The Classics Never Go Out of Style

An Empirical Study of Downgrades from the Bazel Build Technology

Mahmoud Alfadel
Software REBELs
University of Waterloo, Canada
malfadel@uwaterloo.ca

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

ABSTRACT

Software build systems specify how source code is transformed into deliverables. Keeping build systems in sync with the software artifacts that they build while retaining their capacity to quickly produce updated deliverables requires a serious investment of development effort. Enticed by advanced features, several software teams have migrated their build systems to a modern generation of build technologies (e.g., Bazel, Buck), which aim to reduce the maintenance and execution overhead that build systems impose on development. However, not all migrations lead to perceived improvements, ultimately culminating in abandonment of the build technology. While prior work has focused on upward migration towards more advanced technologies, so-called downgrades, i.e., abandonment of a modern build technology in favour of a traditional one, remains largely unexplored.

In this paper, we perform an empirical study to better understand the abandonment of Bazel—a modern build technology with native support for multi-language software projects and (local/distributed) artifact caching. Our investigation of 542 projects that adopt Bazel reveals that (1) 61 projects (11.2%) have abandoned Bazel; and (2) abandonment tends to occur after investing in Bazel for a substantial amount of time (a median of 638 days). Thematic analysis reveals seven recurring reasons for abandonment, such as technical challenges, lack of platform integration, team coordination issues, and upstream trends. After abandoning Bazel, the studied projects have adopted a broad set of alternatives, spanning from language-specific tools like Go Build, to more traditional build technologies like CMake and even pure Make. These results demonstrate that choosing a build technology involves balancing tradeoffs that are not always optimized by adopting the latest technology. This paper also lays the foundation for future work on balancing the tradeoffs that are associated with build technology choice (e.g., feature richness vs. maintenance costs) and the development of tools to support migration away from modern technologies.

KEYWORDS

Build Systems, Downgrades, Empirical Software Engineering

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639169

ACM Reference Format:
Mahmoud Alfadel and Shane McIntosh. 2024. The Classics Never Go Out of Style: An Empirical Study of Downgrades from the Bazel Build Technology. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639169

1 INTRODUCTION

Build systems automate the process of transforming source code into deliverables by invoking tools (e.g., compilers, testing harnesses, code generators) in a dependency-based and configuration-dependent manner. Build systems are used to automate builds in development environments, e.g., within IDEs or on the CLI, and can even be used in deployment environments, e.g., where users invoke the build system to install open-source systems that are compiled from source. An effective build system helps to manage risk in software development by helping developers to detect compilation and integration problems early in the development cycle [12].

Maintaining a fast and robust build system requires a continuous investment of maintenance effort to stay in sync with the other software artifacts [32]. When software projects exceed the complexity that a developer can grasp, code changes can impact unexpected parts of the system, which lead to inconsistencies during the build and at runtime [28]. Moreover, as projects age, their build processes tend to slow down [26, 27]. Since build speed directly impacts the pace at which developers can test their changes, slow builds are a common complaint among developers [29].

In response to the maintenance and execution overhead of build systems, the software community has created modern build technologies, such as Bazel[1] and Buck.[2] Such build technologies provide native support for multi-language software projects, which are known to be problematic [36], and support local/distributed artifact caches to accelerate individual/team builds [13].

Prior work also shows that migrating to modern build technologies can be achieved with promising results [9, 30, 45, 47]; however, not all migrations lead to perceived improvements. Indeed, framework-driven build technologies (e.g., Maven) tend to be more tightly coupled to source code than more traditional technologies (e.g., Make, CMake) [31]. Moreover, framework-driven build technologies are often more prone to cloning-related issues than the more traditional technologies. This stems from strict version compatibility requirements leading to version conflicts and limited customization options. In contrast, the more traditional technologies offer a greater degree of control and flexibility, which contributes to their lower proneness to cloning-related pitfalls [33].

[1] https://bazel.build/
[2] https://buck.build/

Upward migrations that are not perceived improvements may influence stakeholders to abandon feature-rich build technologies. For example, within the KUBERNETES/KUBERNETES project,[3] the complexity and maintenance overhead of their Bazel build system led the community to abandon Bazel in favour of Go Build—a less feature-rich build technology. A similar event within the ISTIO/ISTIO project prompted a lively discussion about abandoning Bazel.[4]

Understanding why communities choose to migrate away from modern build systems can help stakeholders to adopt these build systems with realistic expectations and in a fashion that suits their needs. It can also help researchers and tool developers to focus on the most relevant barriers to adoption within these build systems. Hence, in this paper, we set out to study the prevalence of and rationale for downgrades of build technology. We conduct an empirical study of 542 open-source projects that adopt Bazel—one of the most (if not *the* most) popular modern build technologies available.

Prevalence (Section 3). First, we study the extent to which open-source projects that adopt Bazel end up abandoning it. We observe that 61 of the 542 studied projects (11.2%) that adopted Bazel abandoned it. The abandonment occurred in the studied projects despite maintaining Bazel specifications for a median of 638 days.

Rationale (Section 4). Next, to understand the reasons that led adopters to abandon Bazel, we conduct a thematic analysis on 212 records of commits, issue reports, and pull requests that document the final removal of Bazel specifications in the 61 projects that abandoned Bazel. We identify seven emergent themes, which span from technical challenges, such as the complexity of the build specifications and their maintenance, to a (perceived) lack of platform integration and interoperability, team coordination and onboarding challenges, and the influence of community trends.

Replacements (Section 5). Then, we analyze the build technologies that the 61 projects adopted after abandoning Bazel, observing a broad set of common replacements, including language-specific tools, such as Go Build, as well as more conventional options like CMake and even low-level build technologies like pure Make. These replacements are often less feature-rich, but more familiar to their community members, simplifying their maintenance.

Generalizability (Section 6). Finally, we explore the analytic generalizability of our themes by conducting a confirmatory study of a second downgrade case (from Gradle to other build technologies) and a contradictory upgrade case (from other build technologies to Bazel). We find that five of the seven emergent themes from the Bazel downgrade context also appear in the Gradle downgrade context, and that only one of the themes (the influence of community trends) appears in the contradictory Bazel upgrade context.

Contributions. This paper contributes: **(1)** *empirically grounded estimates* of the prevalence of build technology downgrades; **(2)** a *catalogue of seven emergent themes* that explain why 61 projects have downgraded their build systems; **(3)** a *list of popular replacement technologies* while shedding light on key factors that influence their adoption; and **(4)** *a checklist* that stakeholders can consult before adopting modern build technologies. To foster future work, we make our replication package publicly available.[5]

[3]https://github.com/kubernetes/kubernetes/pull/99561
[4]https://twitter.com/kelseyhightower/status/958834738650755072
[5]https://zenodo.org/records/10499104

2 STUDY DESIGN

In this section, we provide an overview of the most common build technologies (Section 2.1). Then, we describe our rationale for focusing on the Bazel build technology (Section 2.2) and explain our approach to project selection (Section 2.3).

2.1 Existing Build Technologies

Dozens of build technologies are available for communities to use.[6] Broadly speaking, these technologies have converged on various design paradigms [30, 42, 43]. The four most common ones are:

- *Low-level* technologies that require explicitly defined build dependencies (which may also be pattern-based or inferred) between each input and output file (e.g., Make [14]).
- *Abstraction-based* technologies that use high-level project information, such as the project name and the list of files to build, to generate low-level specifications (e.g., CMake).
- *Framework-driven* technologies that reduce "boilerplate" dependency expressions that are typical of low-level technologies in favour of conventions by expecting that, e.g., input and output files appear in predefined locations (e.g., Maven).
- *Modern* build technologies provide native support for multi-language software projects and support local/distributed artifact caches to accelerate individual/team builds [13]. Such build technologies differ from traditional systems by using fine-grained dependency management, monorepo support, and integrated static analysis (e.g., Bazel).

