# Studying the Software Development Overhead of Build Systems

by

Shane McIntosh

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
August 2015

# Abstract

Software is developed at a rapid pace. Software development techniques like continuous delivery have shortened the time between official releases of a software system from months or years to a matter of minutes. At the heart of this rapid release cycle of continuously delivered software is the *build system*, i.e., the system that specifies how source code is translated into deliverables. An efficient build system that quickly produces updated versions of a software system is required to keep up with market competitors. However, the benefits of an efficient build system come at a cost — build systems introduce overhead on the software development process.

In this thesis, we use historical data from a large collection of software projects to perform four empirical studies. The focus of these empirical studies is on two types of software development overhead that are introduced by the build system.

We first present three empirical studies that focus on the *maintenance overhead* introduced by the need to keep the build system in sync with the source code that it builds. We observe that: (1) although modern build technologies like Maven provide additional features, they tend to be prone to additional build maintenance activity and more prone to *cloning*, i.e., duplication of build logic, than older technologies like `make` are; (2) although typical cloning rates are higher in build systems than in other software artifacts (e.g., source code), there are commonly-adopted patterns of creative build system abstraction that can keep build cloning rates low; and (3) properties of source and test code changes can be used to train accurate classifiers that indicate whether a co-change to the build system is necessary.

We then present an empirical study that focuses on the *execution overhead* introduced by the slow nature of (re)generating system deliverables using a build system. We find that build optimization effort: (1) will yield more build performance improvement by focusing on *build hotspots*, i.e., files that are not only slow to rebuild, but also tend to change frequently; and (2) should be aligned with architectural refinement in order to yield the most benefit.

# Related Publications

Earlier versions of the work in this thesis were published as listed below:

- **A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance** (Chapter 4). Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. Springer Journal of Empirical Software Engineering (EMSE), 47 pages. In Press.

- **Collecting and Leveraging a Benchmark of Build System Clones to Aid in Quality Assessments** (Chapter 5). Shane McIntosh, Martin Poehlmann, Elmar Juergens, Audris Mockus, Bram Adams, Ahmed E. Hassan, Brigitte Haupt, and Christian Wagner. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Software Engineering In Practice (SEIP), pp. 145–154.

- **Mining Co-Change Information to Understand when Build Changes are Necessary** (Chapter 6). Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pp. 241–250.

- **Identifying and Understanding Header File Hotspots in C/C++ Build Processes** (Chapter 7). Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. Springer Journal of Automated Software Engineering (AUSE), 29 pages, In Press.

The following publications are not directly related to the material in this thesis, but were produced in parallel to the research performed for this thesis.

- **An Empirical Study of goto in C Code**, Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. To appear in Proceedings of the 10th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE 2015), 12 pages.

- **An Empirical Study of the Impact of Modern Code Review Practices on Software Quality**. Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. Springer Journal of Empirical Software Engineering (EMSE), 45 pages. In Press.

- **The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models**. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, Kenichi Matsumoto. In Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), pp. 812–823.

- **Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models**. Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. In Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), pp. 789–800.

- **Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt Systems**. Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. In Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015), pp. 168–179.

- **Cross-Project Build Co-change Prediction**. Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E. Hassan. In Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pp. 311–320.

- **Do Code Review Practices Impact Design Quality? An Empirical Study of the Qt, VTK, and ITK Projects**, Rodrigo Morales, Shane McIntosh, and Foutse Khomh. In Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pp. 171–180.

- **An Empirical Study of Delays in the Integration of Addressed Issues**. Daniel Alencar da Costa, Surafel Lemma Abebe, Shane McIntosh, Uirá Kulesza, and Ahmed E. Hassan. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pp. 281–290.
  🏆 *Nominated for best paper* 🏆

- **Tracing Software Build Processes to Uncover License Compliance Inconsistencies**. Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. In Proceedings of the 29th International Conference on Automated Software Engineering (ASE 2014), pp. 731–741.

- **The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects**. Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. In Proceedings of 11th Working Conference on Mining Software Repositories (MSR 2014), pp. 192–201.
  🏆 *Distinguished paper award* 🏆

- **An Empirical Study of Just-In-Time Defect Prediction Using Cross-Project Models**. Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014), pp. 172–181.
  🏆 *Nominated for distinguished paper award* 🏆

- **Magnet or Sticky? An OSS Project-by-Project Typology**. Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014), mining challenge, pp. 353–357.

- **Orchestrating Change: An Artistic Representation of Software Evolution**. Shane McIntosh, Katie Legere, and Ahmed E. Hassan. In Proceedings of the 1st joint meeting of the Conference on Software Maintenance, and Reengineering and the Working Conference on Reverse Engineering (CSMR-WCRE 2014), Early Research Achievements (ERA), pp. 348–352.

# Acknowledgments

Well, here we are! As I near the end of my PhD studies, I suppose that it's natural to reminisce about the important role that others have played in my life to get me here. First, I would like to thank my co-advisors, Bram Adams and Ahmed E. Hassan, who have both played formative roles in my development as a researcher and as a person. They are remarkable people from whom I have learned a great deal.

I have had the privilege of working with an amazing group of young researchers at the Software Analysis and Intelligence Lab (SAIL). Each SAILer has had an impact on me and my work, but I am particularly grateful to Yasutaka Kamei, Meiyappan Nagappan, Weiyi Shang, and Emad Shihab, who have been like older brothers to me, offering advice and sharing their experiences.

I'm grateful for the support of my PhD committee, James R. Cordy and Patrick Martin, who have provided me with insightful feedback and guidance. I also extend thanks to the other members of my examination committee, Gail C. Murphy, Thomas R. Dean, and David A. Lamb, for taking the time to critique my work.

I feel lucky to work with an amazing network of collaborators. Those who co-wrote publications related to the content of this thesis (Audris Mockus, Meiyappan Nagappan, Elmar Juergens, Martin Poehlmann, Brigitte Haupt, and Christian Wagner) shared their thoughts and helped to shape my work. Also, those collaborators on topics outside of the scope of this thesis (Surafel Lemma Abebe, Daniel Alencar da Costa, Julius Davies, Eelco Dolstra, Akinori Ihara, Takafumi Fukushima, Baljinder Ghotra, Hajimu Iida, Daniel M. Germán, Armijn Hemel, Yasutaka Kamei, Foutse Khomh, Uirá Kulesza, Katie Legere, David Lo, Kenichi Matsumoto, Rodrigo Morales, Thanh H. D. Nguyen, Romain Robbes, Weiyi Shang, Emad Shihab, Éric Tanter, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, Naoyasu Ubayashi, Sander van der Burg, Xin Xia, Kazuhiro Yamashita, and Ying Zou) have broadened my research perspective.

Without the support of my family and friends, this thesis would not have been possible. Finally, I am eternally indebted to my wife Victoria, for her devotion, sacrifice, patience, and understanding.

# Statement of Originality

I, Shane McIntosh, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

# Table of Contents

# List of Tables

# List of Figures

Chapter 1

# Introduction

Key Concept

*Although build systems provide critical infrastructure that software organizations require to keep pace with market competitors, they introduce overhead on the software development process.*

Modern software is developed at a breakneck pace. It is not uncommon for large software systems like Mozilla to receive hundreds of change requests (e.g., defect reports, feature requests) on a daily basis [10]. After these change requests have been triaged to the appropriate team members, developers update the codebase to implement the change requests. In August 2014, the codebase of Mozilla was updated 13,090 times — an average of 422 times per day.[1]

In general, such an update would not be released immediately to the end user. In the past, software releases would take several months or even years to prepare, while modern software organizations like Google, LinkedIn, and Facebook release several

[1] http://relengofthenerds.blogspot.ca/2014/09/mozilla-pushes-august-2014.html

times daily [1, 2], grouping the updates since the previous release. Techniques like *Continuous Delivery* (CD) [45] enable software organizations to quickly produce official releases by automatically packaging and deploying software changes that satisfy automated testing criteria.

One of the key components of the software release process is the *build system*, i.e., the specifications that define the complex build process of large software systems. Such a process may involve hundreds of compiler and tool invocations that must be executed in a specific order. The build system interfaces with Integrated Development Environments (IDEs) to provide developers with the means of compiling and testing their code changes in their local environments prior to queuing up their code changes for integration. While modern IDEs can generate build systems for simple applications, complex software systems still require manually specified build systems [86, 99].

An effective build system helps to manage risk in software development by helping developers to detect code compilation and integration problems early in the development cycle. For example, if any of the hundreds of daily code changes cause an error in the build process, team members should be notified immediately so that reactionary measures can be taken. To provide this rapid feedback loop, software organizations adopt techniques like *Continuous Integration* (CI) [32] that routinely download the latest source code changes onto dedicated servers to ensure that they are free of compilation and test failures. This rapid feedback loop provided by CI would not be possible without a robust and efficient build system. Moreover, the rapid release cycle fueled by CD would be error-prone (and thus, too risky) without a reliable build system in place. Indeed, Neville-Neal speculates that the build system is one of the most important development tools [87].

## 1.1 Problem Statement

Although build systems provide critical infrastructure that software organizations require in order to keep pace with market competitors, they introduce overhead on the software development process:

**Thesis Statement:** *The overhead introduced by the build system is an important issue that development teams need to manage. Historical data extracted from software repositories and facts extracted from the build system itself can inform organizational decisions that aim to mitigate this overhead.*

Indeed, the build systems of some software projects introduce more overhead than others, i.e., they require developers and/or testers to spend some of their time updating the build system rather than the code or tests. For example, while 27% of code changes require accompanying build changes in the Mozilla system, only 4% require accompanying build changes in the Jazz system [70]. Hence, in this thesis, we set out to glean actionable information from historical data in software repositories and facts specified in the build system itself order to better understand: (1) the nature of the software development overhead introduced by the build system, and (2) what can be done to mitigate this overhead.

## 1.2 Thesis Overview

We now provide a brief overview of the thesis. Figure 1.1 provides an overview sketch of the scope of this thesis. We first provide the necessary background material (purple boxes):

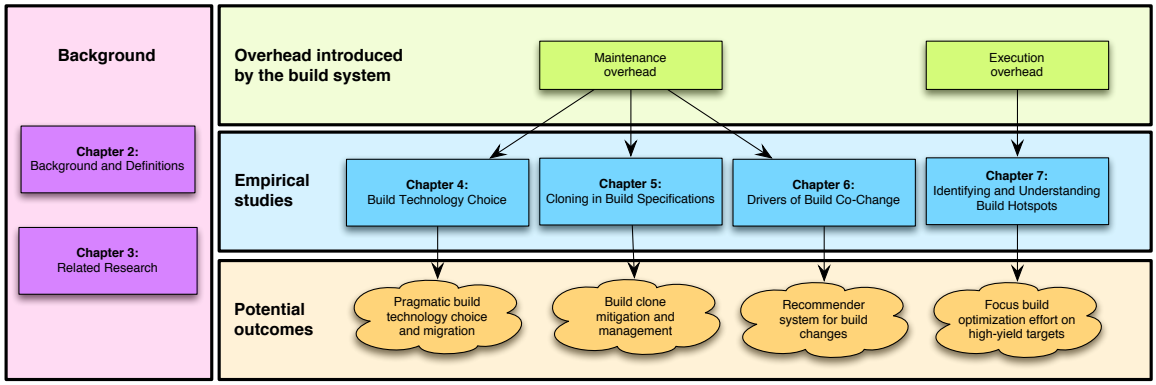



Figure 1.1: An overview of the scope of this thesis.

**Chapter 2:** *Background and Definitions*

Before delving into the software development overhead introduced by build systems, we first provide the reader with background information and define key terms that we will use throughout the thesis.

**Chapter 3:** *Related Research*

In order to situate this thesis with respect to prior research, we present a survey of research on software build systems.

Next, we shift our focus to the main body of the thesis. In this thesis, we focus on two types of overhead introduced by the build system (green boxes). Each type of overhead is further divided into a series of empirical studies (blue boxes) that have compelling potential outcomes (orange clouds). Each empirical study is presented in its own chapter. We introduce the two types of overhead, and the empirical studies below.

### 1.2.1 Maintenance Overhead

Just as source code must be maintained in order to fix defects, add new features, and refactor existing ones, the build system must also be maintained. For example, the

build system must be updated to correctly map the evolving software features to system deliverables and keep up with changing market demands, such as new computing platforms.

*Maintenance overhead* refers to this need to keep the build system synchronized with the other software artifacts (e.g., source code). Kumfert *et al.* argue that there is a "hidden overhead" associated with the maintenance of the build system [58]. Hochstein *et al.* refer to this overhead as the "build tax" [44]. Adams *et al.* [4] and our prior work [66, 67] show that from release to release, source code and build system tend to *co-evolve*, i.e., changes to the source code can induce changes in the build system, and vice versa. Moreover, up to 27% of source code changes and 44% of test code changes are accompanied by changes to the build system [70]. To counteract the overhead introduced by the maintenance of build systems, it is a common practice in industry to dedicate specialized personnel to *build teams* [90], i.e., teams that are entirely focused on maintaining the build system.

In this thesis, we perform three empirical studies that focus on the maintenance overhead introduced by the build system:

**Chapter 4:** *Build Technology Choice*

There are numerous build technologies abound, each with its own nuances. Developers often make build technologies choices for their projects based on anecdotal evidence [100]. In order to help practitioners make more sound, data-driven build technology choices, we analyze historical trends in maintenance activity with respect to ten popular build technologies in a corpus of 177,039 software repositories.

**Chapter 5:** *Cloning in Build Specifications*

Like other software artifacts, build systems are susceptible to *anti-patterns*, i.e., poor solutions to common design and implementation problems. Although there are several documented anti-patterns [55], one of the most prominent ones is *code duplication* (a.k.a., code cloning). Prior studies suggest that excessive code duplication may make maintenance more difficult [48]. Since little is known about how cloning impacts build systems, we collect and analyze a large benchmark of clones in build systems.

**Chapter 6:** *Drivers of Build Co-Change*

The overhead introduced by the maintenance of the build system is exacerbated by the difficulty of identifying the code changes that require accompanying build system changes. If build maintenance is neglected when it is required, execution of the build may be "broken," preventing other team members from performing builds and slowing development progress down. To better understand when build maintenance is required, we train and analyze classifiers that are capable of identifying the source and test code changes that require accompanying build maintenance.

### 1.2.2 Execution Overhead

*Execution overhead* is introduced by the slow nature of using the build system to generate (or regenerate) system deliverables. Since large software systems are made up of thousands of files that contain millions of lines of code, the execution of the build system can be prohibitively expensive, often taking hours, if not days to complete. For example, builds of the Firefox web browser for the Windows operating system take more

than 2.5 hours on dedicated build machines.[2] Certification builds of a large IBM system take more than 24 hours to complete [42]. In a recent survey of 250 C++ developers, more than 60% of respondents report that build speeds are an important issue.[3] Indeed, while developers wait for build tools to execute the set of commands necessary to synchronize source code with deliverables, they are effectively idle.

In this thesis, we perform an empirical study that focuses on the execution overhead introduced by the build system:

**Chapter 7:** *Identifying and Understanding Build Hotspots*

Since some source files may trigger a larger number of commands or slower ones, some source files take longer to rebuild than others. Moreover, some source files are more prone to change than others, and hence, trigger updates more frequently than other source files. Indeed, in prior work, we found that only 10%-25% of the source files of ten large systems like Linux and Mozilla change in a typical month. To help developers to focus build optimization effort on the files that will truly make a difference in day-to-day development, we propose an approach to detect *build hotspots*, i.e., files that not only rebuild slowly, but also tend to change frequently. We evaluate our approach using four large software systems. Moreover, to help developers to avoid creating build hotspots in the first place, we train and analyze classifiers that are capable of explaining the incidence of build hotspots in four large software systems.

[2] http://tbpl.mozilla.org/

[3] http://mathiasdm.com/2014/01/24/a-c-questionnaire-on-build-speed-the-results-are-in/

## 1.3 Thesis Contributions

This thesis demonstrates that:

- Although the more modern build technologies (e.g., Maven) provide additional features that older technologies (e.g., `make`) do not provide, they tend to require additional maintenance activity (Chapter 4). Moreover, the more modern technologies tend to be more prone to cloning than the older technologies (Chapter 5).

- While typical cloning rates in build systems are much higher than those of other software artifacts (e.g., source code), build cloning can be mitigated through the use of indirect reuse patterns that are not directly offered by the build technologies themselves (Chapter 5).

- Properties of code changes can be used to accurately explain when build maintenance is required (Chapter 6).

- Our proposed technique for detecting build hotspots flags files that, if optimized, yield a higher return on investment than focusing on the files that: (1) rebuild the slowest, (2) change the most frequently, or (3) are used the most throughout the codebase (Chapter 7).

