dependency optimization is to minimize the cumulative rebuild cost (see Definition 2.8) of the dependency graph DG: $\min_{DG} \sum_{n_i \in \mathcal{N}} |\mathcal{R}_i|$. As the terms in the summation are scalar values, minimizing each term individually ensures that the cumulative rebuild cost is also minimized.

To derive an optimal optimization strategy, we leverage the concept of topological ordering in DAGs. A topological ordering is a linear arrangement of the nodes in the graph such that, for every directed edge $e_{i \to j}$, node n_i appears before node n_j in the ordering. Given a dependency graph $DG = (\mathcal{N}, \mathcal{E})$, we first compute a topological ordering of its nodes. Let the nodes be indexed according to this ordering, such that:

$$n_j \in \mathcal{D}(n_i) \vee n_i \in deps(n_j) \implies j < i \tag{1}$$

To minimize the rebuild set $\mathcal{R}_i$ for each node n_i, we process the nodes in topological order. According to Definition 2.6 and Equation 1, the rebuild set $\mathcal{R}_i$ depends on the rebuild sets of its dependents, i.e. $\mathcal{R}_j$ for all $n_j \in \mathcal{D}(n_i)$. By processing in topological order, we ensure that the rebuild sets of all dependents have been minimized before computing $\mathcal{R}_i$, which makes $\mathcal{R}_j$ a constant when optimizing $\mathcal{R}_i$. The key to minimizing $\mathcal{R}_i$ is to reduce the number of dependents of n_i. To achieve this, we attempt to remove each dependent $n_j \in \mathcal{D}(n_i)$ from the build script in *reverse* topological order. This ensures that optimization can be performed in *a single pass*. The removal of each dependent n_j is carried out in the following four steps.

(1) **Direct Removal:** Attempt to remove the edge $e_{j \to i}$ from the build script. Rebuild and test the project to verify whether the removal is valid. If the project builds successfully, the removal is considered successful.
(2) **Dependency Lifting:** If direct removal fails, lift n_i as a dependency to all dependents of n_j. This operation, illustrated in Figure 3, ensures that the functionality provided by n_i remains accessible to the dependents of n_j. Then, rebuild the project again.
(3) **Dependency Flattening:** If dependency lifting also fails, flatten n_i for n_j. This is achieved by replacing n_i with all of its direct dependencies, as shown in Figure 4. Flattening allows n_j and

Algorithm 1: Dependency Optimization

```
1  Function DepReduce(DG):
2      N ← |N_DG|, E_original ← E_DG          // N: the number of nodes in DG, E_original: the original edges of DG
3      n_1, ..., n_N ← topological sort of DG
4      for i ← 1 to N do                                   // outer loop over nodes in topological order
5          pq_D ← {(j, n_j) | n_j ∈ D(n_i)}                // max-priority candidate queue
6          while pq_D is not empty do                      // inner loop over dependents in reverse topological order
7              (j, n_j) ← pop from pq_D
8              if not IsRemovable(DG, e_{j→i}, E_original) then continue
9              if not RemoveDep(DG, e_{j→i}) ∧ not LiftDeps(DG, e_{j→i}, pq_D)
10                 ∧ not FlattenDeps(DG, e_{j→i}) then restore modified build scripts      // abort removal
11 Function RemoveDep(DG, e_{j→i}):                        // direct removal
12     remove edge e_{j→i} from build scripts
13     if not BuildAndTestProject() then return false
14     E_DG ← E_DG \ e_{j→i}  return true
15 Function LiftDeps(DG, e_{j→i}, pq_D):                   // dependency lifting
16     foreach n_l ∈ D(n_j) s.t. IsAddable(DG, e_{l→i}) do
17         add edge e_{l→i} to build scripts
18     if not BuildAndTestProject() then return false
19     E_DG ← E_DG \ e_{j→i}
20     foreach e_{l→i} ∈ added edges do E_DG ← E_DG ∪ e_{l→i}, pq_D ← pq_D ∪ {(l, n_l)}
21     return true
22 Function FlattenDeps(DG, e_{j→i}):                      // dependency flattening
23     if IsAliasLikeTarget(n_i) then return false
24     foreach n_k ∈ deps(n_i) s.t. IsAddable(DG, e_{j→k}) do add edge e_{j→k} to build scripts
25     if not BuildAndTestProject() then return false
26     E_DG ← E_DG \ e_{j→i}
27     foreach e_{j→k} ∈ added edges do E_DG ← E_DG ∪ e_{j→k}
28     return true
```

its dependents to access their transitive dependencies without directly depending on n_i. The project is then rebuilt again to ensure correctness.

(4) **Abort Removal:** If all attempts fail, retain the edge $e_{j\to i}$ in the graph and restore the modified build scripts.

Algorithm 1 outlines the overall optimization strategy. The algorithm employs an outer loop (line 4), which traverses nodes in topological order, while an inner loop (line 6) iterates over the dependents of each node in *reverse* topological order. For each dependent edge, the algorithm attempts removal through a sequence of four steps (lines 9–10). Before performing dependency removal and addition, `IsRemovable` and `IsAddable` (lines 8, 17 and 24) are incorporated to avoid introducing unintended architectural changes.

3.3 Properties of the Optimization Strategy

We now analyze the key *monotonicity*, *invariance*, *completeness*, and *local-optimality* properties of DepReduce. Collectively, these properties establish the correctness and global optimality of the strategy.

3.3.1 Monotonicity. The monotonicity property guarantees that the optimization strategy of each operation either reduces or maintains the cardinality of the rebuild set.