2.2 Studied Build Technology

To counteract increases in project complexity, organizationally-scaling build technologies have been created to build, test, and package large amounts of cross-language code simultaneously. In contrast to prior build technologies (e.g., Make, CMake, Maven), these modern ones use more abstraction for targets and sources, and provide native support for producing deliverables of common types, such as libraries, binaries, scripts, and datasets.

These modern technologies also natively support projects that are implemented in multiple languages, which makes them an attractive choice for managing the build process of so-called "monorepos" [34], i.e., where several related projects are versioned within a single repository. The entirety of the dependency graph is computed, inferred, and tracked to parallelize work, cache results, and mitigate non-determinism in build behaviour.

In our study, we choose to focus on the Bazel technology due to its popularity among modern build technologies. For example, compared to Buck and Pants (i.e., its closest competitors), Bazel is gaining more traction in the StackOverflow community,[7] with 3,229 questions posted (as of December 2023), whereas only 112 and 48 questions are tagged with Buck and Pants, respectively. Furthermore, our preliminary analysis in Section 3 reveals that the number of projects that have adopted Bazel exceeds the combined count of projects that have adopted Buck and Pants by a substantial margin, i.e., the number of projects adopting Bazel, Buck, and Pants in the World of Code corpus [24] is 542, 8, and 7, respectively.

[6]https://en.wikipedia.org/wiki/List_of_build_automation_software
[7]https://stackoverflow.com/

Table 1: Typical build technology paradigms and supported features (adapted from McIntosh et al. [30, 32]).

Feature	Low-level	Abstraction-based	Framework-driven	Modern
Manual build steps.	✓	✓	✓	✓
Configuration-based build.		✓	✓	✓
Cross-platform support.		✓	✓	✓
Convention-based build.			✓	
Scalability and optimizations.				✓
Caching/incremental builds.				✓
Language-agnostic.				✓
Support for monorepo.				✓
Example	Make	CMake	Maven	Bazel

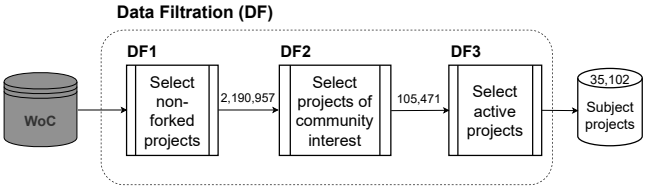

Figure 1: An overview of our data filtering approach.

Bazel is an open-source, declarative build system that was initially released by Google in 2015. Bazel is fast because it only rebuilds what is needed, which is achieved by its built-in support for local and distributed caching of both build artifacts and tests, along with an optimized dependency analysis.[1] Moreover, although Bazel already supports a plethora of platforms and languages, it can be extended to support other frameworks and languages.

2.3 Candidate Projects

Our goal is to study abandonment of the Bazel build technology. To perform our study, we need to collect a dataset of projects that have, at one point, adopted Bazel. It is important that we study a large sample of software projects in order to improve confidence in the conclusions that we draw. Hence, we analyze the large corpus of open-source version history collected in the World of Code (WoC) [24]. The WoC corpus is updated on a monthly basis and contains over 18 billion Git objects. We begin our analysis by querying the WoC corpus for projects that are hosted on GitHub. This query returns 7,731,242 public GitHub repositories.

Since GitHub hosts projects that are not yet mature or of sufficient complexity to warrant analysis, we apply three filters to select projects that are appropriate for our study. Figure 1 provides an overview of those filters, each of which we describe below.

DF1: Select non-forked projects. Forking a repository allows developers to experiment with changes without affecting the original project. We remove forks to reduce the quantity of duplicated project history, which can skew our results. This filter reduces our corpus to 2,190,957 candidate projects.

DF2: Select projects of community interest. We choose projects with at least ten stars, as such projects are considered of interest to the development community. Prior work [10] has shown that a ten-star threshold is a reasonable mechanism to remove most projects that are unlikely to be relevant for empirical studies. Also, a survey of over 700 developers shows that most developers consider the number of stars before using or contributing to GitHub projects [7]. This filter reduces our corpus to 105,471 candidate projects.

DF3: Select active projects. We select active projects to prevent the swaths of immature projects from impacting our conclusions. To detect projects with a high level of activity, inspired by prior work [15, 32], we plot the distributions of the number of commits and contributors per project. We select a threshold for them to ensure that our analyzed projects are representative and have attracted a contributor team of a size where build systems play an important coordination role. Due to space limitations, we relegate the corresponding plots to our replication package.[5] Selecting thresholds of 150 commits and 10 contributors reduces the corpus to 76,282 and then 35,102 candidate projects, respectively.

After applying the three filters, our corpus of candidate projects comprises popular and large projects from large organizations and communities, e.g., Google, Microsoft, Golang, Kubernetes, and Pytorch. Overall, the projects in our dataset have a rich development history (medians of 641 commits and 21 contributors).

3 PREVALENCE OF BAZEL ABANDONMENT

Previous studies have shown that migrating to more powerful build technologies is a promising solution to reduce maintenance overhead and build durations [47]. While newer technologies provide numerous features to support the build process, prior work (e.g., [31, 33]) demonstrates that they tend to induce more churn and be more tightly coupled to source code than traditional technologies, which exacerbates the overhead rather than mitigating it.

If a build technology is costing more than anticipated, it would not be unreasonable to explore alternatives. Thus, in this section, we study the degree to which Bazel is being abandoned in our corpus of projects. Our analysis focuses on the rate of Bazel abandonment (Section 3.1) and the duration of Bazel adoption (Section 3.2).

3.1 Rate of Bazel Abandonment

If Bazel is only rarely abandoned, it may not be a phenomenon worthy of further study. Therefore, we set out to study the rate at which projects that adopt Bazel have eventually abandoned it.

Approach. We perform two analyses to study the abandonment rate of Bazel among the studied projects. First, we examine whether each studied project is still using Bazel at the time when our analysis was performed. Typically, a project that adopts Bazel must contain at least one `BUILD.bazel` or `BUILD` file located in its root folder, which specifies what to build and how to build it.[1] Therefore, to identify projects that use Bazel, we first clone a local copy of the 35,102 repositories in our corpus. Then, we traverse the root directory of the `HEAD` commit of each cloned repository in search of a `BUILD.bazel` or `BUILD` file. If a match is found, we store the repository and the file for further inspection.

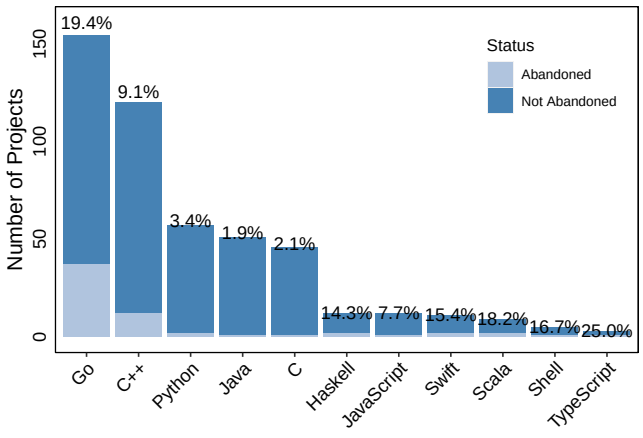

Figure 2: Bar-plot showing the distribution of both the number of projects abandoning and not abandoning Bazel, per programming language (the primary language). The rate of Bazel abandonment per language appears on top of each bar.

Our second analysis focuses on the projects that did not contain a Bazel specification. These projects either adopted Bazel in the past but abandoned it prior to the time of our analysis, or have never adopted Bazel. We set out to identify projects that abandoned Bazel and the commit where the final removal was performed. To do so, we first mine the commit records for each repository that does not contain a Bazel specification in the `HEAD` commit in reverse chronological order. We search the history of each repository for commits that delete the `BUILD.bazel` or `BUILD` file from the root directory. When we detect such commits, we store them for further inspection. For each removal commit, we extract (meta)data, such as the commit SHA, message, and timestamps.