- In large projects, build optimization benefits more from architectural refinement than from acting on code properties like file fan-in alone (Chapter 7).

## 1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information and defines key terms. Chapter 3 presents research related to our analysis of the overhead introduced by build systems. Chapters 4 and 5 present the results of our large-scale analyses of build technology choice and build system cloning, respectively. In Chapter 6, we present our study of the drivers of build co-change. Chapter 7 presents the results of our study of build hotspots. Finally, Chapter 8 draws conclusions and discusses promising avenues for future work.

CHAPTER 2

# Background and Definitions

KEY CONCEPT

*The* **build system** *is the set of specifications that describe how development artifacts, such as source code, are transformed into deliverables, such as executables.*

We use the term *build system* to refer to specifications that outline how a software system is assembled from its sources. More specifically, for the purposes of this thesis, the build system is the set of specifications that describe how development artifacts, such as source code, are transformed into deliverables, such as executables.

In the remainder of this chapter, we describe the typical architecture of a build system, the paradigms of build technologies that are used to specify build systems, and the central role that the build system plays in modern software development.

## 2.1 An Overview of the Typical Build Process

Figure 2.1 provides an overview of the typical *build process*, i.e., the steps of assembling a software system that are specified by the build system. The process is typically composed of five steps. During the execution of the build process, if any of the five steps are not successfully completed, the build is "broken."

### 2.1.1 Configuration

The *configuration* step uses environment settings and user preferences to select the set of software features that should be included in the final product from the codebase snapshot that is being built. Moreover, this set of selected software features may influence the set of tools that are required to translate sources into deliverables.

After the configuration step has been executed, a concrete build system capable of producing a concrete set of deliverables has been instantiated.

### 2.1.2 Construction

The *construction* step uses the concrete build system produced by the configuration step to issue the commands (e.g., compilers and linkers) that are required to produce deliverables. These commands are order-dependent, e.g., source code files must be compiled into object code before the object code can be linked together into executables. Hence, specifications of the construction step often describe the relationship between sources, deliverables, and intermediary files using dependencies.

Traditionally, the construction step is conceptually represented using *build targets*.

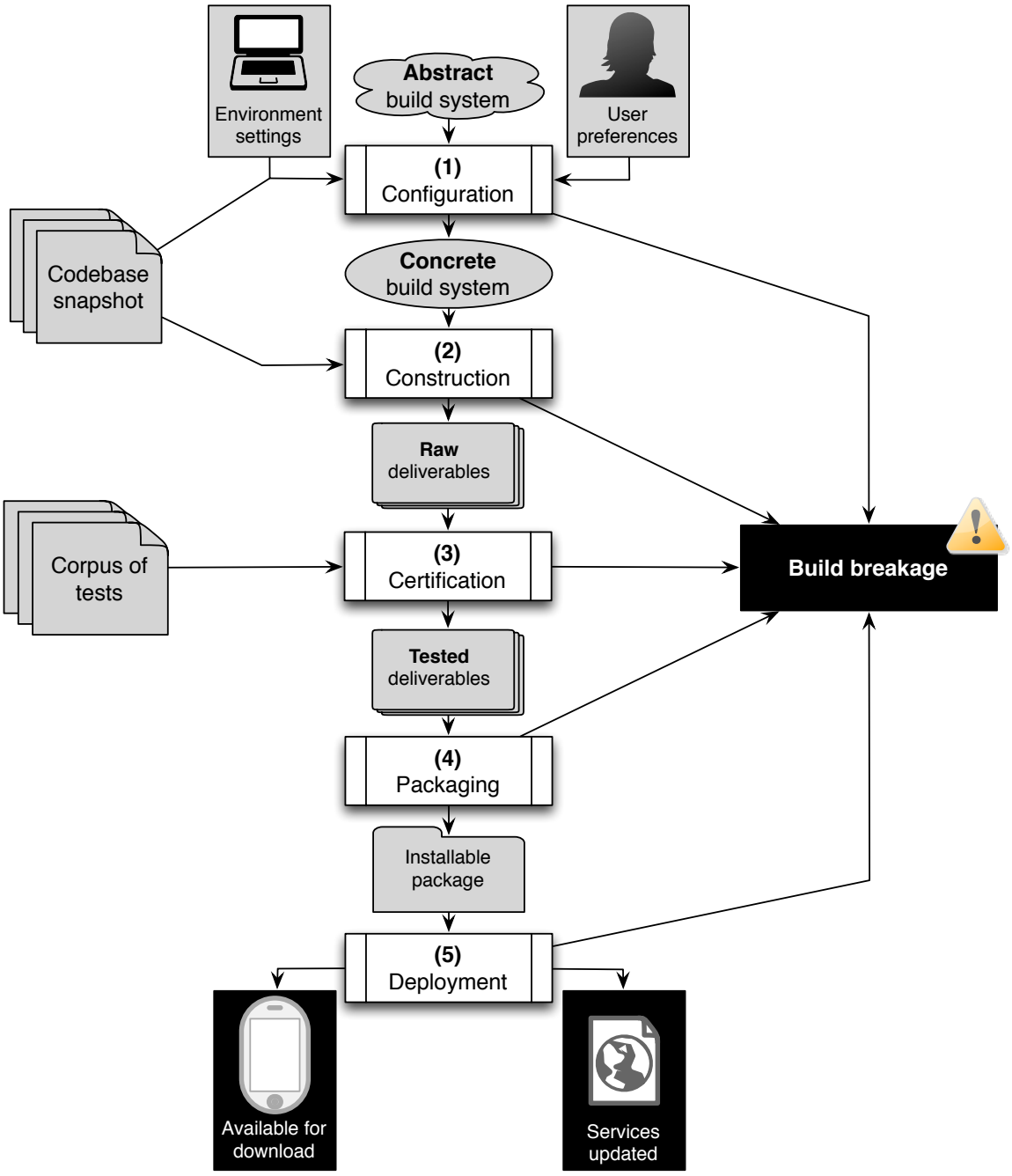



Figure 2.1: An overview of the typical build process.

A build target describes an abstract build goal (or collection of goals) $T$, such as "complete all compilation commands." A target T typically has two key characteristics: (1) a build rule that defines the sequence of commands that must be executed when T is triggered, and (2) a list of dependent targets that determine whether or not T should be triggered. Heuristics are used to speed up a build such that a target is only triggered if its output files do not exist yet or at least one dependent target has been triggered.

After executing the construction step, a set of raw deliverables has been produced.

### 2.1.3 Certification

The *certification* step follows, automatically executing tests to ensure that the raw deliverables produced by the construction step have not introduced regression of system behaviour. It is important to note that while we consider the infrastructure used to automate the execution of the tests as part of the build system, we exclude the automated tests themselves.

After executing the certification step, the raw deliverables produced by the construction step have cleanly passed the suite of automated tests and are ready for packaging.

### 2.1.4 Packaging

The *packaging* step bundles the tested deliverables together with required libraries, documentation, and data files. The gathered materials are collected into a package that can be easily installed directly onto end-user machines, or deployed on organization web infrastructure. After executing the packaging step, the installable package has been created.

### 2.1.5 Deployment

The final step in the build process is the *deployment* step, which either: (1) makes installable packages available for end-users either via organizational means (e.g., a product website) or via a third-party distributor (e.g., software package distributions or so-called "app" stores); or (2) updates the live application code being accessed via the web (e.g., web applications).

## 2.2 Build Technology Paradigms

Build systems are supported by a variety of technologies that subscribe to different design paradigms. In this thesis, we focus on four of the most common build technology paradigms [99]. Furthermore, we study a broad spectrum of technologies that are spread across these build technology paradigms, and as we will show in Chapter 4, are also adopted by several open source repositories. We briefly introduce each of the studied paradigms below. Detailed information about the studied technologies can be found in Appendix A.

### 2.2.1 Low-Level

Low-level technologies explicitly define build dependencies between input and output files, as well as the commands that implement the input-output transformation. For example, one of the earliest build technologies on record is Feldman's `make` tool [35] that automatically synchronizes program sources with deliverables. `Make` specifications outline target-dependency-recipe tuples. Targets specify files created by a recipe, i.e., a shell script that is executed when the target either: (1) does not exist, or (2) is older

than one or more of its dependencies, i.e., a list of other files and targets. Targets may also be *phony*, representing abstract phases of a build process rather than concrete files in a filesystem. Ant borrows the tuple concept from `make`, however all Ant targets are abstract. When an Ant target is triggered, a list of specified tasks are invoked that each execute Java code rather than shell script recipes to synchronize sources with deliverables. Similarly, Rake, Jam, and SCons also follow the `make` tuple paradigm, but allow build maintainers to write specifications in portable scripting languages: Ruby, Perl, and Python respectively. We study the `make`, Ant, Rake, Jam, and SCons low-level technologies.

### 2.2.2 Abstraction-Based

Platform-specific nuances forced maintainers of portable applications, but using low-level build technologies, to repeat several "boilerplate" low-level build expressions for handling variability in platform implementation over and over again (e.g., different compilers, library support). Abstraction-based tools attempt to address this flaw by automatically generating low-level specifications based on higher level abstractions. For example, GNU Autotools specifications describe external dependencies, configurable compile-time features, and platform requirements. These specifications can be parsed to generate `make` specifications that satisfy the described constraints. Similarly, CMake abstractions can be used to generate `make` specifications, as well as Microsoft Visual Studio and Apple Xcode project files. We study the Autotools and CMake abstraction-based technologies.

### 2.2.3 Framework-Driven

Framework-driven technologies favour build convention over configuration. For example, the Maven technology assumes that source and test files are placed in default locations and that projects adhere to a typical Java dependency policy (e.g., `.class` files are generated by compiling `.java` files of the same name), unless otherwise specified. If projects abide by the conventions, Maven can infer build behaviour automatically, without any explicit specification. We study the Maven framework-driven technology.

### 2.2.4 Dependency Management

Dependency management tools augment the three types of build systems above by automatically managing external API dependencies. Developers specify the names and exact or minimum version numbers of API dependencies. The dependency management tool ensures that a local cache contains the APIs necessary to build the project, downloading missing ones from an upstream repository server when necessary. Dependency management tools offer two advantages: (1) users no longer need to carefully install the required versions of libraries manually, and (2) production and development environments can coexist, since the potentially unstable versions of libraries that are required for development are placed in a local cache that is quarantined from the running system. We study the Ivy and Bundler dependency management technologies.

## 2.3 The Central Role of the Build System

The build system plays an important role in modern software development. Figure 2.2 shows the various practitioners, release automation, and development tools that need

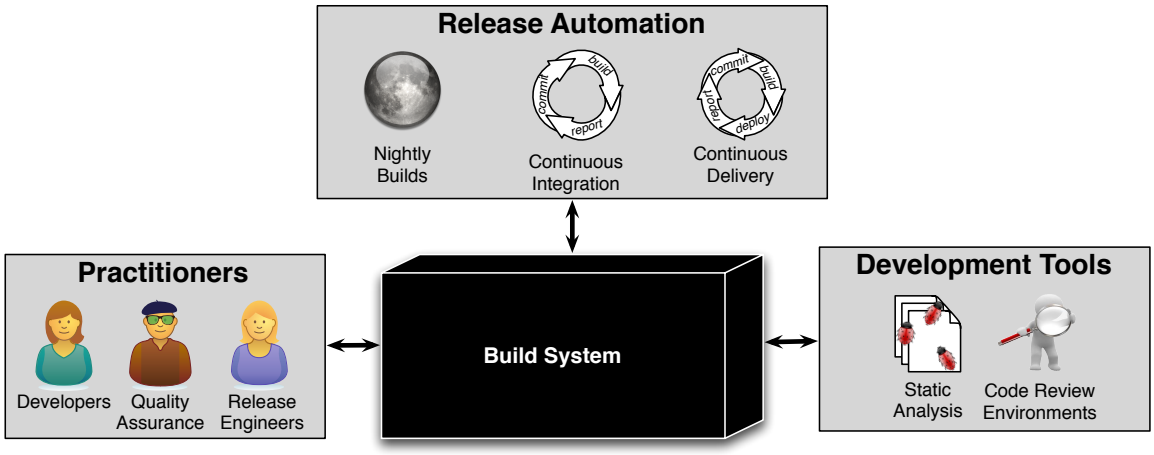



Figure 2.2: The interactions between the build system and various practitioners, release automation, and development tools.

to interact with the build system regularly. We describe each of these build system interactions below.

### 2.3.1 Practitioners

Many types of practitioners rely on the build system in order to accomplish their jobs in a timely manner. For example, after modifying the source code, *developers* rely on the build system in order to integrate their code changes into system deliverables. *Quality Assurance* (QA) personnel integrate automated tests into the build process to help development teams to avoid introducing regression of system behaviour. *Release engineers* maintain the build system to ensure that the process of producing official releases is sound and repeatable.

### 2.3.2 Release Automation

Various forms of release automation are also reliant on the build system. For example, nightly builds are performed to integrate the set of changes that occurred during the day into a testable version of the software system that QA personnel can test on the following day. Moreover, techniques like *Continuous Integration* (CI) automatically perform builds when a new change is recorded in the project's Version Control System (VCS) in order to detect compilation errors and test failures on a more frequent basis than nightly builds do. Recently, companies such as Google have adopted *Continuous Delivery* (CD) [45], a development approach that facilitates rapid distribution of newly produced versions of a software system by extending CI builds, which typically terminate after the certification step, to include packaging and deployment steps.

### 2.3.3 Development Tools

Finally, various tools that have become critical parts of the development process also depend on the build system. For example, *static analysis* tools (e.g., Coverity Code Advisor[1]) construct system-level data structures, such as abstract syntax trees, by instrumenting the build system. Furthermore, *code reviewing environments* (e.g., Gerrit[2]) provide an interface for automation tools to scan changes that have been posted for review. Software teams such as Qt, VTK, and ITK connect the build process with the code review environment in order to detect build and test errors automatically before human reviewers invest (expensive) manual effort [71].

[1] http://www.coverity.com/products/code-advisor/
[2] https://code.google.com/p/gerrit/

## 2.4 Chapter Summary

This chapter defines foundational concepts and provides motivation for the general study of build systems. Specifically, we define the build system, describe its typical structure, the technologies that support modern build systems, and describe the critical role that build systems play in modern software development.

In the next chapter, we survey prior research on build systems in order to situate our empirical studies of the software development overhead introduced by build systems within the broader scope of the body of knowledge.

CHAPTER 3

# Related Research

KEY CONCEPT

*The critical role that build systems play in the modern software development process has inspired a wealth of recent studies of build systems.*

In this chapter, we survey the related research on build systems. We organize the work along the maintenance and execution themes of software development overhead that are the focus of this thesis. More specifically, we describe how the related work motivates our four empirical studies.

## 3.1 Maintenance Overhead

The critical role that build systems play in the modern software development process has inspired recent studies of the reliability of build systems. For example, by mining the issue reports of the Ant, Maven, CMake, and QMake build tools, Xia *et al.* find that the most defect-prone component of a build tool is the external interface with which

users interact [109, 110]. Nadi *et al.* develop techniques for reporting anomalies between the source and build system as likely defects in Linux [83, 84, 85]. Al-Kofahi *et al.* propose a fault localization technique for `make`-based build systems [6, 7].

To help practitioners to cope with the overhead of maintaining the build system, recent research has proposed several tools. Adams *et al.* develop the MAKAO tool to visualize and reason about build dependencies [3]. Tamrawi *et al.* propose a technique for visualizing and verifying build dependencies using symbolic dependency graphs [101, 102]. Al-Kofahi *et al.* extract the semantics of build system changes using MkDiff [8]. Buffenbarger proposes a variant of GNU `make` called `amake` [18] that uses hash signatures of files to detect when files need to be rebuilt instead of using the last modification timestamp in order to avoid inconsistent builds caused by unsynchronized machine clocks in a multi-machine environment. Hardt and Munsen propose Formiga — a tool that assists in performing common Ant maintenance tasks [41].

Little, however, is known about the factors that drive build maintenance. Although modern Integrated Development Environments (IDEs) provide support for building simple applications, complex software systems still require manually maintained build systems [86, 99]. Indeed, Martin *et al.* find that hand-written build systems tend to use more advanced features of the GNU `make` build technology [64]. In this section, we motivate our three empirical studies of what drives build maintenance, and what can be done to mitigate it.

## Empirical Study 1: Build Technology Choice

There are dozens of build technologies available for developers to select from,[1] each with its own nuances. These technologies adopt various design paradigms. As shown in Chapter 2, four of the most common design paradigms are [99]:

**Low-level technologies** (e.g., `make` [35]) require explicitly-defined build dependencies between each input and output file.

**Abstraction-based technologies** (e.g., CMake[2]) use project metadata, such as the list of files to build, to generate low-level build systems.