Theorem 3.1 (Monotonicity). *Let $\mathcal{R}_i$ and $\mathcal{R}'_i$ denote the rebuild set of a target n_i before and after a removal operation involving a dependent n_j, respectively. Then, the updated rebuild set is a subset of the original: $\mathcal{R}'_i \subseteq \mathcal{R}_i$.*

Proof of Theorem 3.1. We verify this property by systematically examining each component operation of the strategy. We omit the explicit analysis for the dependencies of n_i, as the derivation for these rebuild sets is analogous to that of $\mathcal{R}_i$. For direct removal, when the edge $e_{j\to i}$ is removed, the rebuild set of n_i is updated as follows: $\mathcal{R}_i = \mathcal{R}'_i \cup \{n_j\} \cup \mathcal{R}_j \implies \mathcal{R}'_i \subseteq \mathcal{R}_i$.

After lifting n_i to all dependents of n_j as illustrated in Figure 3, the rebuild sets are updated as follows (only $\mathcal{R}_i$ is updated, while the rebuild sets of n_j and its dependents remain unchanged):

$$\mathcal{R}_i = \mathcal{R}'_i \cup \{n_j\},\ \mathcal{R}_j = \mathcal{R}'_j,\ \mathcal{R}_l = \mathcal{R}'_l,\ \forall n_l \in \mathcal{D}(n_j) \implies \mathcal{R}'_i \subseteq \mathcal{R}_i$$

After flattening n_i for n_j as shown in Figure 4, the rebuild sets are updated as follows (although new edges are added to n_k such that $n_k \in \mathit{deps}(n_i)$, the rebuild set $\mathcal{R}'_k$ remains unchanged because n_j and its dependents are already part of $\mathcal{R}_k$):

$$\mathcal{R}_i = \mathcal{R}'_i \cup \{n_j\} \cup \mathcal{R}_j,\ \mathcal{R}_j = \mathcal{R}'_j,\ \mathcal{R}_k = \mathcal{R}'_k,\ \forall n_k \in \mathit{deps}(n_i) \implies \mathcal{R}'_i \subseteq \mathcal{R}_i$$ □

3.3.2 Invariance. The invariance property ensures that the optimization strategy preserves the minimized state of previously processed nodes.

Theorem 3.2 (Invariance). *Suppose nodes are processed in topological order, i.e., Equation 1 holds. When processing a node n_i, the already-minimized rebuild set $\mathcal{R}'_k$ for any node n_k such that $k < i$ remains unchanged after the removal of a dependent $n_j \in \mathcal{D}(n_i)$.*

Proof of Theorem 3.2. In the outer loop of Algorithm 1 (line 4), nodes are processed in topological order. Therefore, when processing n_i, all nodes n_k with $k < i$ have already been processed, and their rebuild sets $\mathcal{R}'_k$ are finalized. Moreover, from the proof of Theorem 3.1, we know that all operations only modify the rebuild sets of n_i and its dependencies, i.e., $\mathcal{R}_k$ for all $n_k \in \mathit{deps}(n_i) \cup \{n_i\}$. Combining Equation 1, only $\mathcal{R}_k$ for $k \geq i$ can be affected. Hence, the rebuild sets $\mathcal{R}'_k$ for $k < i$ remain unchanged after processing n_i. □

3.3.3 Completeness. The completeness property ensures that *all* edges $e \in \mathcal{E}$ in the dependency graph are guaranteed to be examined in *a single pass*, including the newly added edges resulting from dependency lifting and flattening. Otherwise, some edges may be skipped and lead to suboptimal results, spelling multiple passes to achieve optimality, which is inefficient.

Lemma 3.3 (Completeness on Existing Edges). *All existing edges are guaranteed to be examined.*

Proof of Lemma 3.3. It directly follows from the structure of Algorithm 1. □

Lemma 3.4 (Completeness on New Edges). *When processing a node n_i, all newly added edges resulting from dependency lifting and flattening must be examined in subsequent steps.*

Proof of Lemma 3.4. As only dependency lifting and flattening introduce new edges, we analyze these two operations separately. *Dependency lifting* creates new edges from each $n_l \in \mathcal{D}(n_j)$, to n_i, as illustrated in Figure 3. By Equation 1, it holds that $l < j$. Since the inner loop in Algorithm 1 (line 6) processes dependents of n_i in reverse topological order, the newly added edges $e_{l\to i}$ will be processed after the current edge $e_{j\to i}$ in the same inner loop. *Dependency flattening* introduces new edges from n_j to each $n_k \in \mathit{deps}(n_i)$, as shown in Figure 4. By Equation 1, it follows that $i < k$. The outer loop processes nodes in topological order, ensuring that the newly added edges $e_{j\to k}$ will be processed later than the current edge $e_{j\to i}$. Thus, in both cases, newly added edges are guaranteed to be examined in subsequent steps. □

Corollary 3.5 (Completeness). *All edges $e \in \mathcal{E}$ are guaranteed to be examined, including the newly added edges resulting from dependency lifting and flattening.*

3.3.4 Local Optimality. The local-optimality property states that as long as an edge does not correspond to an actual direct dependency, it must be removed.

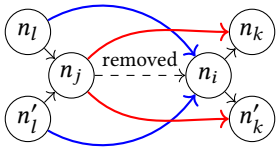


Fig. 5. Illustration of all operations while processing $e_{j\to i}$. Dashed edge is the removed edge. Bold blue edges are added due to dependency lifting. Bold red edges are added due to dependency flattening.

Without loss of generality, we assume both dependency lifting and flattening are performed when removing an edge $e_{j\to i}$, and we denote dependents of n_j as $n_l \in \mathcal{D}(n_j)$, and dependencies of n_i as $n_k \in deps(n_i)$, as shown in Figure 5.