Results. Of the 35,102 candidate projects, 542 adopted Bazel at one point in a project's lifetime. Although our primary focus in this paper is on Bazel abandonment, and not its overall popularity, we briefly discuss the proportion of Bazel projects that we detected next. While the rate of Bazel adoption (~1.5%) in our corpus may seem low, there are two factors that we believe illustrate the impact and importance of studying Bazel. First, among the 542 adopters are projects that are hosted by several organizations of influence in the software development community, such as Google, Microsoft, and Meta. Second, Bazel was only publicly released in 2015. When one also considers the crowded marketplace of build technologies (e.g., Wikipedia's incomplete list of build automation solutions[6] already lists 52 technologies), the fact that Bazel has reached 1.5% market share in our curated corpus of large, active, and popular open-source projects is impressive.

Turning to Bazel abandonment, we observe that while a total of 542 projects initially adopted Bazel at some point in the project's lifetime, **61 projects (11.2%) abandoned Bazel.**

Additionally, we stratify our analysis by programming language. To identify the main programming language of each studied project, we use the Linguist tool,[8] which is the tool that the GitHub platform uses to detect language usage. We invoke the tool once for each project, which generates a report of the percentage of code bytes written in each language. The language with the highest proportion of bytes is considered the main language of the project.

We find that the abandonment of Bazel varies across different languages, with the **highest abandonment rates being observed in Go and C++ projects.** Figure 2 illustrates the pattern of adoption and abandonment of Bazel per programming language. For example, among the projects that are mainly implemented in Go and initially adopted Bazel, 37 projects (19.4%) eventually abandoned Bazel. Similarly, in the case of C++ projects, 12 projects (9.1%) decided to abandon Bazel after being initially adopted.

For the other programming languages depicted in the figure, the rate of Bazel abandonment is relatively low, with only one or two projects each showing such a trend. It is important to note, however, that projects written in these languages had a lower overall rate of Bazel adoption when compared to the adoption rates of Go and C++ projects. For instance, for Python projects, of the 59 Python projects that initially adopted Bazel, two eventually abandoned Bazel, highlighting a modest rate of Bazel adoption in Python.

Moreover, we study the organizations that develop the projects that have abandoned Bazel, since abandonment may be unfairly inflated by immature organizations that were unprepared to adopt Bazel. To achieve this, we group the studied repositories that abandon Bazel by their organization name, which can be extracted from the URI of a GitHub repository. For example, the organization of the REDDIT/BASEPLATE.GO repository is REDDIT.

We discover that even well-established organizations are not exempt from choosing to abandon Bazel. Indeed, 20 of the 61 studied projects that abandoned Bazel are hosted by prominent organizations, such as Google, Meta, Kubernetes, Microsoft, Reddit, and Chromium. These findings suggest that even for large-scale projects, organizationally-scaling build technologies like Bazel may not always be the most suitable technology choice.

Finally, we also consider the number of contributors and project size to characterize projects that abandon Bazel. Below, we provide a summary of the results. We compute project size in Source Lines Of Code (SLOC) using `cloc` and extract the number of contributors using the GitHub API. We find that projects that abandoned Bazel have medians of 133,452 SLOC and 70.5 contributors. These metrics indicate that these projects are not small in terms of the volume of code or the number of contributors.

Complementary Analysis of Buck and Pants. As mentioned in Section 2.2, our primary focus remains on the popular Bazel build technology. Nonetheless, we also explore Buck and Pants using the same approach from Section 3.1. A project that adopts Buck must contain a `BUCK` file located in its root directory.[2] Similarly, projects that adopt Pants must have a `pants.toml` file in its root directory.[9] We only find eight projects using Buck and nine projects using Pants within our dataset. Furthermore, we find one project that abandoned Buck (12.5%), which is roughly on par with the rate of Bazel abandonment that we observed earlier in this section (11.2%). Nonetheless, the limited quantity of projects that have adopted Buck and Pants within our dataset prevent us from drawing stable conclusions about their rates of abandonment at this time.

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

[9] https://www.pantsbuild.org/

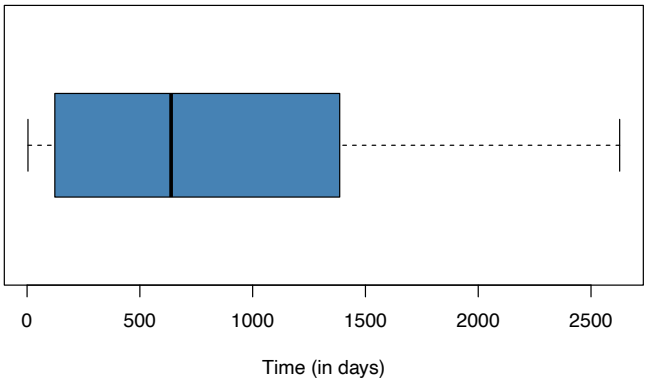

Figure 3: Box-plot showing the distribution of the duration of Bazel adoption.

3.2 Duration of Bazel Adoption

In Section 3.1, we observe that Bazel abandonment occurs in the studied projects; however, it is unclear if such projects have truly invested in their Bazel build systems before abandonment. Since adopting a build technology requires time for project maintainers to become familiar enough to perform required tasks, if the abandonment event happens too soon, it may have occurred due to premature decision making. Therefore, we study the duration of adoption of Bazel in the projects that have abandoned it.

Approach. To measure the duration of Bazel adoption, we count the number of days between the introduction date and the abandonment date of Bazel for each studied project where Bazel was eventually abandoned. To identify the introduction (removal) of Bazel, we mine each repository for the first (last) commit in which `BUILD.bazel` or `BUILD` file was added (removed).

Results. Figure 3 shows the distribution of the duration in which Bazel was adopted. From the figure, we observe that **Bazel remains adopted in the abandoning projects for a median of 638 days.** We also find that renowned projects had invested in Bazel for a considerable amount of time prior to abandonment. For example, the main Kubernetes project[10] started adopting Bazel on December 13th, 2016 and only abandoned it on February 28th, 2021. This is an investment of more than four years. Similarly, the Reddit Baseplate project[11] is another large and popular project that adopted Bazel for more than a year (19.6 months) before abandonment. This indicates that even after investing in it for a substantial amount of time, large and active projects can still end up abandoning Bazel.

> **Summary.** 11.2% of large and active projects in our dataset abandoned Bazel. The counts and rates of Bazel abandonment are largest in projects that are primarily implemented in Go and C++. Projects tend to abandon Bazel even after investing in it for a substantial amount of time (a median of 638 days).

[10] https://github.com/kubernetes/kubernetes
[11] https://github.com/reddit/baseplate.go

4 THEMATIC ANALYSIS

In this section, we set out to understand the underlying dynamics of abandonment of Bazel. More specifically, we set out to understand the rationale that the studied projects have for the abandonment of Bazel by analyzing historical records that document its removal.

Approach. First, for each of the 61 projects that have abandoned Bazel, we obtain links to community discussions, issue reports, and Pull Requests (PRs) that document Bazel abandonment by analyzing the set of commits that we collect in Section 3.1 (i.e., the commits that abandon Bazel). For example, *commit 14a0346*[12] in the Berty project describes their abandonment of Bazel. The PR that corresponds to the commit contains further detail and discussion about the Bazel removal event.[13] In some cases, we find that links from the commit record are insufficient to determine the cause of the abandonment. In such cases, we use GitHub's query feature to search for issues, discussions, and PRs that include the "bazel" keyword. We inspect each match to determine if it provides any context about the rationale for Bazel abandonment. In total, we collect a set of 212 documents from the 61 studied projects.

After collecting the set of documents that we deem to be relevant, we perform a systematic inspection and a *thematic analysis* [41]. We perform the analysis in multiple rounds. In the first round, the first and last authors meet and inspect the collected documents, focusing on their titles, description fields, and discussion threads, to generate *codes*, i.e., brief descriptions of the content that summarizes the reasons for abandoning Bazel. Subsequently, we identify common *themes* that span across the set of discovered codes. The themes that emerge group together codes that relate to common topics or that point to similar underlying concerns.