**Framework-driven technologies** (e.g., Maven[3]) eliminate the "boilerplate" dependency expressions that are typical of low-level technologies in favour of conventions, e.g., expecting input and output files to appear in default locations.

**Dependency management technologies** (e.g., Ivy[4]) are used to automatically manage external API dependencies.

When the costs associated with build maintenance grow unwieldy, software teams like KDE,[5] MySQL,[6] and Hibernate[7] have migrated between technologies, reimplementing thousands of lines of build logic using a (perceived to be superior) technology [100]. Zadok reports that the size and complexity of the Berkeley Automounter build system was reduced by migrating from `make` to GNU Autotools [113].

[1] http://en.wikipedia.org/wiki/List_of_build_automation_software
[2] http://www.cmake.org/
[3] http://maven.apache.org/
[4] http://ant.apache.org/ivy/
[5] http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4
[6] http://www.lenzg.net/archives/291-Building-MySQL-Server-with-CMake-on-LinuxUnix.html
[7] http://lists.jboss.org/pipermail/hibernate-dev/2007-May/002075.html

Thus far, build system studies have focused on a small sample of between one and ten projects. In such small samples, confounding factors like build technology choice can only be modestly controlled, with most of the studies being performed on `make`, Ant, and Maven build systems. Hence, it is not clear whether a large investment in migration to a different technology can truly impact build maintenance overhead.

We, therefore, set out to investigate the role that build technology choice plays with respect to build maintenance. In order to ensure that our conclusions are valid and repeatably observed, we mine the version history of a large corpus of open source VCS repositories that was collected by Mockus [77].

## Empirical Study 2: Cloning in Build Specifications

The design of software build systems is critical. De Jonge [25, 26] and Elsner *et al.* [33] argue that the reuse of software components is limited by the design of its build system. Tu and Godfrey found examples of complex build system designs in the GCC and Perl systems [104]. They identify the *code robot* design pattern, where the build system produces a partial version of the system that lacks some features (e.g., the GCC compiler) in order to build the remainder of the system.

Complex software architectures like Software Product Lines (SPLs) [22] are often implemented through carefully designed build systems. For example, Unphon shows that investment in a well-designed *build hierarchy*, i.e., organization of software components and dependencies, led to improved quality in an industrial SPL [105]. Berger *et al.* find that the build systems of the Linux and eCos systems, which describe and constrain how various system features may be combined, are realizations of several theoretical variability modelling concepts [14]. Nadi *et al.* argue that the variability model

of Linux can be divided into high-level feature constraints described using configuration tools like Linux KConfig (i.e., the problem space), and low-level details described in source code and construction layer specifications (i.e., the solution space) [81].

Indeed, it is crucial that the build system is analyzed when studying SPLs in practice. For example, Dietrich *et al.* extract feature-to-code mappings from the Linux build system [28]. Passos *et al.* catalog several patterns of co-evolution between the variability model of the Linux kernel and its other artifacts, finding that many of these patterns trigger changes in the build system [89].

On the other hand, poorly-designed build systems increase the difficulty of build maintenance. For example, Miller shows that the commonly-adopted *recursive-make* build system design, where build modules are implemented in independent build specification files, can produce indeterminate build results [74]. Adams *et al.* found that due to maintenance difficulties, the initial build system for the Linux kernel needed to be redesigned during the 2.5 release [4]. Suvorov *et al.* report that an initial build system migration attempt was abandoned by the KDE team due to insufficient solicitation of requirements [100].

One of the most common anti-patterns in software systems is cloning [57, 94], i.e., duplication of software logic. There is a lack of consensus about the harmfulness of cloning in general. For example, Kasper and Godfrey show that there are positive ways that cloning is used in open source systems [51], and Rahman *et al.* show that this link between defect proneness and cloning is tenuous at best [92]. Nonetheless, Juergens *et al.* find that there are open source and proprietary systems where clones are unintentionally made inconsistent with one another, which introduces defects that are often difficult to diagnose [48].

Kasper and Godfrey also show that cloning is often a reactionary measure when programming in languages that lack the mechanisms for properly abstracting a concept [51]. Indeed, the *recursive-make* build system design may have stemmed from this lack of abstraction mechanisms provided by early versions of the `make` technology. Since build technologies share other similarities with programming languages, we suspect that cloning may also impact build systems as well. We, therefore, set out to empirically study cloning in build systems.

## Empirical Study 3: Drivers of Build Co-Change

Broken builds can slow development progress and the release process down. Seo *et al.* find that 30%-37% of builds triggered by Google developers on their local copies of the source code are broken [95]. If those local build breakages are not fixed before the changes are committed to upstream repositories, then the software team as a whole will be negatively impacted. Neitsch *et al.* find that several Ubuntu 9.10 source code packages implemented using multiple programming languages do not build cleanly due to subtle differences in build environments [86]. Kwan *et al.* find that 31% (60/191) of the studied IBM team builds were broken [59]. Furthermore, Hassan and Zhang find that 15% (209/1,429) of the studied IBM certification builds (i.e., builds that the development team believed were ready for testing) were broken [42]. Kerzazi *et al.* estimate that between 893-2,133 man-hours are wasted due to a build breakage rate of 19% in a large industrial system [52]. Downs *et al.* show that ambient devices can help to raise awareness of build breakage in a non-intrusive manner [30]. To reduce this build downtime, Van der Storm proposes an algorithm that automatically backtracks changes to a VCS branch if the changes cause build breakage [107].

Recent studies have found that neglected build maintenance is a commonly detected root cause of broken builds [95]. Indeed, neglecting to propagate changes to the build system when it is necessary can generate lingering inconsistencies in the build process. Morgenthaler *et al.* show that neglected build maintenance at Google has generated *build debt* [80], i.e., a form of technical debt [23] that accumulates in the build system due to neglected build change propagation. Nadi *et al.* find that Linux kernel variability anomalies, i.e., inconsistencies between source code and build system are rarely caused by trivial, typo-related issues, but they are more often caused by incomplete changes, e.g., changes to source code that are not entirely propagated to the build system [82]. Furthermore, these variability anomalies tend to linger for as many as six Linux releases before they are fixed.

Yet, despite the importance of performing source-build co-changes when they are necessary, the driving factors of this co-change relationship are not well understood. While recent work by Shridhar *et al.* study the frequency and invasiveness of different types of build changes [98], little is known about the characteristics of changes to other project artifacts like source and test code that would require accompanying build changes. If source and test code co-change with the build system frequently, those source and test changes may contain information about what would likely trigger changes to the build system. Hence, we set out to empirically study the characteristics of code changes that trigger accompanying build changes.

## 3.2 Execution Overhead

Large software systems can require more than 24 hours to completely rebuild [42]. Developers need to execute several builds on a daily basis. For example, Seo *et al.* show

that each developer at Google executes an average of 6-10 builds daily [95]. To avoid incurring large build performance penalty for each build, build tools such as `make` [35] provide *incremental builds*, i.e., builds that calculate and execute the minimal set of commands necessary to synchronize updates to the source code with deliverables. Humble and Farley suggest that incrementally building and testing a change to the source code should take no more than a few minutes [45]. Developers have even scrutinized 5-minute long incremental build processes,[8] calling the process "abysmally slow."[9] Again, the slower the incremental build process, the longer the idle period, frustrating developers and slowing down development progress.

Like build maintenance, slow build performance is another form of software development overhead introduced by the build system. In this section, we motivate an empirical study that aims to identify and understand *build hotspots*, i.e., source code files that not only rebuild slowly, but also change often.

## Empirical Study 4: Identifying and Understanding Build Hotspots

Prior work has explored how slow build processes can be accelerated. Adams *et al.* achieve up to an 80% improvement in build performance through intelligent recompilation algorithms and elimination of unused environment symbols [5]. Yu *et al.* improve build speed by automatically removing unnecessary dependencies between files [112] and redundant code from C header files [111]. Dayani-Fard *et al.* propose semi-automatic architectural refactorings that improve build performance [24]. Telea and Voinea propose Build Analyzer [103] — a tool that mines build dependencies to identify bottlenecks in the build process.

[8] https://bugs.webkit.org/show_bug.cgi?id=32921
[9] https://bugs.webkit.org/show_bug.cgi?id=33556

While these studies propose approaches that can holistically improve build performance, it is not clear if they truly target the files that slow typical builds down the most. Since in prior work, we found that only 10%-25% of the source files of ten large systems like Linux and Mozilla change in a typical month [70], this suggests that traditional build profiling techniques may miss the files that would really make a difference in day-to-day development. Instead, we believe that build optimization effort should be focused on *build hotspots*, i.e., files that not only take a substantial amount of time to rebuild, but also require frequent maintenance, and thus generate considerable overhead on incremental builds.

Since rebuild cost, rate of change, and impact on other files can also be used to prioritize files for build optimization, we want to evaluate whether build hotspots are truly the most costly files. We comparatively study build hotspots with respect to other prioritization schemes.

Furthermore, since code changes are required to address defects or add new features, one cannot simply avoid changing the code. Instead, build optimization effort must focus on controllable properties that influence the likelihood of a file becoming a build hotspot. Hence, we set out to study controllable file properties that have an influence on the likelihood of a file becoming a build hotspot.

## 3.3 Chapter Summary

In this chapter, we survey prior research along the build maintenance and execution overhead themes that are central to this thesis. We find that while the related work supports our hypothesis that build systems introduce overhead on the software development process, it is not yet clear: (1) what factors drive the maintenance of build

systems, and (2) where optimization effort should be invested in order to reduce the build overhead and streamline the development and release processes.

Broadly speaking, the remainder of this thesis describes our empirical studies that set out to tackle these two gaps in the literature. We begin, in the next chapter, by studying the impact that technology choice can have on build maintenance activity.

Chapter 4

# Build Technology Choice

Central Question

? *Is there a relationship between build technology choice and build maintenance activity?*

**An earlier version of the work in this chapter appears in** the Springer Journal of Empirical Software Engineering (EMSE) [72]

## 4.1 Introduction

Prior research on build systems has shown that: (1) they require non-trivial maintenance effort [70] in order to stay in sync with the source code that it builds [4, 67], and

(2) when the maintenance effort associated with the build system grows unwieldy, development teams opt to migrate to a different (perceived to be superior) build technology [100]. Furthermore, anecdotal evidence[1,2] indicates that developers who need to make modifications to the build system are rarely fluent with them, making it hard for them to keep up with the demanding requirements of the build system.

Thus far, build system studies have focused on a small sample of between one and ten projects. In such a small sample, confounding factors like build technology choice can only be modestly controlled, with most of the studies being performed on `make`, Ant, and Maven build systems. Hence, it is not clear what role technology choice plays in *build maintenance*, i.e., the amount of activity required to keep the build system in sync with the source code.

We, therefore, set out to empirically study how widely a sample of popular build technologies are adopted, and their relationship with build maintenance activity. We set out to address the following question:

**Central Question:** *Is there a relationship between build technology choice and build maintenance activity?*

In order to ensure that our conclusions are valid and repeatably observed, we mine version history in a corpus of 177,039 open source code repositories. We record our observations with respect to five research questions and three themes:

## Theme 1: Build Technology Adoption

Adoption trends can provide insight into the build technologies that development communities are using in practice. Much research focuses on the `make`, Ant, and Maven

[1] http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4
[2] http://argouml.tigris.org/ds/viewMessage.do?dsForumId=450&dsMessageId=2618367

build systems. However, little is known about how broadly these technologies are adopted in practice, nor which other technologies require attention from researchers and service providers. In order to bridge this gap, we formulate the following two research questions:

(RQ1) *Which build technologies are broadly adopted?*

**Motivation:** It is unknown how widespread each technology is. Understanding the market share associated with each technology would help: (1) projects decide which technology to use, (2) researchers to select which technologies to study, and (3) individuals and companies who provide products and services that depend on or are related to build technologies to tailor their solutions to fit the needs of target users.

**Results:** We find that while traditional build technologies like `make` are frequently adopted, a growing number of projects use newer technologies like CMake.

(RQ2) *Is choice of build technology impacted by project characteristics?*

**Motivation:** The flexibility of build technologies enables use cases beyond those for which they were initially designed. For example, the `make` technology was not intended for use in large systems [35], nor for use in the recursive paradigm that is frequently adopted [74]. Hence, it is unclear whether project characteristics like system size or programming language influence build technology adoption. Understanding whether these factors are related to build technology use may help in the design of better build tools and help build service providers select more effective solutions.

**Results:** Programming language choice influences build technology choice — language-specific build technologies that are more attuned to the compile-time

and packaging needs of a programming language are more frequently adopted than language-agnostic ones that are not.

## Theme 2: Build Maintenance

Although the more modern build technologies offer powerful abstraction techniques, it is not clear whether they actually ease the burden of build maintenance. The advantages of a more rapid build cycle enabled by a more powerful build technology may be outweighed by the complexity of build maintenance associated with it. Therefore, we set out to examine the following two research questions:

(RQ3) *Does build technology choice correlate with build change activity?*

**Motivation:** Build systems require maintenance to remain functional and efficient as source files, features, and supported platforms are added and removed. Reducing the amount of build maintenance is of concern for practitioners who often refer to build maintenance as a "tax" on software development [44]. We are interested in studying whether build technology choice can have an influence on the build "tax."

**Results:** Surprisingly, the modern, framework-driven and dependency management technologies tend to induce more maintenance activity than low-level and abstraction-based specifications. Indeed, for systems implemented using Java and Ruby, a large portion of build specification churn is spent on external dependency management.

(RQ4) *Does build technology choice correlate with the overhead on source code development?*

**Motivation:** Developers rely on the build system to test their incremental source code changes. Our prior work shows that source code changes frequently require accompanying build changes [70]. We are interested in studying whether the development overhead of build maintenance is influenced by technology choice.

**Results:** Complementary to the results of RQ3, we find that the modern, framework-driven and dependency management technologies tend to be more tightly coupled to source code than low-level and abstraction-based specifications.

## Theme 3: Build Technology Migration

Any build system requires maintenance, which can quickly become unwieldy.[3] Software teams take on build migration projects to counteract this, where build specifications are reimplemented, often using different (perceived to be superior) build technologies (e.g., MySQL[4] and KDE[5]). These build migration projects require a large investment of team resources, both in terms of time and effort. Even then, Suvorov *et al.* find that migration projects can fail due to a lack of build system requirements [100]. Indeed, build maintainers often select build technologies based on "gut feel." For example, the first KDE build migration attempt failed partly because the build technology was hastily selected by taking a vote at a developer conference [100]. To assess the impact of build technology migration on build maintenance, we formulate the following research question:

[3] http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4

[4] http://www.lenzg.net/archives/291-Building-MySQL-Server-with-CMake-on-LinuxUnix.html

[5] http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4

(RQ5) *Does build technology migration reduce the amount of build maintenance?*

**Motivation:** Migration from one technology to another is often perceived as a reasonable solution, however there is little quantitative evidence to indicate whether these migrations are "worth it," i.e., whether they really increase or reduce build maintenance activity.

**Results:** Most technology migrations successfully reduce the impact of build maintenance on developers. Migrations are often accompanied with a shift to a specialized build maintenance team, reducing the build "tax" that other developers must pay.

**Chapter organization.** The remainder of this chapter is structured as follows. Section 4.2 describes the design of our empirical study, while Sections 4.3 to 4.5 discuss the results with respect to our five research questions. Section 4.6 discloses the threats to the validity of our empirical study. Finally, Section 4.7 draws conclusions.

## 4.2 Empirical Study Design

Figure 4.1 presents an overview of the approach that we took to address our research questions. This design is based on the four steps suggested by Mockus for analyzing software repositories [76]. We describe each of the four steps below.

### 4.2.1 Retrieve Raw Data

It is important that we study a large sample of software projects in order to improve confidence in the conclusions that we draw. However, investigating a large number of software projects leads to much diversity in terms of development processes and

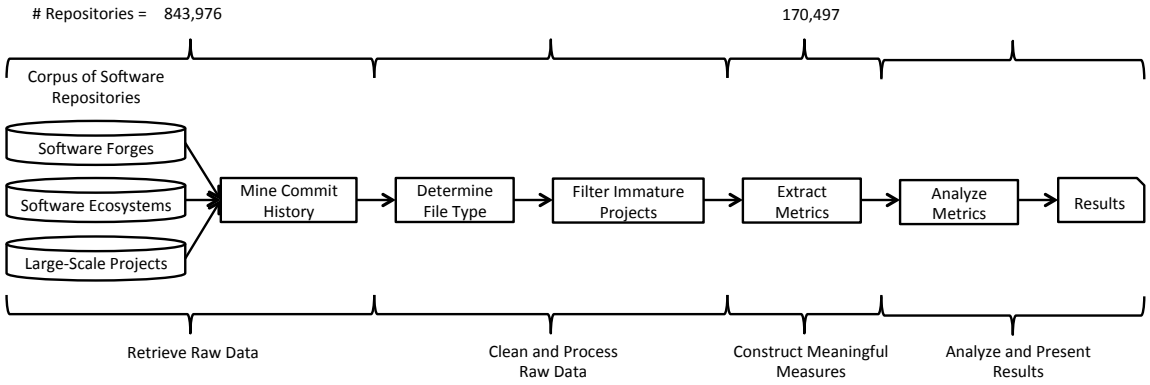



Figure 4.1: [Empirical Study 1] Overview of our approach to the impact that technology choice has on build maintenance activity.

practices. In order to control for this, it is important to stratify the sample accordingly. Stratification of the sample has two benefits: (1) research questions can be addressed for each relevant subsample, and (2) the reliability of the findings improves if the same or similar behaviour is observed among subsamples. Hence, we extract, stratify, and mine a large corpus of open source version history collected by Mockus [77]. We describe the corpus of repositories used in this study and explain our extraction, stratification, and mining approaches below.