Lemma 3.6 (Dependency Changes). *Let $deps'(n)$ denote the updated $deps(n)$ after optimization. When the edge $e_{j\to i}$ is removed after performing dependency lifting and flattening, the dependency changes can be expressed as follows:*

$$deps'(n_j) = \big(deps(n_j) \setminus \{n_i\}\big) \cup deps(n_i),\ deps^+(n_j) = deps^{+\prime}(n_j) \cup \{n_i\}$$
$$deps^+(n_k) = deps^{+\prime}(n_k),\ deps^+(n_l) = deps^{+\prime}(n_l),\ \forall n_k \in deps(n_i),\ n_l \in \mathcal{D}^+(n_j) \quad \textit{(unchanged)}$$

Proof. It directly follows from the definition of the operations and transitive dependencies. □

Then, we define the condition for a successful build, and analyze the condition for build failure after removing an edge $e_{j\to i}$.

Definition 3.7 (Build Success Condition). To ensure a successful build, all actual direct dependencies must be included in the transitive dependencies. Ignoring other factors, the build succeeds if and only if $deps_{actual}(n_i) \subseteq deps^+(n_i),\ \forall n_i \in \mathcal{N}$.

Lemma 3.8 (Build Failure Condition). *After removing the edge $e_{j\to i}$, the build fails if and only if $n_i \in deps_{actual}(n_j) \wedge n_i \notin deps^{+\prime}(n_j)$.*

Proof. By Definition 3.7, when build fails, the build success condition is violated. By Lemma 3.6, only $deps'(n_j)$ and $deps^{+\prime}(n_j)$ can be affected after removing the edge $e_{j\to i}$, and the only change is that n_i is removed from them. Thus, the build success condition is violated if and only if $deps_{actual}(n_j) \not\subseteq deps^{+\prime}(n_j)$, which is equivalent to $n_i \in deps_{actual}(n_j)$ and $n_i \notin deps^{+\prime}(n_j)$. □

Theorem 3.9 (Local Optimality). *An edge $e_{j\to i}$ will be removed if and only if $n_i \notin deps_{actual}(n_j) \vee n_i \in deps^{+\prime}(n_j)$, where $deps^{+\prime}(n_j)$ and $deps'(n_j)$ are the transitive dependencies and direct dependencies of n_j after the removal of $e_{j\to i}$.*

Proof of Theorem 3.9. By Corollary 3.5, the edge $e_{j\to i}$ will be examined during the optimization process, and by Algorithm 1, the edge $e_{j\to i}$ will be removed if and only if build does not fail after the removal and any necessary dependency lifting and flattening. Finally, by Lemma 3.8, the build does not fail if and only if $n_i \notin deps_{actual}(n_j) \vee n_i \in deps^{+\prime}(n_j)$. □

Algorithm 2: Dependency Edge Predicates

```
1 Function IsRemovable(DG, e_{j→i}, E_original):
2     if e_{j→i} ∉ E_original then return true                    // new edge is always removable
3     if IsBlocked(remove, e_{j→i}) then return false
4     if ∃n_i ∈ deps⁺(n_k), ∀n_k ∈ deps(n_j) then return false    // to avoid missing direct dependencies
5     return true
6 Function IsAddable(DG, e_{j→i}):
7     if e_{j→i} ∈ E_DG then return false                         // already exists
8     if IsBlocked(add, e_{j→i}) then return false
9     return true
```

3.3.5 Global Optimality. Combining all properties established in this subsection, we conclude that the proposed optimization strategy is globally optimal.

Theorem 3.10 (Global Optimality). *The proposed optimization strategy produces a globally optimal solution that minimizes the cumulative rebuild cost across all targets in the dependency graph.*

Proof of Theorem 3.10. We prove by induction that rebuild sets for all nodes are minimized after processing the entire graph. Then, the cumulative rebuild cost across all targets is minimized, establishing the global optimality of the strategy.

Base Case. When processing the first node n_1 in topological order, it has no dependents, and thus its rebuild set $\mathcal{R}_1$ is already minimized.

Inductive Step. Assume that for all nodes n_k with $k < i$, their rebuild sets $\mathcal{R}'_k$ are minimized after processing. When processing node n_i, by Theorem 3.2, the already-minimized rebuild sets $\mathcal{R}'_k$ for $k < i$ remain unchanged. By Theorem 3.9, each edge $e_{j\to i}$ is removed if it is not an actual direct dependency or if n_i remains transitively accessible from n_j after removal. Therefore, after processing n_i, its rebuild set $\mathcal{R}'_i$ is also minimized. By induction, all nodes in the graph have their rebuild sets minimized. □

4 Implementation

We have implemented DepReduce as BazelDepReduce, an automated dependency optimization tool for Bazel written in Rust. BazelDepReduce is designed to be extensible to other build systems. In this section, we describe the implementation details of BazelDepReduce, focusing on strategies to prevent unintended architectural changes and the integration with different build systems.

4.1 Preventing Unintended Changes

Aggressively pruning dependencies can introduce unintended architectural changes. We propose the following mechanisms to prevent these issues.

Preventing Missing Direct Dependencies. A missing direct dependency can arise as an unintended consequence of the optimization strategy, because such removal does not necessarily lead to build failures. In fact, Theorem 3.9 states that an edge $e_{j\to i}$ will be removed if $n_i \in deps^{+\prime}(n_j)$, which formally confirms this. However, as discussed in § 2.2.4, this situation introduces a risk of future build failures. Therefore, such unintended changes may not be acceptable to project maintainers. To mitigate this issue, we retain the edge $e_{j\to i}$ if it is not a newly-added edge and n_i is reachable from n_j via transitive dependencies after the removal of the edge $e_{j\to i}$ (see line 4 in Algorithm 2):

$$n_i \in deps^{+\prime}(n_j) \land e_{j\to i} \in \mathcal{E}_{original} \implies e_{j\to i} \text{ cannot be removed.} \tag{2}$$

Preserving such edges does not impact the rebuild set of n_i, as n_j retains transitive access to n_i. Thus, retaining these edges avoids the risk of introducing missing direct dependencies without

compromising the optimization objective. Additionally, because dependents are processed in reverse topological order, when evaluating the edge $e_{j \to i}$ in the inner loop, all edges $e_{k \to i}$ for $n_k \in deps^+(n_j)$ and $k > j$ have already been processed. Consequently, whether $n_i \in deps^{+\prime}(n_j)$ holds can be determined deterministically when enforcing the constraint in Equation 2, since subsequent iterations cannot introduce new paths from n_j to n_i via nodes n_k with $k \leq j$. Hence, the optimization strategy can still be done in a single pass.