It is important for us to ensure that our themes are accurate representations of the abandonment events that they describe. Thus, we set out to ensure that (a) it is clear when a given theme should apply and (b) the themes are independent. Thus, in the second round of our analysis, the first and last authors independently examine abandonment documents, and label each project with the set of themes that apply. Note that although the themes are independent, they are not mutually exclusive, i.e., the same project can be labelled with more than one theme. In other words, a project may abandon Bazel for more than one (coded or thematic) reason.

Finally, when disagreements occurred, the authors met to discuss each one. For each disagreement, the two authors discussed it until a consensus was reached. In theory, the discussion may not produce a consensus. If no consensus was reached, a third rater would cast the deciding vote. In practice, a consensus was reached for each disagreement through discussion. No deciding vote was needed.

To evaluate our list of themes and assess the stability of the coding process, we calculate the Cohen's Kappa coefficient [8]. This statistic is commonly used to evaluate inter-rater agreement. The value of Cohen's Kappa ranges from −1.0 to +1.0, with values greater than 0 indicating a level of agreement that is greater than expected due to chance. We obtained a Kappa score of 0.812, which is considered to be an "excellent" level of agreement [23].

[12] https://github.com/berty/berty/commit/14a03463751db6bc7514964d12444e3c42be87c4
[13] https://github.com/berty/berty/pull/2127

Table 2: The extracted themes for Bazel abandonment. An example of each theme is provided in the footnotes.

ID	Theme & Example	Freq. (#)
T1	Difficulty in maintenance and troubleshooting.[14]	10 (20%)
T2	Lack of platform support and interoperability.[16]	13 (26%)
T3	Replacement by toolchain-native/platform-specific build technology.[4]	11 (22%)
T4	Experimental adoption of Bazel.[24]	2 (4%)
T5	Obstacles faced by external contributors or users.[31]	7 (14%)
T6	Bazel consumer is no longer required.[33]	3 (6%)
T7	Influence of upstream trends.[34]	13 (26%)

Results. Seven themes emerged to describe the reasons for Bazel abandonment. Table 2 summarizes the themes of Bazel abandonment that emerged during our inspection. Note that the total frequency of the identified reasons is greater than 100% because we observe that multiple themes can apply to the studied abandonment examples. Also, note that of the 61 projects that abandoned Bazel, we were not able to find relevant documents (e.g., PRs, issue reports) that explain the reason for abandonment in 11 projects. Below, we describe each emergent theme.

T1. Difficulty in maintenance and troubleshooting (10 projects). This theme describes cases where the project maintainers face difficulties maintaining Bazel files. Bazel's complex rule structure can make it difficult to understand why a build is failing and how to fix it. In such cases, projects face recurring and frequent build failures related to Bazel that are difficult for maintainers to fix. For example, *PR #23693*[14] reports a test failure due to an issue in Bazel configurations.

Through our inspection of such cases, we observe issue reports stating that maintaining Bazel is not practical when most of the code is implemented in a single programming language. In *issue #23796*,[15] the maintainer states that more than 77% of the code is implemented in Go, and the bulk of the build breakages of the project are not related to its own code, but rather to other Bazel rules that do not follow module properties in the `go.mod` file.

To gain a richer perspective on this theme, we examine whether abandonment of Bazel is associated with the proportion of code that is implemented in the most prevalent programming language within the project. We perform a statistical analysis to evaluate if it is likely that the project will abandon Bazel if the majority of the code is implemented in a single programming language. First, we group projects into those that, during the analysis period, (a) abandon Bazel or (b) continue to use Bazel. Then, we perform statistical tests comparing the projects that abandon Bazel to those that do not. For each project, we measure the proportion of code that is implemented in the primary programming language. To do so, we again use the Linguist tool.[8] We invoke the tool once for each project, which generates a report of the percentage of code bytes of each language. Note that for projects abandoning Bazel, we calculate the proportion of the primary language at the time of abandonment (i.e., the abandoning commit). For non-abandoning projects, we evaluate the latest snapshot.

[14]https://github.com/kubernetes/test-infra/pull/23693
[15]https://github.com/kubernetes/test-infra/issues/23796

Table 3: The proportion of bytes written in the primary language of the abandoning and non-abandoning projects.

Project type	Min.	Median	Max.
Abandoning	47.42	91.54	97.6
Non-abandoning	19.67	80.46	94.31

Table 3 lists minimum, median, and maximum proportions of code that is implemented in the primary language per project type. It shows that, the primary programming language comprises a median of 91.5% in the projects that abandoned Bazel, whereas the median for non-abandoning projects is eleven percentage points less. A Mann-Whitney U test (unpaired, two-tailed, $\alpha = 0.05$) indicates that a statistically significant difference exists between abandoning and non-abandoning projects ($p = 0.0143$). Furthermore, we find, that the Cliff's delta effect size [16] has a medium magnitude ($|\delta| = 0.391$). This statistical evidence supports our suspicion that abandonment of Bazel is associated with the proportion of code that is implemented using the most prevalent programming language. That is, projects that are more homogeneous in their implementation language tend to abandon Bazel at higher rates than projects that are more heterogeneous in their implementation languages.

T2. Lack of platform support and interoperability (13 projects). Another common reason why projects abandon Bazel is due to a lack of integration with other tools and systems that the development team is already using. For example, the microsoft/tensorflow-directml-plugin project replaced Bazel with CMake in *commit 44d7bef*.[16] The description of *PR #46*[17] explains challenges that the team encountered with respect to Bazel support for Windows and other UNIX-like platforms. The facebookresearch/CompilerGym project provides another example.[18] In the discussion of *PR #478*,[19] project maintainers described the challenges that they encountered when trying to use Bazel with the LLVM toolchain. Ultimately, the project team migrated their build system to CMake, which was known to integrate smoothly with their required external tools and libraries. A third example is present in the discussion of *PR #3* of the joneshf/terraform-provider-openwrt project.[20] The discussion describes the challenges that project maintainers encountered while trying to use Bazel on macOS devices.

T3. Replacement by toolchain-native/platform-specific build technology (11 projects). Another reason for Bazel abandonment is to replace it with a toolchain-native/platform-specific build technology. In these cases, the projects abandoned Bazel because a less complicated build technology has advanced, achieving parity with respect to Bazel features that compelled the projects to use Bazel in the first place. In such cases, discussions pointed out that Bazel was adding complexity without providing benefits to offset the cost.

[16]https://github.com/microsoft/tensorflow-directml-plugin/pull/46/commits/44d7bef871b92d34d39c2da5669fffbfaefcb61f
[17]https://github.com/microsoft/tensorflow-directml-plugin/pull/46
[18]https://github.com/facebookresearch/CompilerGym/issues/506#issuecomment-1058975790
[19]https://github.com/facebookresearch/CompilerGym/pull/478
[20]https://github.com/joneshf/terraform-provider-openwrt/pull/3

This complexity can make it difficult to understand how to correctly configure and use Bazel, leading to frustration and a lack of confidence in project build systems. An example of T3 can be seen in *PRs #652*[21] and *#903*[22] of the CORTEXTPROJECT/CORTEX project. As indicated by the discussion on *PR #672*,[23] the CORTEXPROJECT/-CORTEX project abandoned Bazel because one of the Go language releases (i.e., Go 1.10) accelerated (incremental) builds to a degree that the difference between Bazel and Go Build became negligible.