#### 4.2.1.1 Corpus of Software Repositories

Table 4.1 provides an overview of the corpus of studied repositories of varying size and purpose. The data in the corpus has been meticulously collected from numerous public Version Control Systems (VCSs) over the past 10 years [77]. The corpus contains over 1.3 terabytes of textual data describing source code, build system, and other development artifact changes that occurred in the VCS commit logs of various open source

software projects. We first stratify the sample by:

**Software forge:** A service provider that hosts repositories for development teams. Since forge repositories are contributed by a plethora of unrelated development teams, they are rarely reliant on one another. We analyze repositories from the Github, repo.or.cz, RubyForge, and Gitorious forges.

**Software ecosystem:** A collection of software that is developed using the same process, often by a large team. Repositories are loosely reliant on one another. We analyze repositories from the Apache, Debian, and GNU ecosystems.

**Large-scale project:** A software project that records changes to each subsystem using separate repositories. Repositories are heavily reliant on one another. We analyze the Android, GNOME, KDE, and PostgreSQL large-scale projects.

The majority of the repositories that we study are from the Github forge. The reason for this is twofold. First, Github is a very popular software forge, perhaps the largest of its kind, with millions of developers relying on it daily. This inflates the number of repositories that originate there. Second, to ensure that our authorship analyses are valid, we require that the original author of each code change is carefully recorded, which the underlying Git VCS allows developers to do. In addition, sets of file changes that authors submit together need to be recorded atomically with a single revision identifier (i.e., atomic commits). To that end, we narrow our scope of study to repositories using a VCS that records these details, which artificially reduces the size of some ecosystems that support several VCS tools (e.g., Debian).

Table 4.1: [Empirical Study 1] Overview of the studied repositories. The most frequently used build technologies and programming languages in the filtered set of repositories are shown in boldface. Percentages will not add up to 100%, since multiple technologies can be used by a single repository.

| | | | **Forges** | | | | **Ecosystems** | | | **Projects** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Github | Gitorious | repo.or.cz | RubyForge | Apache | Debian | GNU | Android | GNOME | KDE | PostgreSQL |
| # Repositories | | | 832,379 | 2,693 | 1,823 | 539 | 179 | 3,799 | 412 | 239 | 991 | 858 | 64 |
| # After filtering | | | 169,033 | 645 | 602 | 217 | 179 | 3,799 | 412 | 239 | 991 | 858 | 64 |
| **Build Technology** | **Low-Level** | Ant | 27,014 | 61 | 51 | 4 | 112 | 116 | 7 | 18 | 5 | 27 | 22 |
| | | | 16% | 9% | 8% | 2% | 63% | 3% | 2% | 8% | 1% | 3% | 34% |
| | | Jam | 851 | 14 | 15 | 0 | 0 | 22 | 2 | 3 | 0 | 2 | 0 |
| | | | 1% | 2% | 2% | 0% | 0% | 1% | <1% | 1% | 0% | 1% | 0% |
| | | Make | 62,107 | **381** | **395** | 24 | 41 | 1,890 | 225 | **227** | 348 | 182 | **42** |
| | | | 37% | **59%** | **66%** | 11% | 23% | 50% | 55% | **95%** | 35% | 21% | **66%** |
| | | Rake | **75,718** | 129 | 43 | **210** | 10 | 21 | 3 | 0 | 0 | 12 | 1 |
| | | | **45%** | 20% | 7% | **97%** | 6% | 1% | 1% | 0% | 0% | 1% | 2% |
| | | SCons | 3,012 | 23 | 26 | 0 | 1 | 52 | 5 | 4 | 4 | 56 | 0 |
| | | | 2% | 4% | 4% | 0% | 1% | 1% | 1% | 2% | <1% | 1% | 0% |
| | **Abstr** | Autotools | 34,318 | 292 | 347 | 6 | 35 | **2,210** | **263** | 65 | **854** | 388 | 37 |
| | | | 20% | 45% | 58% | 3% | 20% | **58%** | **64%** | 27% | **86%** | 45% | 58% |
| | | CMake | 7,920 | 74 | 66 | 0 | 2 | 138 | 18 | 2 | 3 | **705** | 4 |
| | | | 5% | 11% | 11% | 0% | 1% | 4% | 4% | 1% | <1% | **82%** | 6% |
| | **FW** | Maven | 17,958 | 9 | 19 | 3 | **135** | 81 | 2 | 10 | 0 | 0 | 0 |
| | | | 11% | 1% | 3% | 1% | **75%** | 2% | <1% | 4% | 0% | 0% | 0% |
| | **Dep** | Ivy | 2,341 | 0 | 1 | 0 | 19 | 8 | 0 | 0 | 0 | 0 | 0 |
| | | | 1% | 0% | <1% | 0% | 11% | <1% | 0% | 0% | 0% | 0% | 0% |
| | | Bundler | 37,394 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | | | 22% | 0% | 0% | 1% | 0% | 0% | 0% | 0% | 0% | <1% | 2% |
| **Programming Language** | | Ruby | **70,680** | 126 | 44 | **217** | 6 | 66 | 11 | 2 | 10 | 39 | 1 |
| | | | **42%** | 20% | 7% | **100%** | 3% | 2% | 3% | 1% | 1% | 5% | 2% |
| | | Javascript | 33,307 | 25 | 16 | 9 | 17 | 173 | 10 | 7 | 28 | 48 | 3 |
| | | | 20% | 4% | 3% | 4% | 9% | 5% | 2% | 3% | 3% | 6% | 5% |
| | | Java | 25,436 | 80 | 45 | 3 | **134** | 169 | 8 | 84 | 14 | 10 | 7 |
| | | | 15% | 12% | 7% | 1% | **75%** | 4% | 2% | 35% | 1% | 1% | 11% |
| | | Python | 19,280 | 60 | 62 | 0 | 6 | 428 | 46 | 17 | 141 | 66 | 10 |
| | | | 11% | 9% | 10% | 0% | 3% | 11% | 11% | 7% | 14% | 8% | 16% |
| | | C++ | 17,582 | **349** | **380** | 10 | 13 | 525 | **198** | 54 | 87 | **603** | **31** |
| | | | 10% | **54%** | **63%** | 5% | 7% | 14% | **48%** | 23% | 9% | **70%** | **48%** |
| | | C | 16,918 | 225 | 280 | 17 | 12 | **1,363** | 178 | **93** | **523** | 44 | **31** |
| | | | 10% | 35% | 47% | 8% | 7% | **36%** | 43% | **39%** | **53%** | 5% | **48%** |
| | | Objective-C | 15,905 | 0 | 1 | 0 | 15 | 877 | 1 | 56 | 488 | 4 | 0 |
| | | | 9% | 0% | <1% | 0% | 8% | 23% | <1% | 23% | 49% | <1% | 0% |
| | | PHP | 7,198 | 28 | 23 | 0 | 4 | 110 | 19 | 4 | 18 | 30 | 3 |
| | | | 4% | 4% | 4% | 0% | 2% | 3% | 5% | 2% | 2% | 3% | 5% |
| | | Shell | 3,253 | 17 | 24 | 3 | 16 | 863 | 86 | 39 | 232 | 101 | 14 |
| | | | 2% | 3% | 4% | 1% | 9% | 23% | 21% | 16% | 23% | 12% | 22% |
| | | Perl | 857 | 3 | 5 | 0 | 5 | 420 | 13 | 11 | 95 | 7 | 10 |
| | | | 1% | <1% | 1% | 0% | 3% | 11% | 3% | 5% | 10% | 1% | 16% |

#### 4.2.1.2 Mine Commit History

Our corpus contains 843,976 distinct repositories. Each repository contains a set of atomic commits describing the change history of various source code, build system, and other development artifacts. Each atomic commit includes a unique identifier, the author name, a listing of file changes, and the time when the changes were submitted.

### 4.2.2 Clean and Process Raw Data

We process the raw commit data to identify the source and build files in each repository. Once we have preprocessed the data, we need to filter out immature or inactive software projects because they may not require a build system.

#### 4.2.2.1 Determine File Type

We mark each commit as changing either source, build, both, or neither. In our prior work [70], we categorized source and build files semi-automatically, however with a corpus of this scale, manual categorization is infeasible. To address this, we conservatively categorize source and build files based on filename conventions with an extended version of the Github Linguist tool.[6] We have made our extended version available online.[7] An overview of the filename conventions that we map to each technology is given in Table 4.2.

[6] https://github.com/github/linguist/
[7] http://sailhome.cs.queensu.ca/replication/shane/PhD/

Table 4.2: [Empirical Study 1] The adopted file name conventions for each build technology.

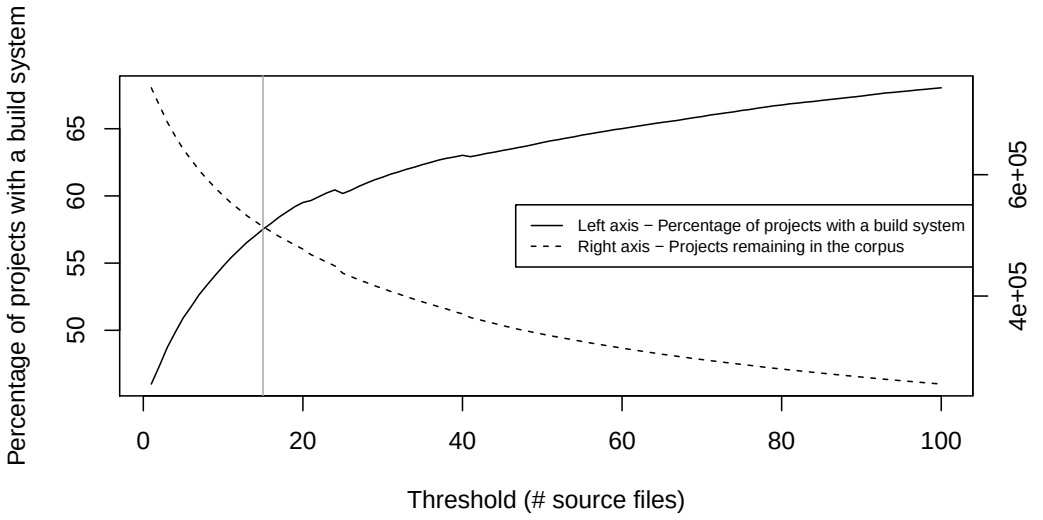
| Paradigm | Technology | Conventions |
|---|---|---|
| Low-Level | Ant | `build.xml`, `build.properties` |
| | Jam | `[Jj]amfile`, `*.jam` |
| | Make | `(GNU)?[Mm]akefile`, `*.mk`, `*.mak`, `*.make` |
| | Rake | `[Rr]akefile`, `*.rake`, |
| | SCons | `SConstruct`, `SConscript`, `*.scons` |
| Abstraction-Based | Autotools | `[Cc]onfigure.(ac\|in)`, `ac(local\|site).m4`, `[Mm]akefile.(am\|in)`, `config.h.in` |
| | CMake | `CMakeLists.txt`, `*.cmake` |
| Framework-Driven | Maven | `pom.xml`, `maven([123])?.xml` |
| Dependency Management | Ivy | `ivy.xml` |
| | Bundler | `[Gg]emfile`, `[Gg]emfile.lock` |

#### 4.2.2.2 Filter Immature Projects

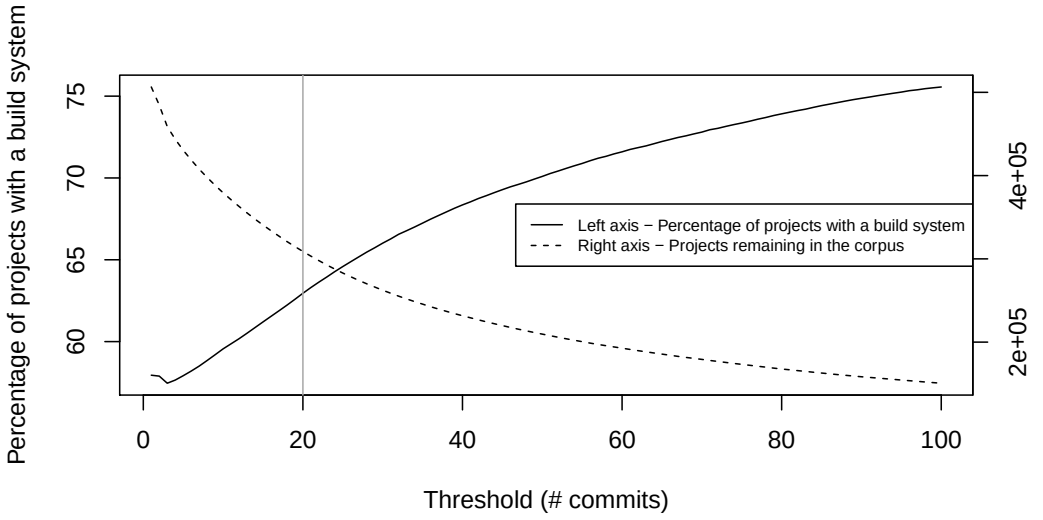
Software forges often contain projects that have not yet reached maturity. We apply three filters to remove repositories that: (1) do not represent software projects, or (2) are too small to require a build system and are hence not of interest in this study.

F1. Select a threshold for project size (measured in number of source files). Figure 4.2a plots threshold values against the number of surviving repositories and the percentage of those with detected build systems. We select a threshold of 15 source files because it appears near the knee of these two curves, and increases the percentage of repositories with detected build systems to 57% while only reducing corpus size to 506,413 repositories.

F2. Select a threshold for development activity (measured in number of commits). Figure 4.2b shows that selecting a 20 commit cutoff (at the knee of the two curves) reduces the corpus size to 306,798 repositories, while increasing the number of




(a) Project size.




(b) Development activity.

Figure 4.2: [Empirical Study 1] Threshold plots for filtering the corpus of repositories.

repositories with build systems to 193,283 (63%).

F3. Remove repositories where our classification tool marks more than 20% of the project files as unknown, since our results would ignore too much project activity. After applying this filter, 261,367 repositories survive.

Table 4.1 shows that of the 261,367 forge repositories that survive our filtering process, a corpus of 169,033 Github, 645 Gitorious, 602 repo.or.cz, and 217 RubyForge repositories contain detectable build systems, i.e., a total of 170,497 repositories (65%). Surprisingly, 35% of the surviving forge repositories did not have a detectable build system. The majority of these repositories contained web applications, e.g., PHP or JSP code. Savage suggests that the lack of build system uptake from web developers is worrisome.[8] For example, build systems for web applications are necessary to drive continuous delivery [45], i.e., automation of the source code deployment process, such that automatically tested code changes can be quickly deployed for end user consumption. Without a build system to automate the testing and deployment of web applications, projects often rely on error-prone, manual deployment processes. Since we focus on build maintenance in this chapter, we filter away projects without detected build systems.

Overall, we filtered the dataset to study software projects that are more likely to benefit from build technology. Our selection criteria eliminated 80% of the projects, i.e., those that are very small (less than 15 files) and those with little development activity (less than 20 commits). We also report results for the four large-scale software projects and three ecosystems to check if our findings are consistent in smaller, more carefully controlled development environments.

[8] http://www.brandonsavage.net/build-systems-relevancy-of-automated-builds-in-a-web-world/

### 4.2.3 Construct Meaningful Measures

For each of our research questions, we extract a set of measures from the repositories that survive the filtering process. We present the set of measures that we extracted for each research question in more detail in Sections 4.3 to 4.5.

### 4.2.4 Analyze and Present Results

After extracting metric values, we analyzed them using various visual aids such as line graphs, boxplots, and beanplots. These figures are also discussed in more detail in Sections 4.3 to 4.5.

## 4.3 Build Technology Adoption

In this section, we study build technology adoption by addressing our first two research questions.