Ignoring Alias-Like Targets. In Bazel, developers often introduce alias-like targets to simplify dependency management. These targets are typically defined using the `alias` rule [5]. Similarly, rules like `java_library` may omit the `srcs` and instead use the `exports` to expose dependencies transitively [9]. Alias-like targets are commonly used to abstract away complex target names or group related dependencies. Flattening such alias-like targets is generally undesirable, as they are deliberately introduced to preserve architectural clarity and maintainability. To address this, we identify and exclude targets without source files to avoid unnecessary flattening.

Node Filtering Mechanism. While successful builds and passing tests are necessary for validating dependency optimizations, they do not always guarantee build soundness, particularly in projects involving scripting languages such as Python. With limited test coverage, missing dependencies can lead to runtime errors. To address this, we introduce a configurable node filtering mechanism that enables fine-grained control over the dependency optimization process.

4.2 Build System Integration

We abstract the integration with build systems as a set of common operations, including extracting declared dependencies, editing build configurations, and building and testing the project. This abstraction facilitates extending BazelDepReduce to support other build systems.

Extracting Declared Dependencies. Query APIs [10, 15] are provided by Bazel and Buck, which allow retrieval of the declared dependency graph in a structured format, like XML or JSON. We then parse it into a *DG* data structure for further processing. For Cargo projects, we use Cargo's metadata library to extract declared dependencies from `Cargo.toml`.

Editing Build Configurations. Bazel and Buck build configurations are written in Starlark, a Python-like language. To programmatically edit dependencies, we parse these scripts using a Python parser library. We then utilize metadata returned from the Query API, which provides the build configuration location for each target. Dependency additions or removals are performed directly on the dependency lists of the relevant targets. To preserve line number consistency, we insert dependencies as single lines and avoid reformatting the scripts after modification. For Cargo projects, we use a TOML editing library to edit the `Cargo.toml` files directly.

5 Evaluation

In this section, we evaluate the effectiveness and efficiency of DepReduce by applying it to selected real-world Bazel projects written in seven programming languages supported by Bazel.

The experiments are performed on a machine running Ubuntu 22.04 LTS. The hardware comprises an Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz with 112 cores, and 3.89TB of RAM.

5.1 RQ1: Effectiveness

To evaluate the effectiveness of DepReduce, we applied BazelDepReduce to 19 selected open-source Bazel projects that are non-trivial and diverse in terms of programming languages. These projects are listed in Table 1. To evaluate how developers react to the DepReduce-suggested improvements, we submitted Pull Requests (PRs) containing the proposed dependency optimizations to these projects.

Table 1. Dependency optimization results on 19 open-source Bazel projects. Projects marked with undesired indicate that the optimization results introduced many unintended changes. We report n/a for **#Commits** in projects without merged PRs or subsequent commits after PRs being merged, as it is not applicable in those cases. Further details are provided in § 5.1.2.

Project	Stars	Language	PR ID	Merged	#DR/TR/L/F	Δ Rebuild Cost	#Commits
rocketmq	22.1k	Java	#9610	✓	12/ 12/ 0/ 0	−2.80 % 2710 → 2634	83/103 = 80.58%
grpc-java	11.8k	Java	#12310	✓	3/ 3/ 0/ 0	−0.06 % 12,893 → 12,885	6/124 = 4.84%
copybara	2.4k	Java	#329	✓	1/ 1/ 0/ 0	−0.02 % 25,294 → 25,290	1/43 = 2.33%
buildfarm	733	Java	#2332	✓	35/ 35/ 0/ 0	−6.08 % 12,523 → 11,762	30/114 = 26.32%
gerrit	n/a	Java	#500801	✓	10/ 25/17/ 0	−4.60 % 23,650 → 22,563	288/500 = 57.60%
closure-templates	663	Java	#2550		23/ 24/ 0/ 1	−4.00 % 13,078 → 12,555	n/a
jazzer	1.1k	Java/Kotlin	#949		11/ 11/ 0/ 0	−0.36 % 14,703 → 14,650	n/a
perses	208	Kotlin	#42	✓	30/ 41/16/ 5	−2.88 % 65,340 → 63,467	n/a
hirschgarten	119	Kotlin	#303	✓	33/ 48/15/10	−2.94 % 45,098 → 43,771	90/167 = 53.89%
angular	98.6k	TypeScript	#63348	✓	50/ 51/ 1/ 0	−8.17 % 24,261 → 22,270	136/500 = 27.20%
components	25k	TypeScript	#33004		96/103/ 2/ 6	−8.96 % 53,071 → 48,319	n/a
dev-infra	73	TypeScript	#3623	✓	7/ 7/ 0/ 0	−1.07 % 5706 → 5645	n/a
typedb	4k	Rust	#7558	✓	7/ 7/ 0/ 0	−1.44 % 8121 → 8004	6/49 = 12.24%
trpc-cpp	337	C++	#220	✓	19/ 33/ 9/ 7	−0.24 % 128,144 → 127,842	0/1 = 0.00%
zirgen	124	C++	#325		14/ 24/10/ 5	−3.69 % 33,159 → 31,934	n/a
buildbuddy	739	ProtoBuf/Go	#11955	✓	5/ 5/ 0/ 0	−0.01 % 719,333 → 719,296	n/a
zml	2.5k	Zig	n/a		0/ 0/ 0/ 0		
cockroach	31.2k	Go	n/a		undesired		
icu	3.2k	C++	n/a		undesired		
Total (#Removal > 0)		6	16	12	356/430/70/34		