T4. Experimental adoption of Bazel (2 projects). This theme describes the challenges and consequences that arise when organizations or development teams experimentally adopt Bazel, encountering issues related to the lack of ongoing support and maintenance. For example, a contributor in the SKYPJACK/ENTT project suggested introducing Bazel. As discussed in *issue #287*,[24] maintainers state that they lack knowledge of Bazel, but they are still willing to merge a Bazel build system if the contributor agrees to maintain it. The contributor configured Bazel in textitcommit 90798c1.[25] Three months later, Bazel was removed due to lack of maintenance support from the original contributor.[26] In fact, *PR #430*[27] shows that the original contributor of the Bazel system later returned to the project, and suggested adding Bazel again.

In other cases, we find that the projects introduced Bazel as a secondary build system. In other words, the Bazel build system was added for specific purposes, such as its dependency caching feature or its capacity for producing faster builds, as an additional build system to be maintained in tandem with the other (official) build system. In fact, some projects (e.g., the main Kubernetes project[28]) reported that it is challenging to maintain multiple build systems simultaneously in their Bazel removal documents.

Build systems can also be used in a nested fashion, rather than in a parallel one. We discuss two examples as follow. The first one is parallel usage with Bazel (e.g., the SKYPJACK/ENTT project). In this case, a contributor suggested Bazel integration as another build tool to support the project, which was accepted by maintainers but later removed due to maintenance issues.[29] The second example is nested usage with Bazel and Make (e.g., the KUBERNETES/KUBERNETES project). In such a case, we observe the Kubernetes project using Bazel alongside Make through the "make bazel-build" rule.[30]

To gain a richer perspective on this theme, we examine whether projects adopting multiple build systems in parallel tend to abandon Bazel. We count the number of build systems in abandoning and non-abandoning projects. To do so, for each studied project, we inspect the repository documentation. Such resources often describe how to build and install the project in detail. We perform the evaluation on the abandoning commit for abandoning projects. For non-abandoning projects, we consider the latest snapshot.

[21]https://github.com/cortexproject/cortex/pull/652
[22]https://github.com/cortexproject/cortex/pull/903
[23]https://github.com/cortexproject/cortex/pull/652#issuecomment-397822795
[24]https://github.com/skypjack/entt/issues/287
[25]https://github.com/skypjack/entt/commit/90798c161b08a0ec24da268267d58b3d327bf776
[26]https://github.com/skypjack/entt/commit/99f81e82d57fbe21e18e1f2ff26e37363b8992d2
[27]https://github.com/skypjack/entt/pull/430
[28]https://github.com/kubernetes/kubernetes/issues/88553
[29]https://github.com/skypjack/entt/issues/287
[30]https://github.com/kubernetes/kubernetes/commit/f73254826d1a8c22b215bc25d60fe724f93e4221

Table 4: The number of build systems being maintained in the abandoning and non-abandoning projects.

Project type	Min.	Median	Max.
Abandoning	1	2	2
Non-abandoning	1	1	2

Table 4 provides an overview of the number of build systems adopted in the abandoning and non-abandoning projects. The table shows that abandoning projects adopt a median of two build systems simultaneously, whereas non-abandoning projects adopt a median of one. A Mann-Whitney U test (unpaired, two-tailed, $\alpha = 0.05$) indicates a statistically significant difference between the number of build systems adopted in the abandoning and non-abandoning projects ($p = 1 \times 10^{-7}$). Moreover, the magnitude of the Cliff's delta effect size is large ($|\delta| = 0.872$).

T5. Obstacles faced by external contributors or users (7 projects). Another reason that Bazel is being abandoned is that projects find Bazel presents challenges for onboarding new contributors. Such projects abandon Bazel to remove this barrier to entry for their communities. Also, projects report that Bazel has a smaller community of users and contributors than other build technologies, which can make it difficult to find help when working with Bazel. For example, the PIPE-CD/PIPECD project abandoned Bazel in *commit 6e8eb9b*[31] due to this very reason.

Issue #1634[32] of the PIPE-CD/PIPECD project explains that the biggest disadvantage of adopting Bazel was the learning curve. They argue that Bazel has a steeper learning curve compared to other build technologies, and it requires more configuration than other build technologies, which is perceived to be a barrier to adoption.

T6. Bazel consumer is no longer required (3 projects). Another reason to abandon Bazel is when a project has a client project that depends on it, and that client is built using Bazel. The client's adoption of the project may be contingent on the project also being built using Bazel. Bazel abandonment can occur if that client ceases to build using Bazel or no longer depends on the project in question. For example, *issue #4280*[33] of the QUANTUMLIB/CIRQ project explains that their Bazel build system can be retired because the TFQ client (TENSORFLOW/QUANTUM) no longer depends on their project.

T7: Influence of upstream trends (13 projects). In several cases, the abandonment happens for the sake of following a trend set by upstream projects. For example, the ISTIO/TEST-INFRA project abandoned Bazel in *PR #4116*[34] because the other repositories being developed by the Istio organization had already abandoned Bazel.[35]

> **Summary.** Emergent themes of Bazel abandonment span from technical challenges, such as the complexity of the build specifications and tool integration (T1, T2, T3, T6), to team coordination and onboarding challenges (T4, T5) and community trends (T7).

[31]https://github.com/pipe-cd/pipecd/commit/6e8eb9b294f7a27422039beec0cc1b50886996df
[32]https://github.com/pipe-cd/pipecd/issues/1634
[33]https://github.com/quantumlib/Cirq/issues/4280
[34]https://github.com/istio/test-infra/pull/4116
[35]https://github.com/istio/test-infra/issues/1580

Kubernetes Case Study

To enrich our qualitative analysis, we perform an in-depth study of the rationale for Bazel abandonment in the main Kubernetes project. We choose to study the main Kubernetes project (i.e., KUBERNETES/KUBERNETES[36]) because it is a large, thriving, and actively maintained system. The project initially employed both Bazel and Go Build to specify their build process in 2017. Four years later, the project abandoned Bazel. Furthermore, the abandonment of Bazel by Kubernetes has had an influence on the abandonment of Bazel in other Kubernetes subprojects.[37,38] The abandonment by Kubernetes has even been referenced in projects beyond its subprojects, showcasing its wide-ranging influence.[39] We cross-reference our findings with the insights that we derived from the inspected documents. Below, we discuss each emergent abandonment theme that applies to the Kubernetes project.

T1. Difficulty in maintenance and troubleshooting. In Kubernetes, maintenance challenges with respect to three types of tools were implicated in the abandonment of Bazel abandonment.

Linters. Linters (i.e., tools that perform lightweight code analyses to identify potential issues) are typically configured using a dedicated Go module to simplify handling dependencies separately from those needed for the official project deliverables. Decomposing the project into separate Go modules for linter configuration and project deliverables helps to cleanly separate these concerns. Kubernetes uses a build tool called `rules_go` (part of Bazel) to build Go code, but it does not handle Go projects that are decomposed into modules seamlessly. When Go projects are organized into multiple modules, `rules_go` creates build maintenance challenges. For instance, handling dependencies may become complex during the coordination of updates across modules.

Code Generators. A key component of the Kubernetes build process is code generation. While Bazel can invoke code generators, the developers often commit the generated code to be consumed by non-Bazel users (i.e., external projects). As a result, Kubernetes developers have avoided using Bazel's code generation features to re-generate templated code during the build process. Making matters worse, they observed that the generators that they use are largely incompatible with Bazel. For example, Kubernetes code generation is designed to load the entire Go codebase and subsequently generate all required code in a single operation. However, this approach poses challenges as it is incompatible with Bazel's feature, which generates code for each package individually.

Build Systems Themselves. For a period, the Kubernetes project maintained both Go build and Bazel build systems. However, any inconsistencies in the configuration of the two build systems led to differences in the Kubernetes deliverables that were being produced. Consequently, it became a priority to meticulously maintain both sets of build specifications such that identical builds would be produced. This required considerable effort to not only synchronize the build processes, but also to maintain clear developer and user documentation for two build systems.