### (RQ1) Which build technologies are broadly adopted?

We iterate over the changes in each repository, indicating that a repository uses a build technology if any of its files have names that match patterns for that technology (since a repository may use multiple build technologies, the percentages do not sum up to 100%). We show build technology adoption rates in Table 4.1 and Figure 4.3. We discuss our results with respect to the studied forges, ecosystems, and large-scale projects below.

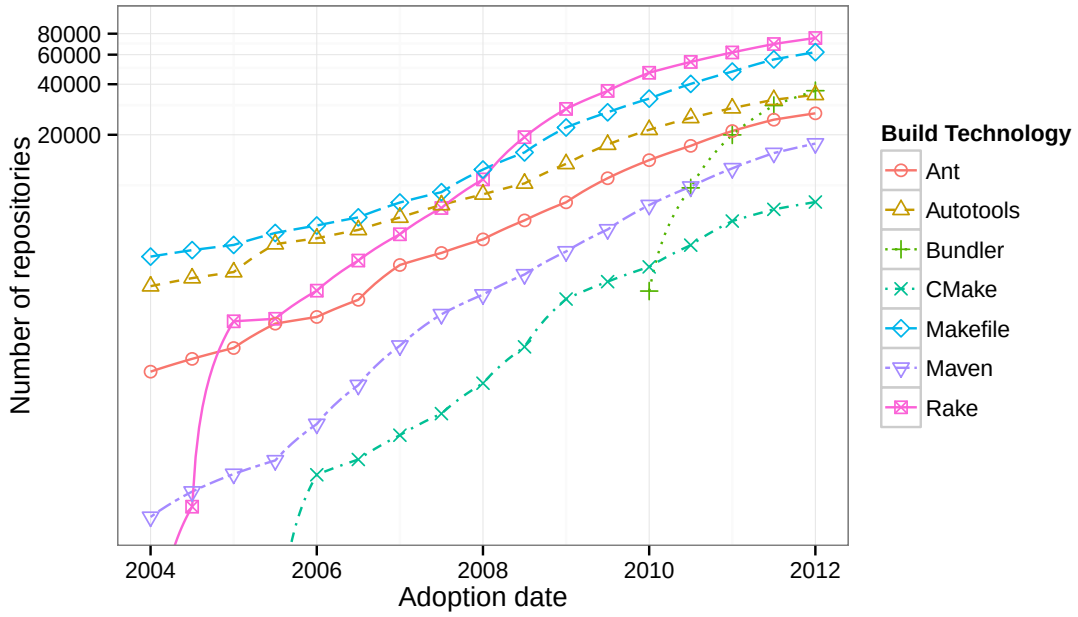
**(RQ1-1) Diversity in technology adoption**

Software forge repositories are rarely coupled to each other. Hence, we expect diversity in software forge build technology adoption. Table 4.1 shows that although there are technologies with broad adoption, there is also much diversity, with many different build technologies appearing in Github, Gitorious, and repo.or.cz forges. Rubyforge is composed of Ruby projects, and hence the Ruby-specific Rake technology is popular.

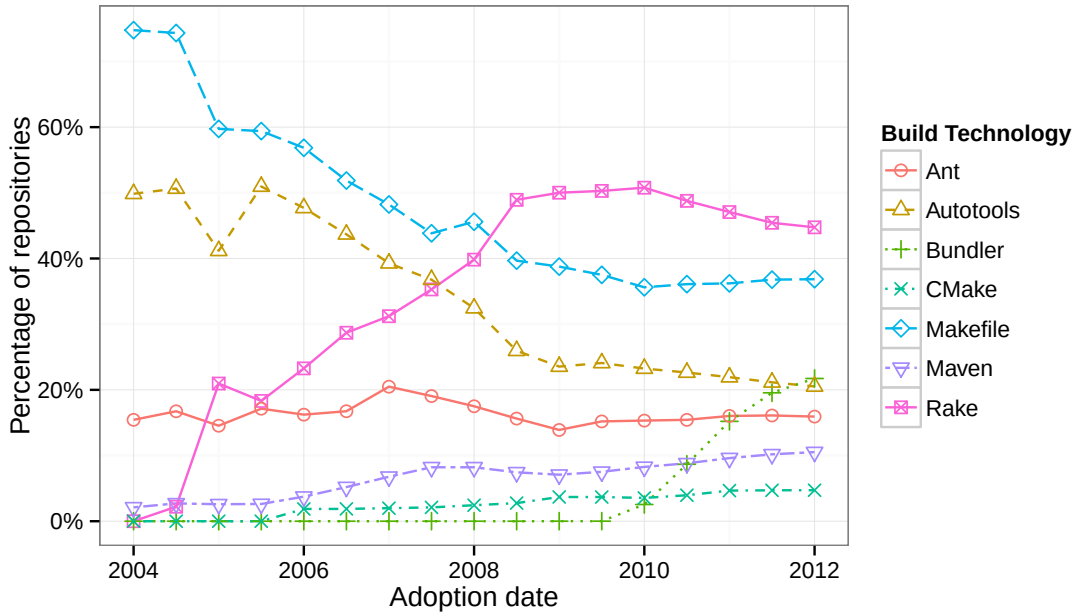
Software ecosystem repositories are loosely coupled, often being free to evolve independently of each other. However, ecosystems often enforce guidelines on project structure. Hence, we expect less diversity in build technology adoption within ecosystems when compared to software forges. Table 4.1 shows that ecosystems tend to converge on a small collection of build technologies. We expect that GNU and Apache ecosystems would use the tools that are developed within the ecosystem, i.e., GNU projects would use GNU Autotools or `make`, while Apache projects would use Apache Ant, Maven, and Ivy tools. The use of exterior tools like CMake and Rake in these ecosystems suggests that while technology convergence is often the case, developers have the freedom to experiment with other build technologies.

Large-scale project repositories are tightly coupled. Repositories encapsulate subsystems that are merged into a larger system using the build system. Hence, we expect to find little diversity in large-scale project technology adoption. Table 4.1 confirms our suspicion, with the Android, GNOME, and KDE projects adopting a single technology in more than 82% of project repositories.

PostgreSQL results in Table 4.1 show that the central technology can be used in tandem with other technologies. Autotools, `make`, and even Ant appear in 66%, 58%, and




(a) Number of repositories (Y-axis begins at 100 projects).




(b) Percentage of repositories.

Figure 4.3: [Empirical Study 1] Build technology adoption over time.

34% of the repositories respectively. Manual inspection of the PostgreSQL build system reveals that build configuration is implemented with GNU Autotools, while the construction step is implemented using `make`. Ant specifications are used to build a PostgreSQL Java Database Connectivity (JDBC) plugin, while the PGXN Utils repository, which provides an extension framework for PostgreSQL plugins is implemented using Ruby and uses the Rake and Bundler technologies to produce Ruby packages.

**Observation 1 — Language-specific technologies are growing in popularity.** Software forges show the highest degree of build technology diversity and hence offer an interesting benchmark for build technology popularity. Table 4.1 shows that `make` is still popular, appearing in many forge repositories. Language-specific tools like Ant and Maven (Java) are also popular. Even Rake and Bundler (Ruby) are popular outside of the Ruby-specific Rubyforge.

Figure 4.3a shows build technology adoption trends between 2004 and 2012 on a logarithmic scale. Prior to 2007, `make` and Autotools were the most popular technologies with consistent growth. However, Figure 4.3b shows that `make` and Autotools began to lose market share in 2005, due to an explosion of Rake-driven Ruby projects. In 2010, CMake began to gather momentum, and Bundler was initially embraced by the Ruby community. Ant and Maven show steady growth, with Ant having slightly more adoption.

*While many projects use traditional technologies like* `make` *and Autotools, language-specific technologies like Rake and Bundler capture are larger share of the open source market (Observation 1). Although researchers and service providers should continue to focus on older build technologies like* `make` *that still account for a large share of the open source market, more modern build technologies are gaining popularity and should also be considered.*

## (RQ2) Is choice of build technology impacted by project characteristics?

To address this research question, we focus on two major factors: (1) the size of the source code in the repository, and (2) the adopted programming languages. We hypothesize that these factors may impose limitations on build technology choice. For example, larger systems may require more powerful and expressive build technologies. Similarly, the use of a programming language may require technology-specific support to handle language-specific nuances. We use the forge and ecosystem data to address this research question because the repositories within them are rarely dependent on each other.

### (RQ2-1) Source Code Size

We use the number of source files within a repository as a measure of source code size. Although source code file count is a coarse-grained metric, prior work suggests that finer-grained metrics, such as SLOC, show similar evolutionary patterns in large datasets [43].

We use boxplots to provide an overview of the data with respect to the studied build technologies. Finally, we use Tukey Honestly Significant Difference (HSD) tests [75] to rank technology-specific samples to confirm that the differences that we observe in the boxplots are statistically significant ($\alpha = 0.01$). Since the Tukey HSD test assumes equal within-group variance across the groups, we transform source code size using $\ln(x + 1)$ in order to make the distribution of variances more comparable among the groups.

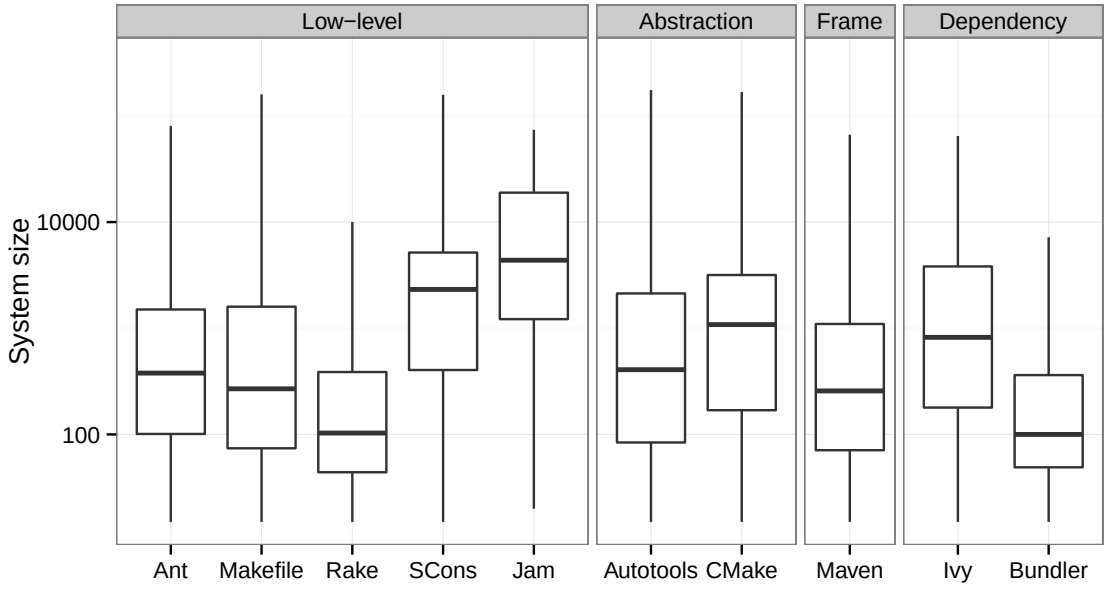



Figure 4.4: [Empirical Study 1] Size of the source code (# files) per repository in the forges and ecosystems.

**Observation 2 — Large repositories tend to adopt newer technologies earlier than smaller ones.** Figure 4.4 shows that the repositories using the Jam, SCons, and CMake technologies, i.e., the three technologies with the least adoption in our corpus (see Table 4.1), tend to have more source code files than the repositories using other build technologies. Tukey HSD test results indeed rank Jam as the largest sample, followed by SCons, and then CMake. On the other hand, the more mature technologies see adoption that spans a broader range of sizes, including several small repositories. Tukey HSD test results rank Maven, Make, and Ant near the bottom due to the mass of small repositories that adopt them. Although Rake and Bundler are newer build technologies, they occupy the bottommost rank according to the Tukey HSD test. We conjecture that this is due to the terse nature of the Ruby language that applications built using

Rake and Bundler are implemented in.

### (RQ2-2) Programming Language

We study the build technologies adopted by each language-specific group of repositories. As done in RQ1, we indicate that a repository uses a build technology if any of its files have names that match patterns for that technology. Since a programming language likely only becomes a build maintenance concern if a considerable proportion of the system is implemented in it, we do not consider programming language used unless at least 10% of its source files are implemented using that language.[9]

A common approach to model count data in contingency tables is via Poisson regression. We use it to describe co-occurrences of build technology and programming language: $\#Projects \sim forge + language + technology + language{:}technology$. A categorical predictor of the forge/ecosystem is included to control for the role that the repository host may play in the adoption of language or technology.

Figure 4.5 shows highly statistically significant connections ($p < 10^{-100}$) between build technologies and programming languages according to that model. The odds ratios are also presented, i.e., the ratio of the observed frequency to the likelihood of the co-occurrences of technology and programming language if they were independent events. We apply the logarithm to the odds ratio, since the values can be quite large.

**Observation 3 — Programming language choice shares a relationship with build technology choice.** If there was truly no relationship between language and build technology choice, we would expect that the technology usage in each group would be

[9] Threshold values of 5% and 15% yielded similar results.

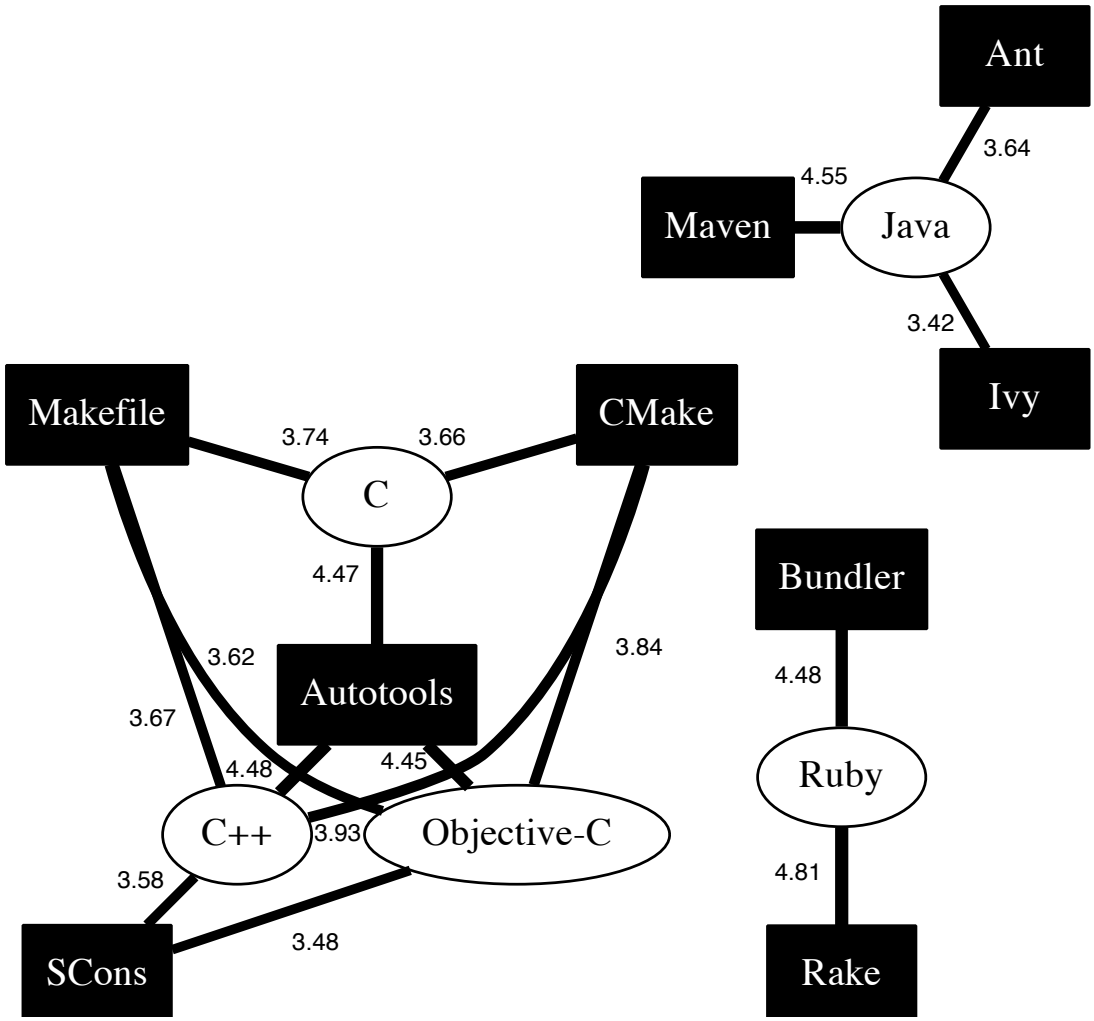



Figure 4.5: [Empirical Study 1] Statistically significant ($p < 10^{-100}$) co-occurrences of build technology (black boxes) and programming language (white ovals) on a fitted Poisson model. The higher the log odds ratio presented above each edge, the higher the likelihood of a non-coincidental relationship.

similar. However, the formation of clustered groups of technologies around programming languages in Figure 4.5 shows that each language has prevailing build technologies. For example, Ant, Maven, and Ivy are quite popular for Java projects, while Rake and Bundler are almost unanimous choices for Ruby projects. C, C++, and Objective-C projects favour `make`, Autotools, and CMake.