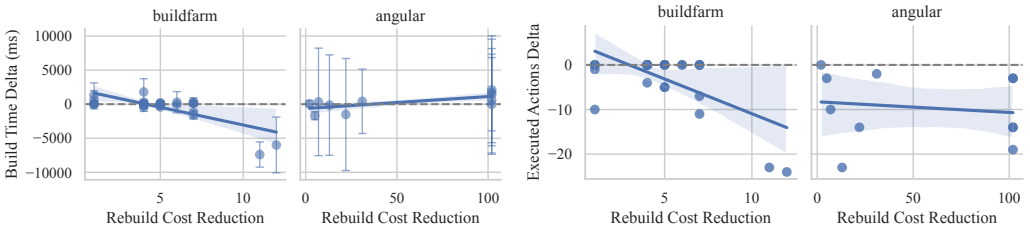
#DR/TR/L/F denotes **#Direct Removal**, **#Total Removal**, **#Lifting**, and **#Flattening** (§ 5.1.2).

5.1.1 Project Selection. We retrieved 126,082 repositories with > 62 stars via the GitHub Search API[3] and selected a population of 560 repositories with Bazel files (e.g., `BUILD`, `.bazelrc`) in their root directory [6] . From that population, we sampled projects with a substantial cumulative rebuild cost (see Definition 2.8) to maximize impact. For practical reasons, we excluded repositories whose build environment setup was too complex. Optimization challenges in C++ imposed further constraints (see Case Study 3 in § 5.1.4), resulting in a final set of 19 projects. This sampling strategy prioritizes impact and feasibility, and does not aim to be representative of all Bazel projects.

5.1.2 Procedure. For each project, we ran BazelDepReduce, iteratively refining filtering configurations (§ 4.1) if unintended changes occurred. We manually reviewed and revised proposals[4] before submission. Two projects yielded undesired optimizations, including cockroach, whose Bazel files are automatically generated by bazel-gazelle and thus do not need dependency optimization, and icu, where the proposed optimizations introduced unintended changes (see Case Study 3).

To evaluate the practical impact, we analyzed up to 500 subsequent commits for each merged PR, with a median of 114 commits. We estimated the impact by counting commits that modified build targets with reduced rebuild costs following our optimization. To measure build time differences, we replayed commits for which our optimization could be reverted in a containerized environment. To ensure that a sample of substance is available for each studied project, we focused on projects with at least six such commits that still built successfully after replaying. The buildfarm (30 commits) and angular (11 commits) projects survived this filtering process. Accordingly, build-time and executed-action analyses provide supporting evidence on a small subset, whereas rebuild-cost reduction remains the primary metric.

[3]We maximized dataset coverage by iteratively lowering the star-count bound to bypass API pagination limits, stopping only when a single query returned repositories with identical star counts (62).
[4]Minor adjustments included formatting, fixing CI pipelines, or selecting precise dependency labels.



(a) Correlation with Build Time Improvement **(b)** Correlation with Reduced Executed Actions

Fig. 6. Linear regression plots showing correlations between rebuild cost reduction and **(left)** build time improvement, and **(right)** the reduced number of executed actions. Each point represents a commit run five times. Error bars show 95% confidence intervals of the mean build time improvement. Regression is performed on commit-level means, and shaded regions show 95% confidence intervals of the regression lines.

We employed the following metrics to evaluate the effectiveness of DepReduce:

- **Merged:** Whether the PR containing the optimizations was accepted by the community and merged into the project.
- **#Direct Removal (DR):** The number of dependencies removed via *Direct Removal*.
- **#Total Removal (TR):** The number of dependencies removed, including those enabled by dependency lifting and flattening.
- **#Lifting (L):** The number of dependencies added through dependency lifting.
- **#Flattening (F):** The number of dependencies added through dependency flattening.
- **Δ Rebuild Cost:** The reduction in the cumulative rebuild cost (see Definition 2.8).
- **#Commits:** The number of commits whose rebuild cost was impacted after merging our PR.
- **Build Time Improvement:** The change in wall-clock build time (ms) for selected commits between the original and optimized codebases.
- **Reduced Executed Actions:** The change in the number of build actions executed for selected commits during the build process between the original and optimized codebases.

5.1.3 Results. Table 1 summarizes the results on 19 open-source Bazel projects. Our method reduces rebuild cost in 16 projects, yielding 430 removed dependencies. More importantly, eight projects benefited from dependency lifting or flattening, which together accounted for 74 of the total removed dependencies. These results show that BazelDepReduce is practically useful on this selected set of Bazel projects. Notably, most projects that accepted our PRs are actively maintained and highly popular, with three out of twelve having over 10k stars on GitHub. Comments from maintainers of these projects indicate that they value the optimization and are interested in our work, further confirming the practical value of DepReduce for developers:

> *Thanks for this PR! Are there any tools you used that we can incorporate to our CI/CD system?* (Link: #2332)
> *Awesome! Please let us know when it's public.* (Link: #500801)
> *Thank you. Always nice to lines removed!* (Link: #11955)

Across the twelve projects, our optimizations influenced 640 of the 1,601 subsequent commits. As illustrated in Figure 6, buildfarm demonstrates a strong positive correlation between the predicted *rebuild cost reduction* and actual *build time improvement*. Conversely, angular exhibits a weak or even negative correlation. We hypothesize that this behavior stems from system-level variance (e.g., I/O fluctuations). The wide 95% confidence intervals indicate substantial runtime noise that may obscure or outweigh the benefits gained from reduced rebuild costs. Despite this noise, the optimization remains effective: the number of *executed actions* decreased in ten of the eleven

```
#include "gtest/gtest.h"
...
int main(int argc, char** argv)
{ ... }
```

(a) C++ Source File of a Test Suite

```
cc_test(
    deps = [ ...
        "@googletest//:gtest_main",
        #           ":gtest" is lifted here
    ] ...
)
```