[36]https://github.com/kubernetes/kubernetes
[37]https://github.com/kubernetes/test-infra/pull/26039
[38]https://github.com/istio/test-infra/pull/4116
[39]https://github.com/cert-manager/cert-manager/issues/4030

T2: Lack of platform support and interoperability. Kubernetes developers have reported that integrating Go code into a Bazel-based project can require additional effort, as Bazel and Go do not have a consistent nomenclature for managing dependencies and reasoning about build commands (Kubernetes is primarily written in Go). Moreover, ensuring that specific platforms and tools (e.g., macOS machines with M1 CPUs) integrate correctly with Bazel can also present non-trivial technical challenges for contributors.

T5. Obstacles faced by external contributors and users. Kubernetes developers also sought to simplify the developer experience when they abandoned Bazel. Bazel builds are specified in `BUILD` files (written in Starlark), which differs from Go Build files like `go.mod` or `go.sum` (written in Go). Since Go has its own ecosystem and tools, such as `go get` and `go build`—tools that are more likely to be familiar for Go developers—shifting contexts to Bazel's `BUILD` files may introduce overhead for contributors.

5 REPLACEMENT BUILD TECHNOLOGIES

In this section, we study the alternative build technologies that have been adopted after Bazel abandonment. These replacements can serve as examples to developers navigating similar transitions in their projects, seeking alternatives to modern build technologies like Bazel. We are particularly interested in studying the extent to which the replacement technologies offer feature parity with Bazel.

Approach. To identify the build technologies that have been adopted by projects after abandoning Bazel, we again inspect the project artifacts. Building upon our prior analysis in Section 4, where we examine 212 documents that are associated with the abandonment of Bazel, we find that these documents often contain information about the post-abandonment build technology.

In some cases, the abandonment documents do not provide sufficient detail to determine the replacement build technology. Thus, we also inspect other project artifacts, such as the README and CONTRIBUTING files, as well as the main project website, to identify the current build technology that has been adopted by each studied project. For instance, in the GOOGLEFORGAMES/OPEN-SAVES project,[40] although Bazel was abandoned, the inspected document did not explicitly specify the adopted build technology; however, the README file[41] contains detailed instructions that explain how to build the project from which we can infer that the replacement technology is Make. In certain instances, when documentation does not clearly explain the replacement build technology, we detect suspected replacements by searching for files with names that are known to map to particular build technologies, such as `build.gradle` (Gradle), `Makefile` (Make), or `CMakeLists.txt` (CMake) within the project repository.

Results. Table 5 provides the list of build technologies that replaced Bazel in the studied projects. As the table shows, **abandoning projects tend to migrate to less feature-rich alternatives.** Specifically, we observe an association between certain languages and build technologies (**D1: language-specific**). For instance, Go Build—a build solution specifically for pure Go projects—naturally

[40]https://github.com/googleforgames/open-saves/pull/41#issuecomment-627033775
[41]https://github.com/firedancer-io/firedancer/blob/main/doc/getting-started.md

Table 5: Build technologies adopted after Bazel abandonment.

Domain	Build Technology	# Projects	Language
D1	Go Build	28	Go
	SPM	2	Swift
	Mage	1	Go
	SBT	1	Scala
	Gradle	1	Java
	Setuptools	1	Python
D2	CMake	11	C++, Go, Python, Shell
	Make	8	Go, JavaScript, C
D3	Nix	4	Go, TypeScript, Scala, Haskell
	Google Cloud Build	4	Go
Total		**61**	

attracted most projects that are primarily implemented in Go (28 projects). Similarly, SPM (Swift Package Manager) replaced Bazel in two projects that are primarily implemented in Swift. In addition, we find one project that has adopted Mage—a Makefile alternative for Go projects[42]—and one that has adopted Setuptools[43]—a Python-specific build technology.

Surprisingly, 19 projects replaced Bazel with conventional and low-level build technologies (**D2: conventional and low-level**), including CMake (11 projects) and pure Make (8 projects). These replacements are far less feature-rich than Bazel, but are well understood within the development communities (cf. T5 in Section 4). Indeed, this tradeoff between feature richness of a modern technology like Bazel and the complexities of its maintenance in an open-source community that must attract and retain contributors to grow seems to have influenced abandonment decisions.

Furthermore, eight projects replaced Bazel with build technologies that are tailored to particular deployment platforms (**D3: platform-specific**). For example, Google Cloud Build—a cloud-based build solution for the Google Cloud Platform—was adopted by four projects. In fact, we find that the four projects that adopted Google Cloud Build implement cloud-based services, specifically in the context of managing Kubernetes cluster deployments on other cloud platforms, e.g., the KUBERNETES-SIGS/CLUSTER-API-PROVIDER-AZURE project facilitates the provisioning and management of Kubernetes clusters on Microsoft Azure.

In addition, four projects replaced Bazel with Nix, which is not a build technology per se, but rather a package manager for the NixOS platform. The choice to replace a build technology with a packaging script is primarily motivated by Nix's simplified dependency management and enhanced platform stability. For instance, *PR #899*[44] of the BRENDANHAY/AMAZONKA PROJECT explains that Bazel was abandoned in favour of Nix to streamline dependencies and prevent potential CI failures by reducing complexities.

> **Summary.** Projects that abandoned Bazel tend to migrate to less feature-rich build technology alternatives, spanning language-specific tools (e.g., Go Build and SPM), conventional and low-level options (e.g., CMake and pure Make), and platform-specific technologies like Google Cloud Build and Nix.

[42]https://github.com/magefile/mage
[43]https://setuptools.pypa.io/en/latest
[44]https://github.com/brendanhay/amazonka/pull/899

6 ANALYTIC GENERALIZABILITY

In this section, we explore the analytic generalizability [50] of our themes by studying a confirmatory downgrade case from Gradle to other build technologies (Section 6.1) and a contradictory upgrade case from other build technologies to Bazel (Section 6.2).

6.1 Confirmatory Downgrade Case

Approach. Projects that adopt Gradle must contain a `build.gradle` file located in its root directory. We follow the same approach in Section 3.1 to identify projects that abandoned Gradle within our curated set of projects. To verify that the detected commits are actually indicating the abandonment of Gradle, we inspect the corresponding GitHub commit records, as well as referenced PRs and issue reports to ensure that they discuss a true removal of Gradle. Then, we follow our approach of thematic analysis in Section 4 to discover the rationale for abandonment by analyzing historical records that document the removal of Gradle.

Results. We find that the phenomenon of build system downgrades is not exclusive to Bazel. Of the 413 projects that initially adopted Gradle, 7.1% eventually abandoned it for less feature-rich alternatives (e.g., Maven, SBT). This finding aligns with our observations regarding Bazel, where we identified an abandonment rate of 11.2%. Furthermore, our manual analysis of the projects that have abandoned Gradle reveals that the themes that we observe in Bazel abandonment context also apply to these projects that abandon Gradle. Specifically, we find examples of T1, T2, T3, T5, and T7 among the projects that have abandoned Gradle.

6.2 Contradictory Upgrade Case

Approach. We follow the same approach in Section 3.1; however, instead of searching for projects that have abandoned Bazel, we search for projects that have upgraded to Bazel. Note that a project might initially adopt Bazel as a complement to their existing build technology rather than as a replacement (cf. the case study of the Kubernetes project in Section 4).

We begin by identifying the latest commit that introduces a Bazel file (i.e., `BUILD.bazel` or `BUILD` in the root folder). To verify that the detected commits are actually indicating the adoption of Bazel as a replacement for another build technology, we inspect the corresponding GitHub commit records, as well as referenced PRs and issue reports. Then, we follow our approach of thematic analysis in Section 4 to discover the rationale for Bazel adoption by analyzing historical records that document the upgrade to Bazel.

Results. Of the 481 projects that adopted Bazel, 439 projects had Bazel as their initial build system. However, 8.7% of the projects transitioned from other build technologies to Bazel. The primary themes that emerged to describe the rationale for upgrading from other build technologies to Bazel include: **(1)** improved dependency management, **(2)** build acceleration through remote caching, **(3)** enabling downstream builds for client projects that have already adopted Bazel, and **(4)** the influence of upstream trends. We can observe that only one emergent theme (the influence of upstream trends) that overlaps with the themes that we identified in cases of Bazel abandonment. This suggests that the reasons for abandoning Bazel tend to differ from those that drive Bazel adoption.