Furthermore, the data suggests that language-specific technologies are growing in popularity. Figure 4.3 shows that language-specific technologies like Rake, Bundler, Ant, and Maven have grown rapidly in the past few years, while Figure 4.5 confirms that Rake and Bundler are de facto build technologies for Ruby repositories, and Ant and Maven share the bulk of Java repositories.

*Large projects tend to adopt newer technologies earlier than small projects do (Observation 2). Furthermore, there is a strong relationship between the programming languages used to implement a system and the build technology used to assemble it, which may limit the scope of technologies considered by software projects (Observation 3). Build technologies that are tailored for specific programming languages have grown quite popular as of late, suggesting that tool developers and service providers should follow suit.*

## Discussion

The studied technology adoption trends (RQ1) indicate that the use of traditional build technologies like `make` and Autotools are still prevalent in the software forges, ecosystems, and large-scale systems. However, language-specific technologies are growing in popularity (Observation 1). We also observe that there is a strong relationship between programming language and technology choice (Observation 3).

### The Trade-off between Language-Agnostic and -Specific Build Technologies

Language-specific tools are almost unanimous choices for Java and Ruby systems. Figure 4.5 indicates that Java projects often select build technologies like Ant, Maven, and Ivy, while Ruby systems select Rake and Bundler most frequently. These language-specific build technologies offer several advanced features that are tailored for building projects of the respective languages. For example, language-agnostic tools like `make` check that each target is up-to-date with its dependencies in order to detect whether the recipe should be executed. However, the Java compiler will perform these same checks, potentially recompiling out of sync dependencies automatically. Being aware of this feature of the Java compiler, Ant and Maven technologies defer `.class` dependency checks to the Java compiler. This feature of Ant and Maven likely make them more appealing to Java developers than language-agnostic alternatives.

When selecting a technology to adopt, software teams evaluate a trade-off between the flexibility of language-agnostic tools like `make` and feature-rich language-specific technologies like Maven. While it appears that repositories using modern languages like Java and Ruby favour the latter, C, C++, and Objective-C teams are still frequently adopting `make`. Indeed, despite lacking the powerful language-specific features that tools like SCons, CMake, and Autotools provide, `make` is still quite popular among C, C++, and Objective-C systems. Figure 4.3a shows that `make` continues to grow, albeit more slowly than more modern technologies. For example, during the planning of a build technology migration, the Apache OpenOffice (AOO) team recently evaluated two primary options: `make` and CMake.[10] While debate is still ongoing, the AOO team

[10] https://wiki.openoffice.org/wiki/Build_System_Analysis

highlights several advantages that `make` maintains over CMake. For example, `make` supports pattern-based dependency expressions, while CMake does not. Moreover, CMake specifications generate build systems on UNIX platforms that follow the notably flawed recursive `make` paradigm [74] that the AOO aims to avoid.

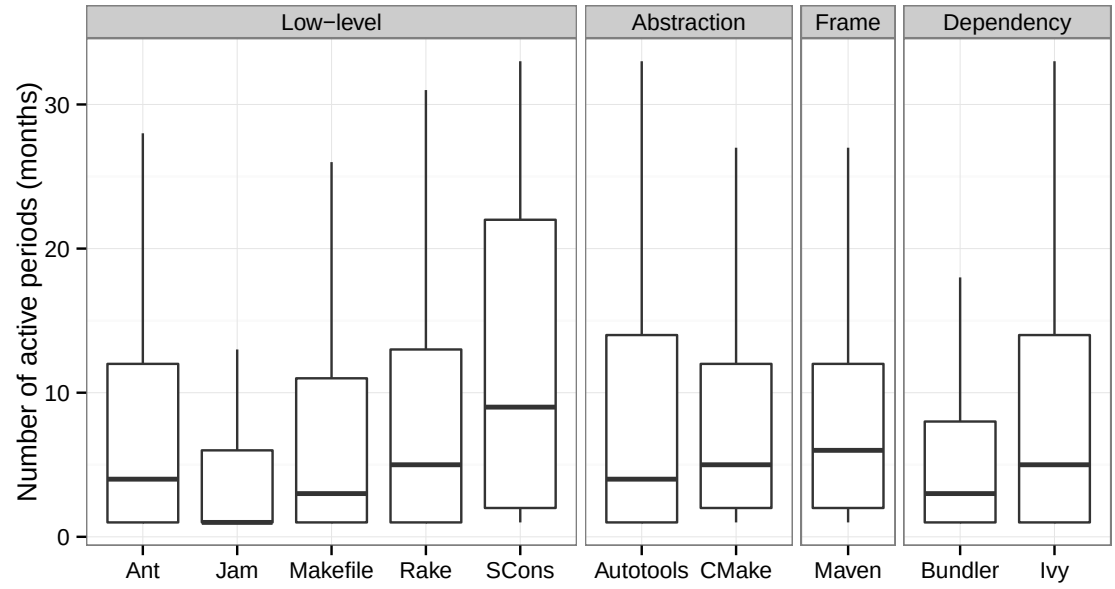
The sustained popularity of `make` among C, C++, and Objective-C repositories may also be due to the fact that the compilation and linking model are congruent with the `make` dependency model. C, C++, and Objective-C compile and link tools require a low-level dependency tool to manage dependencies between source, object, and executable code. On the other hand, there is a mismatch between the dependency model of `make` and the Java compiler, creating the need for language-specific build tool support for Java systems.

## 4.4 Build Maintenance

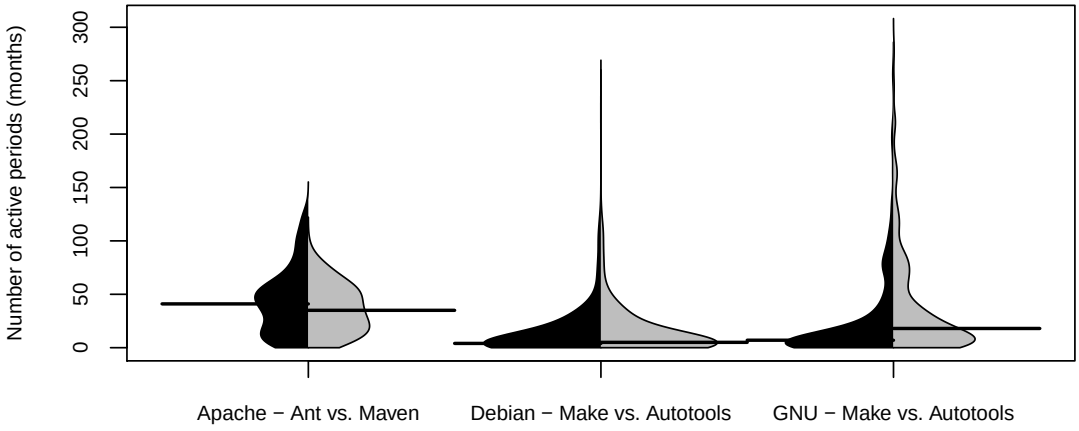
In this section, we study the relationship between build technology and build maintenance by addressing RQ3 and RQ4.

### (RQ3) Does build technology choice correlate with build change activity?

We select metrics that measure three dimensions of build change activity, and calculate them on a monthly basis. Table 4.3 describes the metrics that we consider and provides our rationale for selecting them. The build commit proportion is normalized in order to control for overall system activity. We do not normalize build commit size nor build churn volume in order to simplify interpretation of the results. We use size




(a) Software forges.




(b) Software ecosystems.

Figure 4.6: [Empirical Study 1] Number of active periods (months) per repository in the forges and ecosystems.

Table 4.3: [Empirical Study 1] Build maintenance activity metrics.

| Metric | Description | Rationale |
|---|---|---|
| Build commit proportion | The proportion of commits that contain a change to a build specification. | Frequently changing build systems are likely more difficult to maintain. |
| Build commit size | The median number of build lines changed by a build commit in a given period. | Technologies that often require large changes are likely more difficult to maintain. |
| Build churn volume | The total number of build lines changed in a given period. | Frequently churning build systems are likely more difficult to maintain. |

(i.e., build commit size) and rate of change (i.e., build commit proportion and build churn volume) metrics in lieu of change complexity ones because prior work suggests that complexity tends to be highly correlated with size in both the source code [40] and build system domains [67].

We consider the commits that contain a build change, including those that also contain other changes, as build commits. We include commits that change the build system as well as other parts of the system because any commit that changes the build system is the result of some measure of build maintenance.

Since projects can migrate between technologies, we consider a technology active in a repository for all months between (and including): (1) the month where commit activity of files of its type first appear, and (2) the last month with commit activity of a file of its type. To gain some insight into the maturity of the technology use in the corpus, Figure 4.6 shows the distribution of commit activity (in number of months) for a specific technology. **The upper end of the boxes in Figure 4.6a indicates that at least one quarter of the repositories with Ant, Make, Rake, SCons, Autotools, CMake, Maven, and Ivy have at least 12 active months.**

We focus our ecosystem studies on comparing Ant and Maven in Apache, and Make and Autotools in Debian and GNU, since the ecosystems mostly converge on those technologies. Figure 4.6b compares the distributions of active months in the studied ecosystems. **Figure 4.6b shows that we study several mature Apache projects, with a median active month count of 35 (Maven) and 41 (Ant).** The GNU and Debian ecosystems have longer tails, dating back to 1988 and 1993 respectively.

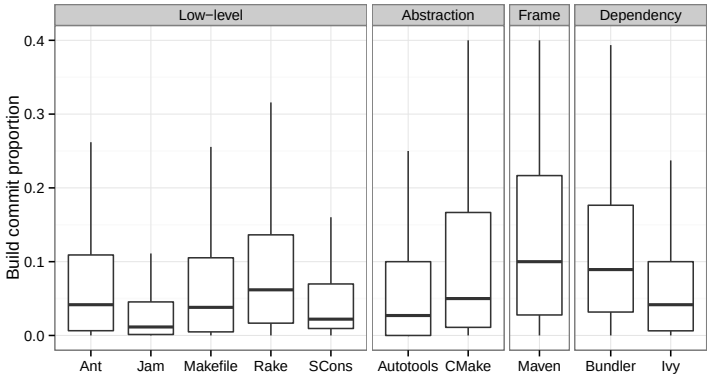
Our analysis treats each technology independently, e.g., if a repository uses both `make` and SCons, we calculate separate values for the metrics in Table 4.3 for `make` and SCons. We then measure the distribution of metric values for each technology and we rank these distributions to identify the build technologies with the highest or lowest values using Tukey HSD tests ($\alpha = 0.01$). We transform the commit proportion using $\arcsin(\sqrt{x})$, and build commit size and build churn volume using $\ln(x+1)$ to make the distribution of variances more comparable among the groups (*cf.* Tukey HSD test assumptions). Since there are two main technologies used in each of the studied ecosystems, we use Mann-Whitney U tests [13] instead of Tukey HSD tests to compare them ($\alpha = 0.01$).

Figure 4.7 shows the distributions of metric values in the forges. To ensure that each repository is equally considered in our analysis, we select the median value for each metric from each repository. We complement our median-based analysis by performing a longitudinal analysis of each metric in the forges. We examine the ranks of each technology as reported by the Tukey HSD tests when applied to each metric on a monthly basis. The ranks are in decreasing order, i.e., the technology that has the highest metric values appears in rank one. Figures illustrating the monthly trends are provided in Appendix B.

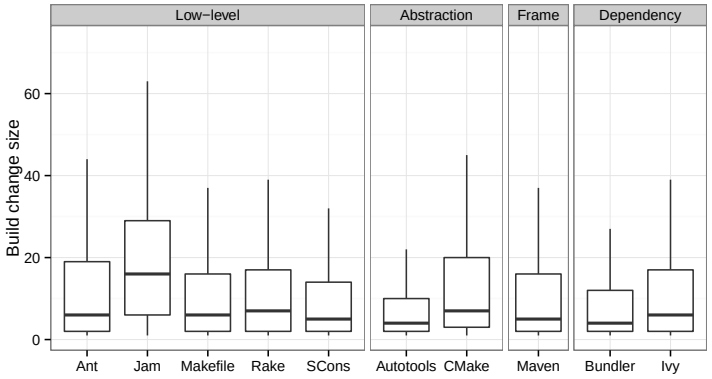
**Observation 4 — Maven requires the most build maintenance activity.** Maven tends to require a larger proportion of monthly commits than low-level technologies do. Figure 4.7a shows that the Maven distribution has the highest median value. Analysis of twelve months of activity shows that Maven maintains the top Tukey HSD rank (see Appendix B).

Figure 4.7a suggest and the Tukey HSD test confirm that the median Autotools build commit proportion tends to be lower than that of the other technologies in the abstraction, low-level, and dependency management categories. Furthermore, Autotools never appears in the top three ranks of the 12-month Tukey analysis, while CMake and Bundler never appear lower than the third rank (see Appendix B). We observe that of the 34,963 forge projects that use Autotools, 7,438 (21%) only implement the configuration step using Autotools while using `make` to implement the construction step. In this case, Autotools cannot be fairly compared to tools being used to implement complete build systems. After filtering away repositories that use both Autotools and hand-written `make` specifications, the Autotools distribution grows to similar proportions as the CMake one. Ivy also ranks near the bottom, but is frequently used in tandem with Ant. When these technologies are grouped together, the distribution grows to proportions similar to Maven.

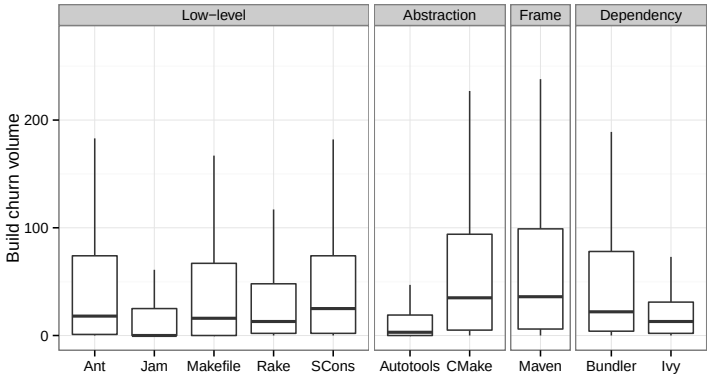
Figure 4.7b shows that there is much more parity in the distributions of build change sizes than of build commit proportion. We observe that Jam, Ant, and CMake stand out as requiring larger changes than the other technologies in the median analysis of Figure 4.7b, while Maven and SCons make more frequent appearances in the top three ranks of the monthly analysis (see Appendix B).




(a) Build commit proportion.




(b) Build change size.




(c) Build churn volume.

Figure 4.7: [Empirical Study 1] Median build commit proportion, size, and churn in the studied forges.

Figure 4.7c shows that the median build churn volume for framework-driven specifications is higher than that of the other technologies. Tukey HSD tests of the median samples confirm that the Maven rates are the highest, followed by CMake, and then SCons. Tukey HSD 12-month analysis complements the median results, with Maven and SCons never appearing below the second rank (see B). CMake only drops to the third rank in the seventh month, appearing in the top two ranks for all other months.

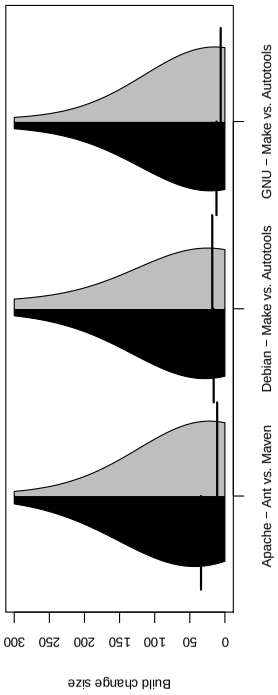
Corroborating our findings in the software forges, Figure 4.8a shows that Maven tends to require a larger proportion of monthly build changes than Ant in the Apache ecosystem. Indeed, while Figure 4.8b suggests that Maven commits tend to be smaller than Ant commits in the Apache ecosystem, Figure 4.8c shows that on a monthly basis, Maven still induces more churn than Ant in the Apache ecosystem. Mann-Whitney U tests confirm that the reported differences are significant.