(b) Bazel Script of the Test Suite

Fig. 7. Duplicated main function in trpc-group/trpc-cpp #220 (Case Study 1).

```
java_library(
    name = "exception",
    srcs = ["exception.java"],
)

java_binary(
    name = "ExampleFuzzTests",
    srcs = glob(["*.java"]),
    deps = [
        ":exception",
        # redundancy detected
        ...
    ],
)
```

(a) Bazel Script Before Fixing

```
java_library(
    name = "exception",
    srcs = ["exception.java"],
)

java_binary(
    name = "ExampleFuzzTests",
    srcs = glob(
        ["*.java"],
        exclude = [
            "exception.java",
        ],
    ),
)
```

(b) Bazel Script After Fixing

Fig. 9. Redundant source files in CodeIntelligenceTesting/jazzer #949 (Case Study 2).

analyzed angular commits. This confirms that the optimization successfully pruned the build graph, even when wall-clock improvements were statistically obscured. We therefore treat build-time measurements as supporting evidence rather than as the primary basis for our effectiveness claims. We also measured clean build time as an end-to-end build-system check. Among the projects with merged PRs, Figure 8 reports the four with the most total removals whose evaluated revisions still build with default containerized build arguments; the small changes (−2.3% to −0.4%) indicate no significant overhead and suggest a slight improvement in clean-build time.

5.1.4 Case Studies. We further analyzed the results and highlight three representative cases that illustrate both the strengths and limitations of our approach.

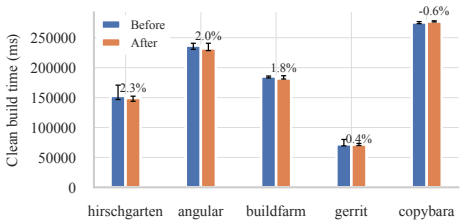


Fig. 8. Clean build time comparison.

Case Study 1: Duplicated Main Function in trpc-cpp #220. As shown in Figure 7, the source file of a test suite contains a `main` function, yet its target `cc_test` still declares a dependency on `gtest_main` from the GoogleTest library, which also provides a `main` function. This dependency is not only redundant but may also introduce undefined behavior. BazelDepReduce successfully identified and removed it by lifting the transitive dependency `gtest`.

Case Study 2: Redundant Source Files in jazzer #949. As depicted in Figure 9a, the dependency `exception` was removed because its source file `exception.java` was already included by the target `ExampleFuzzTests`. Upon manual review, we reverted this removal and instead resolved the redundancy by excluding the duplicated source file from the target, as shown in Figure 9b.

```
cc_library(
    name = "utrie2",
    srcs = ["utrie2.cpp"],
    deps = [":platform"],
)
```

(a) Original Bazel Script

```
cc_library(
    name = "utrie2",
    srcs = ["utrie2.cpp"],
    deps = [":headers"],
)
```

(b) Optimized Bazel Script

Fig. 10. Unintended architectural changes in C++ static library of unicode-org/icu (Case Study 3).

Although this removal was not counted as successful in Table 1, it shows that our approach can uncover subtle redundancies that may escape developers' attention.

Case Study 3: Unintended Architectural Changes in C++ Static Libraries. The scenario illustrated in Figure 10 was encountered in icu (unofficial PR). In C++, a static library is an archive of object files, and its compilation does not require linking the dependencies, i.e., unresolved symbols are permitted. Consequently, the actual dependencies for compiling a static library are limited to header files of its depending libraries. Nevertheless, developers tend to declare the whole library as a dependency to encapsulate the build target, allowing downstream developers to avoid managing transitive dependencies explicitly. In this case, BazelDepReduce removed the dependency `platform` from `utrie2` and added `headers`, reflecting the true compilation requirements of `utrie2.cpp`. While this change does not affect the build of `utrie2` itself, it undermines the original architectural intent, complicating downstream development. For example, when a test target exists downstream, non-header dependencies removed from the library are lifted to the test target. Although BazelDepReduce reduces rebuild cost substantially in such cases, it introduces unintended changes, making the change undesired. Nevertheless, developers who prioritize incremental build performance may still choose to apply such changes.

> **RQ1**: BazelDepReduce reduces rebuild costs on 16 selected real-world Bazel projects, eliminating 430 dependencies. The results cover projects written in several languages supported by Bazel, while also revealing limitations in handling C++ projects. Positive feedback from project maintainers highlights its practical value for developers.

5.2 RQ2: Efficiency

We measured the efficiency of DepReduce by analyzing BazelDepReduce's logging output during dependency optimization on each project. We employ the following metrics:

- **Time:** The total time taken by BazelDepReduce to complete the optimization process.
- **#Target:** The number of target nodes parsed from the dependency graph.
- **#Edge:** The number of edges considered for removal after filtering (see § 4.1 for the node filtering mechanism), and the number of edges in the dependency graph.
- **#Build:** The number of builds (including tests) executed to validate the optimization.

Results. Table 2 presents the efficiency results across 17 open-source Bazel projects (excluding two with undesired results). Even for projects with large dependency graphs—in terms of both targets and edges—the overall optimization time remains practical. The longest observed optimization duration was eight hours and four minutes for perses[5] (excluding gerrit, which encountered Bazel hanging issues). In most cases, the total optimization time is dominated by the repeated incremental builds required to attempt edge removals. This duration is primarily influenced by the project's size, complexity, the number and runtime of test cases, and also programming language.

[5]The long optimization time for perses stems from its extensive suite of long-running system tests.

Table 2. Dependency optimization efficiency results on 17 open-source projects. Time marked with * indicates that Bazel occasionally hangs during builds. We set a ten-minute-per-build timeout to mitigate this, but the reported time may not accurately reflect actual performance for this project.