7 THREATS TO VALIDITY

Internal Validity. Threats to internal validity are related to the experimenter bias and errors. We conduct a thematic analysis in our study to investigate the most common reasons for Bazel abandonment. This analysis is subject to inspector bias. We mitigate this threat by having two authors independently code the reasons for abandoning Bazel and calculate the inter-rater agreement (Cohen's Kappa coefficient). The level of agreement (> 0.8) indicates that our results are likely robust.

External Validity. Threats to external validity relate to the generalizability of our findings. Although we study a corpus of 542 projects that have adopted Bazel (61 of which have abandoned it), we focus on open-source projects that are hosted on GitHub. As such, our results may not generalize to other open-source or proprietary repository hosts. Yet even if our results remain constrained to the studied projects, the set contains several of the most popular projects that are well-renowned in the development landscape.

Construct Validity. We rely on the Linguist tool, which is known to have been initially prone to errors like misclassifying TypeScript languages [5]. The specific errors were addressed in release v2.10.10.[45] Our analysis, based on the latest release in September 2023 (v7.24.0), produced consistent results across projects.

8 RELATED WORK

In our estimation, the most closely related work falls into build maintenance (Section 8.1) and migration (Section 8.2) categories.

8.1 Build Maintenance

Prior work focused on build maintenance effort and how it evolves across different build systems [11, 22, 25, 27, 30]. For example, McIntosh *et al.* [30] found that build maintenance imposes up to a 27% overhead on source code development and a 44% overhead on test development. Hochstein *et al.* [22] found that there is a hidden overhead associated with maintaining build systems. Macho *et al.* [25] proposed an approach to extract fine-grained build changes from Maven specifications, which achieved an average precision of 97%.

Other work focused on proposing frameworks to reduce the overhead imposed by build maintenance [2, 18, 46]. For example, Adams *et al.* [2] proposed MAKAO, a reverse-engineering framework for Make-based build systems. MAKAO constructs a build dependency graph by parsing the trace output produced by a build execution. It displays a Make dependency graph using colour coding, configurable layouts, and zooming, allowing dependency information in the graph to be queried and filtered. Tamrawi *et al.* [46] proposed SYMake, a tool for analyzing Make specifications. SYMake produces a Symbolic Dependency Graph (SDG) by statically analyzing Make code. SYMake extracts a model that describes how build artifacts are initialized and manipulated via the build process. The tool can detect several types of code smells and errors in Makefiles. Formiga [18] is another build maintenance tool for the `Ant` build technology, which focuses on source file changes that require build maintenance. Formiga allows dependencies created during the build process to be identified at a fine granularity.

Bezemer *et al.* [6] developed an approach to detect unspecified dependencies in Make-based build systems. Empirical studies of four open-source projects showed that unspecified dependencies are not uncommon. Sotiropoulos *et al.* [44] present Buildfs, which also uncovers faults related to incremental and parallel builds. AlKofahi *et al.* [4] extract the semantics of build specification changes using MkDiff to aid in change comprehension.

Our study differs from the prior work both in terms of its target and its scope. First, the target of our study is Bazel—a modern, organizationally-scaling build technology. Bazel was designed to address limitations in the technologies that were the focus of prior work (e.g., Make), such as local/distributed artifact caches to accelerate individual/team builds and first-class support for common deliverable types, spanning from executables to datasets. Second, the focus of our study is on (open-source) projects that have migrated away from Bazel to less feature-rich, traditional build technologies.

8.2 Build Migration

Prior work has studied migration between build technologies [1, 32, 45]. For example, McIntosh *et al.* [32] studied projects that migration upwards toward more feature-rich technologies (i.e., Ant to Maven and Make/Autotools to CMake), observing that they often pay off in terms of quantitative indicators of build maintenance activity (e.g., rate of change, logical coupling with source code). Suvorov *et al.* [45] mined the developer mailing lists in KDE and the Linux kernel to understand their build migration projects.

Other work proposed solutions to aid with build migrations [3, 17, 47]. For example, Westfelt and Aleksandrauskas [47] provided tool support to aid in build migrations from Make to Bazel. AlKofahi *et al.* [3] proposed a platform to identify configuration settings that exercise different parts of Makefiles. Gligoric *et al.* [17] proposed Metamorphosis, a dynamic approach to automate the migration of build specifications to a new build technology.

Similar to build technologies, CI tools compete for adoption in the development marketplace. Mazrae *et al.* [40] performed a qualitative analysis of CI tool usage based on interviews with 22 experienced practitioners. The study revealed diverse reasons for usage and a shift towards cloud-based solutions. Other recent studies focused on barriers to the adoption of CI services [20, 37, 48, 49]. For example, Hilton *et al.* [19–21] studied the benefits and costs of using CI at all, observing that practitioners face problems, such as increased complexity and new security concerns when working with Travis CI [19]. Widder *et al.* [48, 49] observed that the use of CI services imposes costs (e.g., resource usage) and that projects that use language toolchains with limited support from CI services are more likely to abandon them.

The aforementioned studies have explored the abandonment of CI services, which differs substantially from build systems. Build systems ensure the repeatable and optimal assembly and testing of project deliverables, while CI services make the integration process routine for providing rapid feedback and producing official releases. This fundamental shift in scope implies that downgrades of build technology will differ substantially from that of CI services. Our approach also differs from that of prior work, as we analyze commit records, inspect issue reports, and analyze PR discussions, providing a fresh perspective on build technology abandonment.

[45] https://github.com/github-linguist/linguist/releases/tag/v2.10.10

Nonetheless, perhaps the most similar work to ours is that of Widder *et al.* [49], who discovered themes of abandonment of the Travis CI platform. The themes of Bazel abandonment that we discover both complement and extend that list of themes. First, T1, T2, and T5 complement Widder *et al.*'s themes of build failures, unsupported technology, and poor user experience, respectively, extending them to the new context of abandonment of modern build technology. The fact that similar themes emerge in our context suggests that these themes may be fundamental to all release automation, but further studies are needed to confirm this. In addition, T3, T4, T6, and T7 are entirely unique to our studied context, further deepening our understanding within the build abandonment scope.

9 CONCLUSION AND LESSONS LEARNED

In this paper, we study the prevalence of and rationale for the abandonment of Bazel—a modern build technology—in large, active, and popular open-source projects. We also investigate which technologies replaced Bazel in the studied projects. Below, we distill lessons for the development, research, and tool building communities, as well as discussing promising opportunities for future research.

On Development. ***The latest build technologies are not without limitations, and may not be the optimal choice for all projects.*** Although prior work shows that migrating to advanced build technologies could be promising to reduce the maintenance overhead of build systems [30, 47], our results show that it is not always the case. We find that 11.2% of the studied Bazel projects downgraded to a less powerful technology, citing challenges, such as the complexity of the build specifications, a lack of tool and platform support, and team coordination and onboarding (Section 4). There are situations where these costs do not outweigh the perceived benefits. For example, the abandonment themes that we identify can form the basis for a checklist for avoiding some common challenges of Bazel adoption. Below, we provide examples of checklist items that we formulate based on our study results.