On the other hand, although Autotools requires a larger proportion of project commits in both the Debian and GNU ecosystems, `make` changes tend to induce more churn. Mann-Whitney U tests confirm that the GNU churn volume differences are significant, however Debian results are inconclusive.

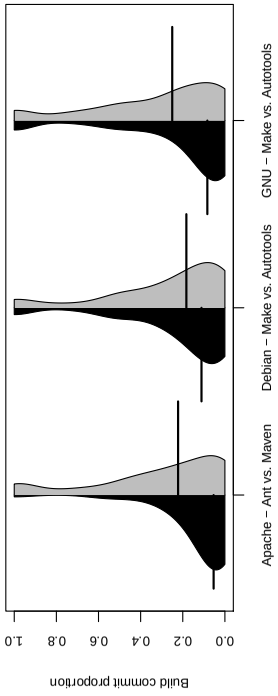
*Framework-driven technologies like Maven tend to have a higher build commit proportion and induce more build churn than low-level or abstraction-based technologies (Observation 4). While modern build technologies provide additional features, development teams adopting them should be aware of potentially higher maintenance overhead.*

## Discussion

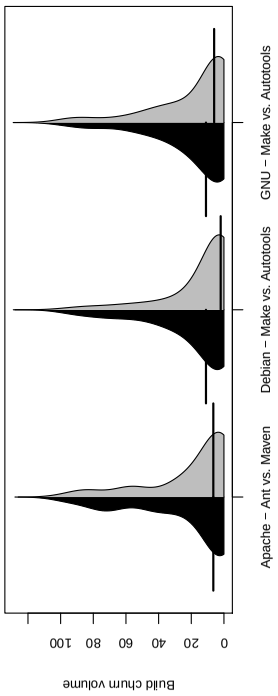
While the sizes of Jam and SCons changes are noteworthy, in addition to the tendency of being used in larger systems (Observation 2), they are also low-level technologies,




(b) Build commit sizes.




(a) Build commit proportion.




(c) Build churn volume.

Figure 4.8: [Empirical Study 1] Median build commit proportion, size, and churn in the studied ecosystems.

Table 4.4: [Empirical Study 1] Build maintenance overhead metrics.

| Metric | Description | Rationale |
|---|---|---|
| Source-build coupling | The logical coupling (Equation 4.1) between source code and build system changes. | High source-build coupling indicates that developers often need to provide accompanying build changes with their code changes, which may be distracting and costly in terms of context switching. |
| Build author ratio | The logical coupling (Equation 4.1) between source code and build system authors. | High build author ratios suggest that a large proportion of source code developers are impacted by build maintenance. |

and are therefore expected to be more verbose than the other technologies. Ant and Maven change sizes may be inflated because of the verbose nature of the XML markup [62]. The verbosity of CMake changes is surprising, since CMake is an abstraction-based technology — a quality that one would expect to decrease change size.

As described in the discussion of Section 4.3, the AOO team has remarked that the feature for expressing pattern-based build dependencies available in the popular GNU variant of `make` was missing in CMake. Hence, pattern-based dependencies need to be repeated several times using CMake. Furthermore, when a change needs to be made, it will need to be repeated several times, which may explain the inflation of CMake build sizes we observe.

## (RQ4) Does build technology choice correlate with the overhead on source code development?

Similar to our prior work [70], we select metrics that measure build maintenance overhead using *logical coupling* [38], which is calculated as shown below:

$$LC(source \Rightarrow build) = \frac{Support(source \cap build)}{Support(source)} \tag{4.1}$$

Note that $Support(X)$ in Equation (4.1) is the number of commits that satisfy the clause $X$.

Table 4.4 describes the metrics we consider and provides our rationale for selecting them. Similar to RQ3, we calculate each metric on a monthly basis. Source-build coupling is calculated independently for each technology used in each repository, e.g., $LC(source \Rightarrow Ant)$ and $LC(source \Rightarrow Maven)$.

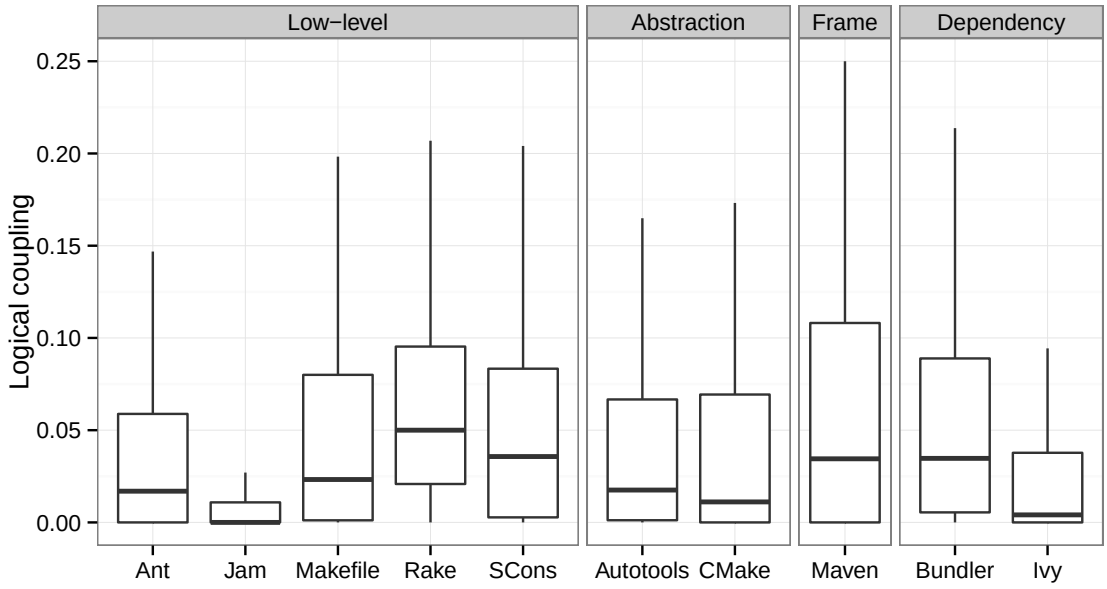
Note that in order to calculate the build author ratio, we need to identify the original author of each change. A common practice in open source development is to restrict VCS write access to a set of core developers [16]. Many authors send their changes to the core developers for their consideration. After engaging in a review process, the core developer will either discard the changes or commit them to the VCS. Note that modern VCSs allows committers to record the original author's name, distinguishing the roles of author and committer. We use the original author name that is recorded in the studied Git repositories as the developer responsible for a change. Thus, insofar as developers use this modern VCS feature to distinguish between authors and committers, our build author ratio analysis does not lose the original authorship information.

Figure 4.9 shows the distribution of median source-build coupling and build author ratio measures in the forge repositories. In the same vein as RQ3, we apply the Tukey HSD test to the software forge data and the Mann-Whitney U test to the software ecosystems data to detect significant differences among the resulting distributions. We again apply the $\arcsin(\sqrt{x})$ to the source-build coupling and build author ratio prior to applying the Tukey HSD test to make the distribution of variances more comparable among the groups (*cf.* Tukey HSD test assumptions). We again complement our median analysis with a monthly analysis of the Tukey ranks in Appendix B.

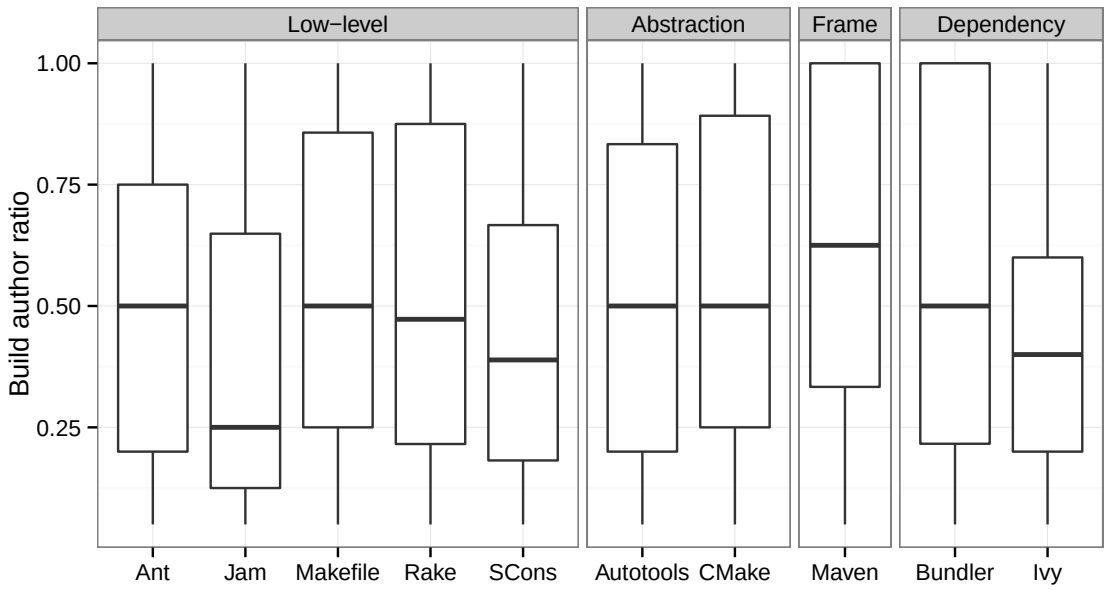
**Observation 5 — Maven changes tend to be tightly coupled to source code changes.** Figure 4.9a shows that Maven changes tend to be tightly coupled with source code changes. A Tukey HSD test ranks Maven in the top rank, followed by Rake, and then `make`. The monthly Tukey analysis shows that Maven also appears alone in the top rank for the twelve analyzed months (see Appendix B). This is surprising because one would expect that Maven's framework-driven behaviour would reduce the source-build coupling.

Figure 4.9b shows that the Maven changes tend to be more evenly dispersed among developers than changes of other technologies are. Tukey HSD tests confirm that a larger proportion of developers for Maven projects make build changes than developers using the other technologies. The median Maven build author ratio is 65%, indicating that in half of the studied Maven repositories, at least 65% of the source code authors also make build changes. Maven and SCons require the largest proportion of developers, with Tukey HSD tests ranking Maven and SCons in the top two ranks consistently throughout the twelve analyzed months (see Appendix B).

Turning to the software ecosystems, Figure 4.10a shows that Maven and Autotools

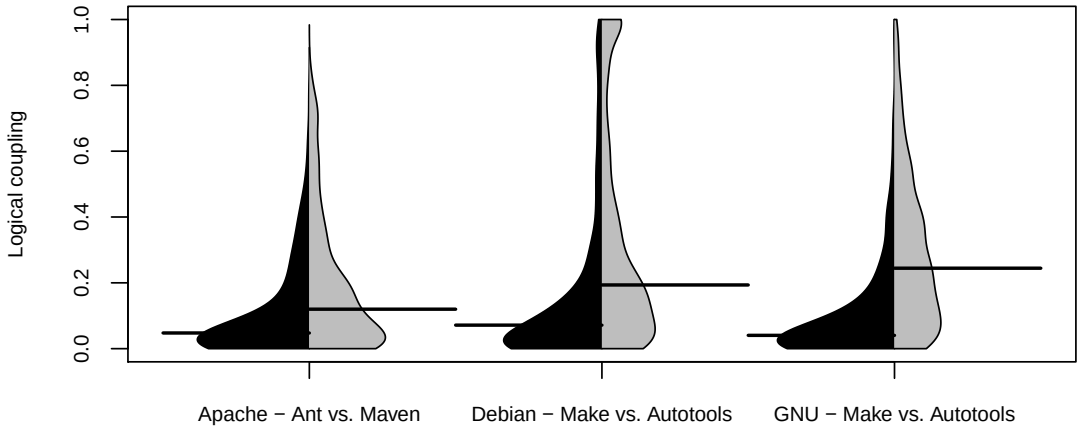



(a) Source-build coupling.

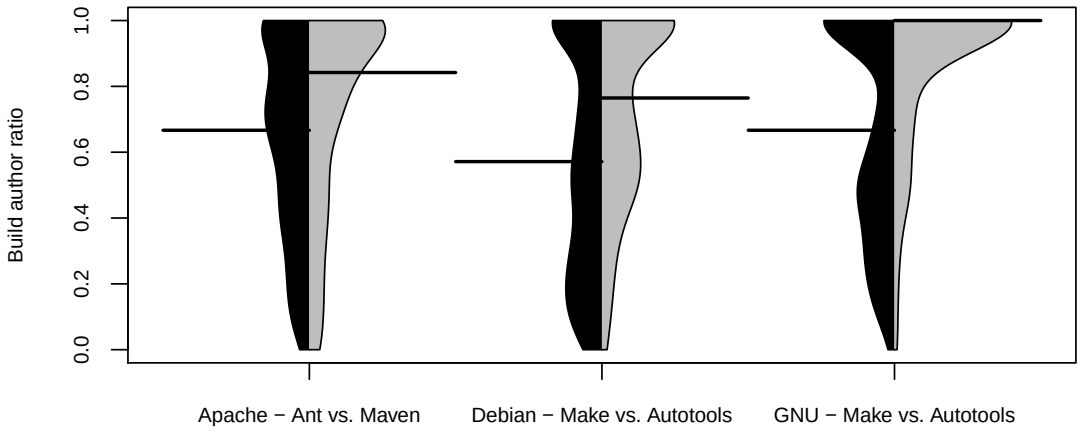



(b) Build author ratio.

Figure 4.9: [Empirical Study 1] Median source-build coupling and build author ratios in the studied forges.




(a) Source-build coupling.




(b) Build author ratio.

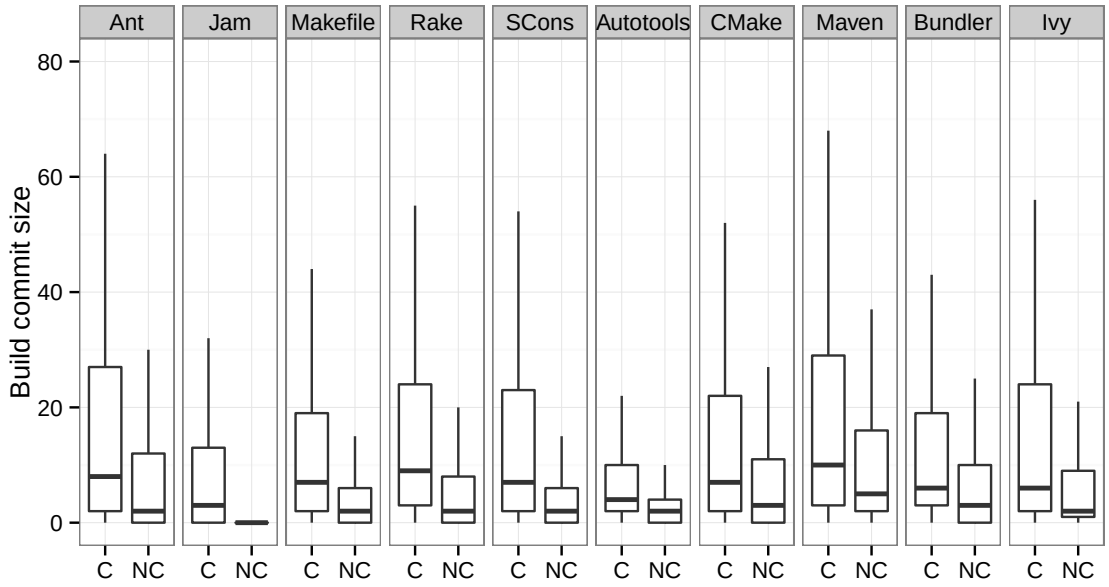
Figure 4.10: [Empirical Study 1] Median source-build coupling and build author ratios in the studied ecosystems.

tend to be more tightly coupled to source changes than Ant and `make`. Furthermore, Figure 4.10b shows that Maven and Autotools changes tend to be more evenly dispersed among developers than Ant and Make changes. Mann-Whitney U tests confirm that these differences are significant.