Project	Language	Time (h:m)	#Target	#Edge	#Build
grpc-java	Java	0:06	338	53/1760	107
jazzer	Java/Kotlin	0:11	503	144/2974	261
buildfarm	Java	0:27	290	176/1970	352
buildbuddy	ProtoBuf/Go	0:26	3756	134/22,783	301
copybara	Java	0:35	342	153/6472	352
dev-infra	TypeScript	0:22	262	142/1345	327
typedb	Rust	0:52	585	112/2158	210
zirgen	C++	1:30	527	230/2686	442
hirschgarten	Kotlin	1:48	597	570/6359	1074
rocketmq	Java	2:19	145	70/ 802	89
angular	TypeScript	2:27	716	634/4739	1391
closure-templates	Java	2:32*	192	369/2152	803
components	TypeScript	3:53	767	1406/9046	2949
trpc-cpp	C++	6:42	914	658/5813	1565
perses	Kotlin	8:04	475	1052/3537	2130
gerrit	Java	49:37*	1582	947/6683	1235

In **#Edge**, X/Y denotes considered and total edges after filtering and in the dependency graph, respectively.

Our filtering mechanisms substantially reduce the number of edges subject to removal, with most filtered edges corresponding to alias-like targets, such as third-party libraries.

RQ2: BazelDepReduce demonstrates practical optimization times on the evaluated Bazel projects. Filtering mechanisms substantially reduce the number of candidate edges, improving efficiency. As dependency optimization is a periodic maintenance task, optimization times of several hours are acceptable in practice.

6 Related Work

We categorize related work into three areas: dependency error detection, mitigation of redundancy impact, and error localization in build systems.

Dependency Error Detection. Dependency errors in build systems have been studied using static, dynamic, and hybrid techniques. Static approaches, such as MAKAO [17] and SYMake [40], extract dependency graphs from build scripts to identify anomalies, while BuildPredictor [45] combines build-script and source-code analysis to predict missing dependencies. Dynamic approaches, including MkCheck [28], and BuildFS [37], analyze build executions or system calls to compare observed accesses with declared dependencies. Hybrid methods, such as VeriBuild [21] and BuildChecker [29], combine static and dynamic information to detect both missing and redundant dependencies. In ecosystem-specific settings, DepClean [39] uses bytecode analysis to prune unused Java dependencies, while Bazel provides tools such as *unused_deps* and *strict-deps* [13]. However, most prior approaches focus on detection rather than automatic repair, and many are tied to particular languages or build systems. In contrast, our approach targets automated optimization in artifact-based build systems and is implemented in a way that can be extended to similar systems.

Mitigation of Redundancy Impact. Recent work mitigates the impact of redundancies without removing them. Dep-Scimitar [41] improves CI efficiency by skipping commits with only updated

unused dependencies. However, it relies on external tools to detect unused dependencies and does not repair the build configuration, leaving the underlying redundancies intact.

Error Localization in Build Systems. Research on build maintenance extends to error localization and repair. MkFault [19] identifies faults in Makefiles via execution traces, while HireBuild [23] and BuildMedic [30] automate repairs for Gradle and Maven using historical patterns and predefined strategies. However, these approaches primarily target build failures. As redundant dependencies rarely cause immediate failures, these tools are inapplicable to dependency optimization.

7 Discussion

We discuss the validity of our evaluation in terms of construct, internal, and external validity.

Construct Validity. We use cumulative rebuild cost as our primary effectiveness metric. Although it may not always reflect the full end-to-end performance impact, it directly captures unnecessary rebuild propagation in the dependency graph. Like prior work [21, 28, 29, 37], we also do not rely on wall-clock build time as the main metric, since it is heavily affected by caching, I/O, and system-level noise. We still report build-time evidence where feasible, but treat it as supporting rather than primary evidence.

Internal Validity. Candidate optimizations are validated mainly through build-and-test outcomes, which may miss issues under limited test coverage particularly for projects written in scripting languages. To reduce this risk, we adopt a node filtering mechanism and manually review proposed patches before PR submission. Integrating language-specific static analysis could further improve the accuracy and efficiency of the optimization process, and is an area for future work.

External Validity. Our evaluation is primarily a Bazel study on 19 selected projects, so the results should be interpreted mainly as evidence in the Bazel setting. We nevertheless implemented adapters for Buck and Cargo, providing preliminary evidence that the implementation can be ported beyond Bazel. Broader evaluation on other build systems is future work.

8 Conclusion

This paper presents DepReduce, an automated dependency optimization approach for artifact-based build systems. We formalized rebuild-cost minimization and proved the correctness and optimality of a strategy based on dependency lifting and flattening.

DepReduce is implemented as BazelDepReduce for Bazel. Evaluated on 19 selected projects, it reduces rebuild costs in 16 projects, with twelve PRs merged and positive maintainer feedback. We also adapted BazelDepReduce to Buck and Cargo, providing preliminary evidence of extensibility. Future work includes support for more build systems, CI integration with incremental optimization support, and user studies.

Acknowledgments

We thank all the anonymous reviewers for ISSTA 2026 for their insightful feedback and comments. This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, and by CFI-JELF Project #40736.

Data Availability

We released DepReduce as open-source software at https://github.com/xuhongxu96/depreduce [43] and provided a dataset [6] containing metadata about the Bazel repositories used in the evaluation.

References

[1] [n. d.]. Architectural Model | Buck2 — buck2.build. https://buck2.build/docs/developers/architecture/buck2/. [Accessed 20-11-2025].