- *Is the project implemented in multiple languages (T1)?* If not, it is unlikely that the project will benefit sufficiently from Bazel, since other key features (e.g., dependency caching) are provided by similar tools (e.g., Gradle) nowadays.
- *Is there an alternative tool that has introduced similar desirable features (e.g., dependency caching) and is easier to configure (T3)?* Our results show that adopting Bazel for common features, such as dependency caching is often associated with abandonment. Platform-specific tools that provide similar features with less configuration will likely be easier to adopt.
- *If Bazel is adopted, will it be handled by a champion maintainer (T4)?* Our inspection shows that when a project has only one developer capable of maintaining Bazel specifications, once the maintainer leaves, the project cannot sustain the Bazel system. Projects that plan to adopt modern build tools like Bazel are encouraged to have multiple maintainers familiar with Bazel to minimize the risk of turnover-induced knowledge loss [35, 38, 39].
- *Does the project maintain another build system in parallel (T4)?* If yes, the project will likely face difficulties keeping both build systems in sync. Our results demonstrate that this is a recurring reason for the abandonment of Bazel.
- *How likely is it that potential contributors will understand how to configure and maintain the build system (T5)?* Our results demonstrate that external contributors are often deterred from projects when complex Bazel specifications are being maintained. Therefore, if a project is interested in attracting such contributors, it would be prudent to adopt a more broadly understood build technology.

On Research. ***Researchers should explore ways that assist projects with recommendations prior to adopting build technologies.*** Our results show that large and mature projects abandoned Bazel, even after they had invested in Bazel for a considerable amount of time (Section 3), suggesting that project maintainers need assistance when adopting a modern build technology. Researchers should explore techniques for helping projects to make pragmatic decisions prior to adopting feature-rich build technologies. One initial approach would be to train machine learning or statistical regression models based on features related to project characteristics. One simple example of potential project features is the proportion of project code that is implemented in the most prevalent programming language in the project. We observe significant differences in these proportions between the projects that abandon Bazel and the projects that do not (Section 4)

On Tools. ***Tool builders should supply development teams with tools to support the downgrade transition.*** Prior work shows that build migration requires substantial effort and resources [45]. To reduce this imposed overhead, the community has developed tools to (partially) automate the upgrade process. For example, Grazel[46] is a tool to migrate Android projects from Gradle to Bazel incrementally and automatically. It generates `BUILD.bazel` and other relevant specifications for a given Android project. Teams that want to migrate away from Bazel face similar challenges. Indeed, we observe that Bazel abandonment involves carefully editing and removing a large number of files. For example, when the main Kubernetes project migrated away from Bazel, 3,355 files[47] needed to be changed and/or removed.

On Bazel. ***Organizations are facing common integration challenges with other build tools and platforms.*** Our results show that Bazel abandonment themes include cases where project maintainers face technical difficulties resolving Bazel issues that appear during build time (T1). For example, we observe that maintainers tend to report that Bazel is particularly problematic when using Go toolchains, e.g., Bazel requires a strict separation between dependency information from the code, while Go tooling is flexible in locating dependencies. In other cases, projects face challenges when integrating Bazel with external tools and systems (T2). For example, project maintainers found it challenging to correctly configure Bazel on a (typical) macOS device. If the Bazel community could provide support for these sorts of issues, it would likely mitigate challenges that are encouraging users to migrate to alternatives.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[46] https://github.com/grab/Grazel
[47] https://github.com/kubernetes/kubernetes/pull/99561/files

REFERENCES

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2012. The evolution of the linux build system. *Electronic Communications of the EASST* 17 (2012), 578–608.

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

[3] Jafar Al-Kofahi, Tien N Nguyen, and Christian Kästner. 2016. Escaping AutoHell: a vision for automated analysis and migration of autotools build systems. In *Proceedings of the International Workshop on Release Engineering*. 12–15.

[4] Jafar M Al-Kofahi, Hung Viet Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2012. Detecting semantic changes in makefile build code. In *Proceedings of the International Conference on Software Maintenance*. 150–159.

[5] Emery D Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems* 41, 4 (2019), 1–24.

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

[7] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[8] James B Campbell and Randolph H Wynne. 2011. *Introduction to remote sensing*. Guilford Press.

[9] Maria Christakis, K Rustan M Leino, and Wolfram Schulte. 2014. Formalizing and verifying a modern build language. In *Proceedings of the International Symposium on Formal Methods*. 12–15.

[10] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *Proceedings of the International Conference on Mining Software Repositories*. 560–564.

[11] Casimir Désarmeaux, Andrea Pecatikov, and Shane McIntosh. 2016. The dispersion of build maintenance activity across maven lifecycle phases. In *Proceedings of the International Conference on Mining Software Repositories*. 492–495.

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

[13] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the International Conference on Software Engineering Companion*. 11–20.

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

[15] Keheliya Gallaba and Shane McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis) use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2018), 33–50.

[16] Jean Dickinson Gibbons and Subhabrata Chakraborti. 2014. *Nonparametric statistical inference*. CRC press.

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

[18] Ryan Hardt and Ethan V Munson. 2013. Ant build maintenance with formiga. In *Proceedings of the International Workshop on Release Engineering*. 13–16.

[19] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 197–207.

[20] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the International Conference on Automated Software Engineering*. 426–437.

[21] Michael C Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, and Darko Marinov. 2016. Continuous Integration (CI) Needs and Wishes for Developers of Proprietary Code. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*.

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

[23] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

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

[25] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. 2021. The nature of build changes: An empirical study of Maven-based build systems. *Empirical Software Engineering* 26, 3 (2021).

[26] Shane McIntosh, Bram Adams, and Ahmed E Hassan. 2010. The evolution of ANT build systems. In *Proceedings of the International Conference on Mining Software Repositories*. 42–51.

[27] Shane McIntosh, Bram Adams, and Ahmed E Hassan. 2012. The evolution of Java build systems. *Empirical Software Engineering* 17, 4 (2012), 578–608.

[28] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E Hassan. 2014. Mining co-change information to understand when build changes are necessary. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 241–250.

[29] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E Hassan. 2016. Identifying and understanding header file hotspots in c/c++ build processes. *Automated Software Engineering* 23, 4 (2016), 619–647.

[30] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of the International Conference on Software Engineering*. 141–150.

[31] 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 (2015), 1587–1633.

[32] 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 (2015), 1587–1633.

[33] 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 International Conference on Software Engineering*. 145–154.

[34] PJ McNerney and PJ McNerney. 2020. Code Organization and Bazel. *Beginning Bazel: Building and Testing for Java, Go, and More* (2020), 97–113.

[35] Mathieu Nassif and Martin P Robillard. 2017. Revisiting turnover-induced knowledge loss in software projects. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 261–272.

[36] Andrew Neitsch, Kenny Wong, and Michael W Godfrey. 2012. Build system issues in multilanguage software. In *Proceedings of the International Conference on Software Maintenance*. 140–149.

[37] Gustavo Pinto, Marcel Rebouças, and Fernando Castor. 2017. Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*. 74–77.

[38] Peter C Rigby, Yue Cai Zhu, Samuel M Donadelli, and Audris Mockus. 2016. Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya. In *Proceedings of the International Conference on Software Engineering*. 1006–1016.

[39] Martin P Robillard. 2021. Turnover-induced knowledge loss in practice. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1292–1302.

[40] Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. 2023. On the usage, co-usage and migration of CI/CD tools: A qualitative analysis. *Empirical Software Engineering* 28, 2 (2023), 52.

[41] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.

[42] Mike Shal. 2009. Build system rules and algorithms. *gitup. org* (2009).

[43] Peter Smith. 2011. *Software build systems: principles and experience*. Addison-Wesley Professional.

[44] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A model for detecting faults in build specifications. *ACM on Programming Languages* 4, 1–30.

[45] Roman Suvorov, Meiyappan Nagappan, Ahmed E Hassan, Ying Zou, and Bram Adams. 2012. An empirical study of build system migrations in practice: Case studies on kde and the linux kernel. In *Proceedings of the International Conference on Software Maintenance*. 160–169.

[46] 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*. 650–660.

[47] Vidar Westfelt and Arturas Aleksandrauskas. 2019. Automated migration of large-scale build systems.

[48] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2018. I'm leaving you, Travis: a continuous integration breakup story. In *Proceedings of the International Conference on Mining Software Repositories*. 165–169.

[49] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A conceptual replication of continuous integration pain points in the context of Travis CI. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 647–658.

[50] Robert K Yin. 2009. *Case study research: Design and methods*. Vol. 5. sage.