[2] [n. d.]. Bazel — bazel.build. https://bazel.build. [Accessed 30-08-2025].
[3] [n. d.]. Buck2 build system website | Buck2 — buck2.build. https://buck2.build/. [Accessed 19-11-2025].
[4] [n. d.]. Dependency Management | Bazel — bazel.build. https://bazel.build/basics/dependencies. [Accessed 04-12-2025].
[5] [n. d.]. General Rules | Bazel — bazel.build. https://bazel.build/versions/8.5.0/reference/be/general. [Accessed 29-01-2026].
[6] [n. d.]. GitHub - xuhongxu96/bazel-repos: Bazel-Based Projects in GitHub — github.com. https://github.com/xuhongxu96/bazel-repos/. [Accessed 30-08-2025].
[7] [n. d.]. Home | Angular — angular.dev. https://angular.dev/. [Accessed 22-11-2025].
[8] [n. d.]. Introduction - The Cargo Book — doc.rust-lang.org. https://doc.rust-lang.org/cargo/. [Accessed 20-11-2025].
[9] [n. d.]. Java Rules | Bazel — bazel.build. https://bazel.build/reference/be/java#java_library.exports. [Accessed 29-01-2026].
[10] [n. d.]. Query quickstart | Bazel — bazel.build. https://bazel.build/query/quickstart. [Accessed 30-08-2025].
[11] [n. d.]. RocketMQ | RocketMQ — rocketmq.apache.org. https://rocketmq.apache.org/. [Accessed 22-11-2025].
[12] [n. d.]. Sandboxing | Bazel — bazel.build. https://bazel.build/docs/sandboxing. [Accessed 30-08-2025].
[13] [n. d.]. Strict Java Deps and ‘unused_deps‘ — blog.bazel.build. https://blog.bazel.build/2017/06/28/sjd-unused_deps.html. [Accessed 30-08-2025].
[14] [n. d.]. The Bazel codebase — bazel.build. https://bazel.build/contribute/codebase. [Accessed 20-11-2025].
[15] [n. d.]. uquery | Buck2 — buck2.build. https://buck2.build/docs/users/commands/uquery/. [Accessed 27-11-2025].
[16] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2008. The Evolution of the Linux Build System. *Electronic Communications of the EASST* 8 (Feb. 2008). doi:10.14279/tuj.eceasst.8.115
[17] 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, Paris, France, 114–123. doi:10.1109/ICSM.2007.4362624
[18] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 14–23. doi:10.1109/SCAM.2012.28
[19] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault Localization for Make-Based Build Crashes. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 526–530. doi:10.1109/ICSME.2014.87
[20] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (Feb. 2019), 381–416. doi:10.1007/s10664-017-9589-y
[21] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 463–474. doi:10.1145/3395363.3397388
[22] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265. doi:10.1002/spe.4380090402
[23] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1078–1089. doi:10.1145/3180155.3180181
[24] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable Incremental Building with Dynamic Task Dependencies. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 76–86. doi:10.1145/3238147.3238196
[25] Olivier Lamy. [n. d.]. Welcome to Apache Maven - Maven — maven.apache.org. https://maven.apache.org. [Accessed 30-08-2025].
[26] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. 1997. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*. 20–32. doi:10.1109/METRIC.1997.637156
[27] M. M. Lehman. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1 (1979), 213–221. doi:10.1016/0164-1212(79)90022-0
[28] Nandor Licker and Andrew Rice. 2019. Detecting Incorrect Build Rules. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1234–1244. doi:10.1109/ICSE.2019.00125
[29] Jun Lyu, Shanshan Li, Bohan Liu, He Zhang, Guoping Rong, Chenxing Zhong, and Xiaodong Liu. 2025. Detecting Build Dependency Errors by Dynamic Analysis of Build Execution against Declaration. *IEEE Transactions on Software Engineering* (2025), 1–18. doi:10.1109/TSE.2025.3566225
[30] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 106–117. doi:10.1109/SANER.2018.8330201

[31] Shane McIntosh. 2011. Build system maintenance. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1167–1169. doi:10.1145/1985793.1986031

[32] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. 2012. The evolution of Java build systems. *Empirical Software Engineering* 17, 4 (Aug. 2012), 578–608. doi:10.1007/s10664-011-9169-5

[33] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering* 20, 6 (Dec. 2015), 1587–1633. doi:10.1007/s10664-014-9324-x

[34] Shane McIntosh, Martin Poehlmann, Elmar Juergens, Audris Mockus, Bram Adams, Ahmed E. Hassan, Brigitte Haupt, and Christian Wagner. 2014. Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 145–154. doi:10.1145/2591062.2591181

[35] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (June 2016), 78–87. doi:10.1145/2854146

[36] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 724–734. doi:10.1145/2568225.2568255

[37] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A model for detecting faults in build specifications. *Replication Package for Article: A Model for Detecting Faults in Build Specifications* 4, OOPSLA (Nov. 2020), 144:1–144:30. doi:10.1145/3428212

[38] César Soto Valero. 2023. *Debloating Java Dependencies*. PhD Thesis. KTH, Software and Computer systems, SCS. Backup Publisher: KTH, Software and Computer systems, SCS.

[39] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering* 26, 3 (May 2021), 45. doi:10.1007/s10664-020-09914-8

[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. doi:10.1109/ICSE.2012.6227152

[41] Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the npm Ecosystem. *Proc. ACM Softw. Eng.* 1, FSE, Article 116 (July 2024), 24 pages. doi:10.1145/3660823

[42] Rongxin Wu, Minglei Chen, Chengpeng Wang, Gang Fan, Jiguang Qiu, and Charles Zhang. 2022. Accelerating Build Dependency Error Detection via Virtual Build. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–12. doi:10.1145/3551349.3556930

[43] Hongxu Xu. 2026. *DepReduce: Automated Dependency Optimization for Artifact-Based Build Systems*. doi:10.5281/zenodo.20670274

[44] Shenyu Zheng, Bram Adams, and Ahmed E. Hassan. 2024. Does using Bazel help speed up continuous integration builds? *Empirical Software Engineering* 29, 5 (July 2024), 110. doi:10.1007/s10664-024-10497-x

[45] Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. 2014. Build Predictor: More Accurate Missed Dependency Prediction in Build Configuration Files. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. 53–58. doi:10.1109/COMPSAC.2014.12

accepted at ISSTA 2026 (DOI: 10.1145/3832191)