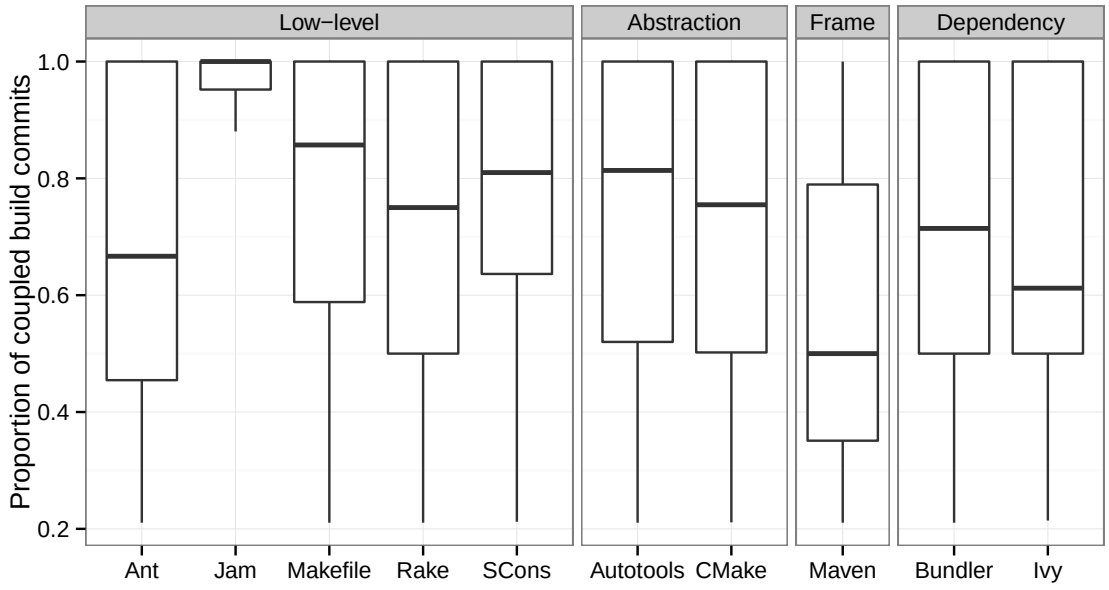
The finding that is most consistent across the software forges and ecosystems is that Maven changes tends to be tightly coupled to source code changes. To that end, a larger proportion of the development team tends to become involved in maintaining the Maven specifications.

**Observation 6 — Build change more often co-occurs with source change than without.** Figure 4.11 shows the distributions of build commit sizes and proportions of source-coupled (and non-coupled) build changes in the software forges. Irrespective of technology, source-coupled build changes tend to induce more build churn than non-coupled ones do, indicating that the build system changes most in tandem with changes in the source code.

Mann-Whitney U tests of the coupled and non-coupled build changes for each technology separately confirm that, as suggested by Figure 4.11a, source-coupled build changes tend to be larger than non-coupled ones. Furthermore, higher-level build technologies such as Maven and CMake have the largest source-coupled changes. A Tukey HSD test of the source-coupled changes of each technology indicates that Maven and CMake source-coupled changes are indeed the largest, however they are indistinguishable from each other. Furthermore, the proportions of build changes that are accompanied with source changes shown in Figure 4.11b indicate that, with the exception of Maven, build changes tend to occur more frequently with source changes than without.




(a) Size of build changes when coupled with (C) or not coupled with (NC) source code changes.




(b) Proportion of build changes that are accompanied with source code changes.

Figure 4.11: [Empirical Study 1] Comparison of coupled and not coupled build changes.

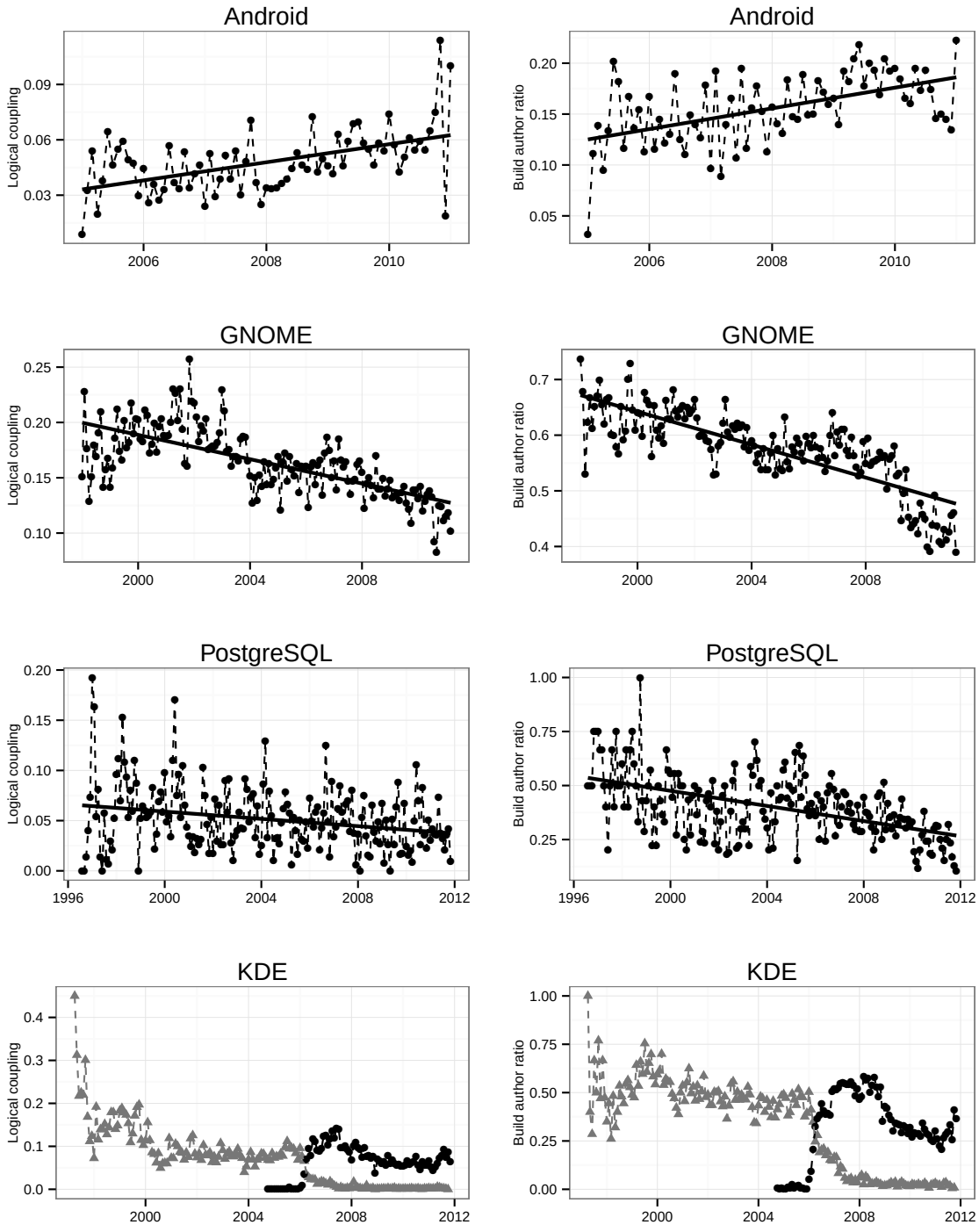



Figure 4.12: [Empirical Study 1] Monthly source-build coupling rate (left) and monthly build author ratio (right) in Android (`make`), GNOME (Autotools), PostgreSQL (Autotools), and KDE (Autotools in grey, CMake in black).

**Observation 7 — Coupling tends to decrease over time.** Interestingly, we find that framework-driven and abstraction-based technologies do not have lower source-build coupling rates than low-level technologies. In fact, Maven build changes in the forges and Apache projects are more tightly coupled to source code changes than Ant build changes are. Moreover, the maintenance of framework-driven specifications typically impacts a larger proportion of developers.

To study the stability of build overhead on source maintenance activities, we analyze how source-build coupling and build author ratio evolve. We analyze stability in the large-scale projects, since the longitudinal analysis required would be infeasible for the number of repositories in the forges and ecosystems. We focus our analysis on the most active build technologies of each large-scale project. Table 4.1 shows that `make` is the most active technology used in Android, while Autotools is used by GNOME and PostgreSQL, and KDE uses CMake. PostgreSQL also uses `make`, but we omit the trend because it is quite similar to the Autotools trend and clutters the figure. KDE used Autotools prior to their migration to CMake [100], hence we study trends with respect to both technologies.

Figure 4.12 shows that source-build coupling tends to decrease over time. Regression lines highlight the decreasing GNOME and PostgreSQL trends. Conversely, Android coupling trends are increasing. However, early Android development months had coupling rates below 0.05, so it is not surprising that the rate has grown to levels that are more comparable to other `make` projects.

The decreasing trends in build author ratio in Figure 4.12 suggest that as projects age, they adopt a concentrated build maintenance style, where a small team produces most of the build changes. Initially, the GNOME project had months where up to 74%

of the developers submitted build changes, while recently, the trend decreased to 39%. Similarly, PostgreSQL build changes were initially quite dispersed, peaking in late 1998 when every active developer submitted a build change. Recently, the trend has dropped as low as 10%.

*Framework-driven and abstraction-based build specification changes tend to be more tightly coupled to source code (Observation 5), impact a larger proportion of developers (Observation 5), and induce more churn (Observation 6) than low-level build specification changes. Yet, as large-scale projects age, source-build coupling tends to drop (Observation 7) and specialized build maintenance teams tend to emerge. Likely due to inflated source-build coupling rates, changes to framework-driven technologies tend to be more evenly dispersed among developers. When selecting build technologies, teams should consider whether this dispersion of build changes is tolerable.*

## Programming Language Centric Technology Analysis

We have shown that framework-driven build technologies trigger the most build activity (Observation 4) and tend to be more tightly coupled to source code changes than the other build technologies (Observation 5). However, in Section 4.3, we observed that build technology choices are often constrained by the programming languages that are used (Observation 3). For example, Maven is a Java-specific build technology, and hence requires additional effort to build C projects. To provide a more practical perspective, we need to compare build technologies within the scope of each programming language. We do so using the software forges, where the most diversity in build technology adoption was observed (*cf.* Section 4.3).

We first categorize the technologies typically used by a programming language by examining Figure 4.5. In doing so, we produce the below mapping:

**Java** $\longrightarrow$ Ant, Ivy, Maven

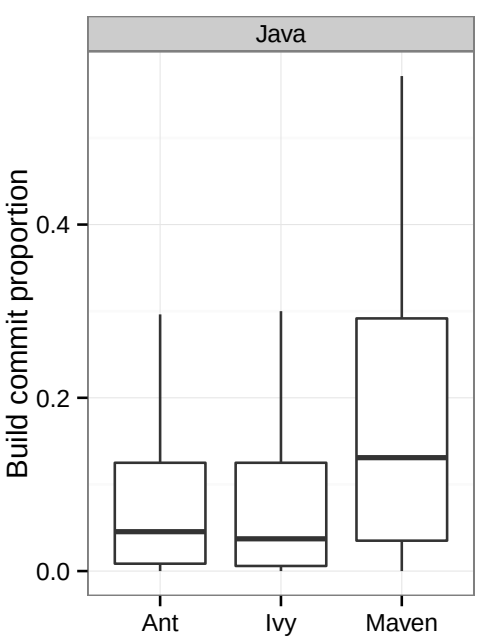
**C, C++, Objective-C** ⟶ `make`, Autotools, SCons, CMake

**Ruby** ⟶ Rake, Bundler

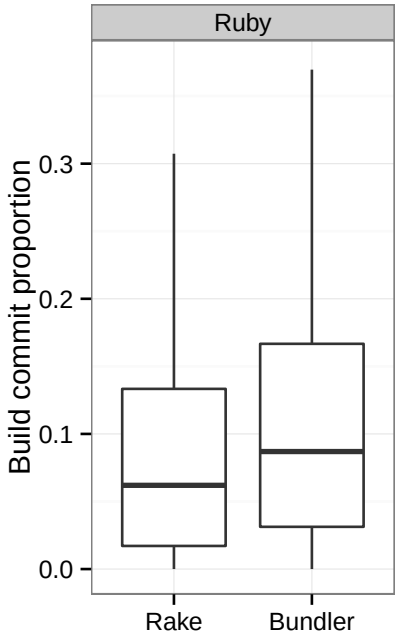
Next, we label each repository by examining the programming languages that are used. Note that a repository may use several programming languages, and hence may be labeled several times. Just as we did in our study of programming languages in Section 4.3, we indicate that a repository uses a programming language if more than 10% of its source files are implemented using that language. Finally, we calculate the build commit proportion and source-build coupling (Equation (4.1)) metrics of each labelled repository to compare the use of build technologies for each programming language separately.

#### Language-Specific Build Commit Proportion

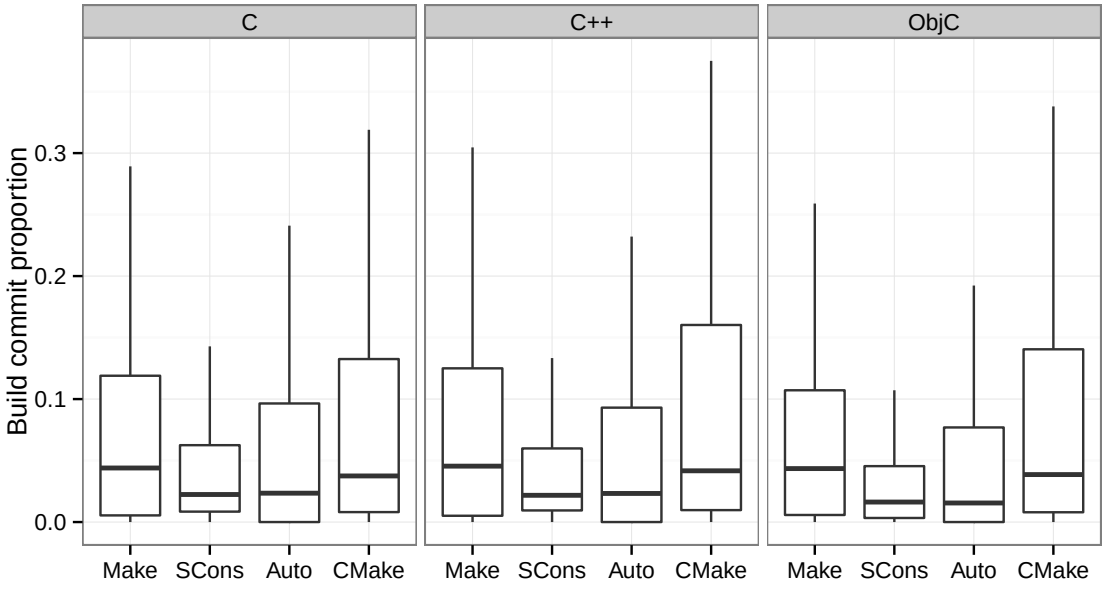
**Observation 8 — External dependency management specifications require plenty of maintenance.** Figure 4.13 shows the monthly build commit proportion for each group of programming languages. Figure 4.13a confirms that Maven specifications do indeed change most frequently among the build technology choices for Java programs. Tukey HSD tests confirm that the differences are statistically significant. Again, Ivy and Ant appear to require the least amount of change, however they are often used in tandem with each other. When combined, the distribution grows to proportions similar to Maven. However, a Mann-Whitney U test indicates that Maven specifications still change more frequently than combined Ant and Ivy specifications do, suggesting that Ant with Ivy may be a more cost-effective alternative than Maven for Java projects that express external dependencies (from the point of view of build maintenance).




(a) Java

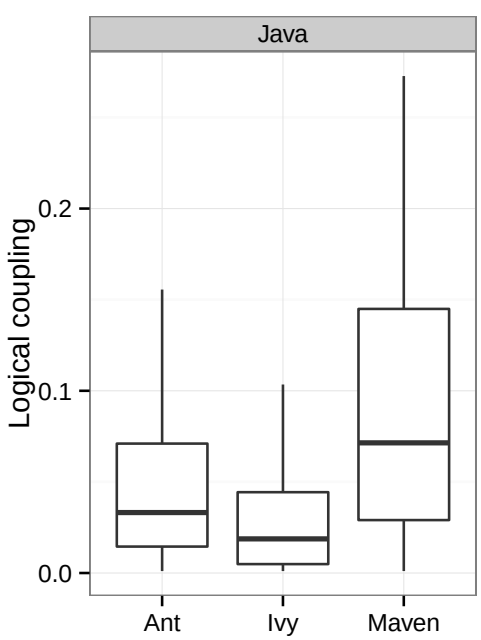



(b) Ruby

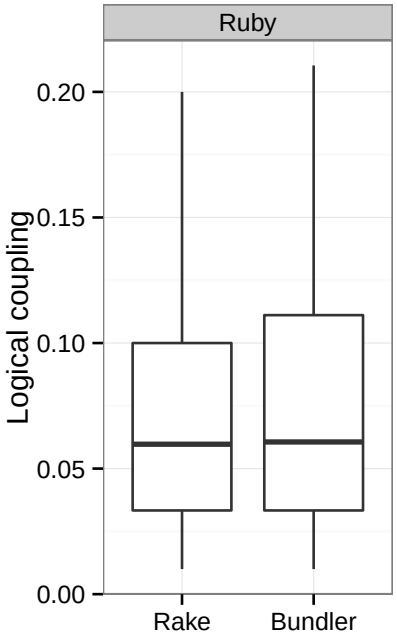



(c) C family

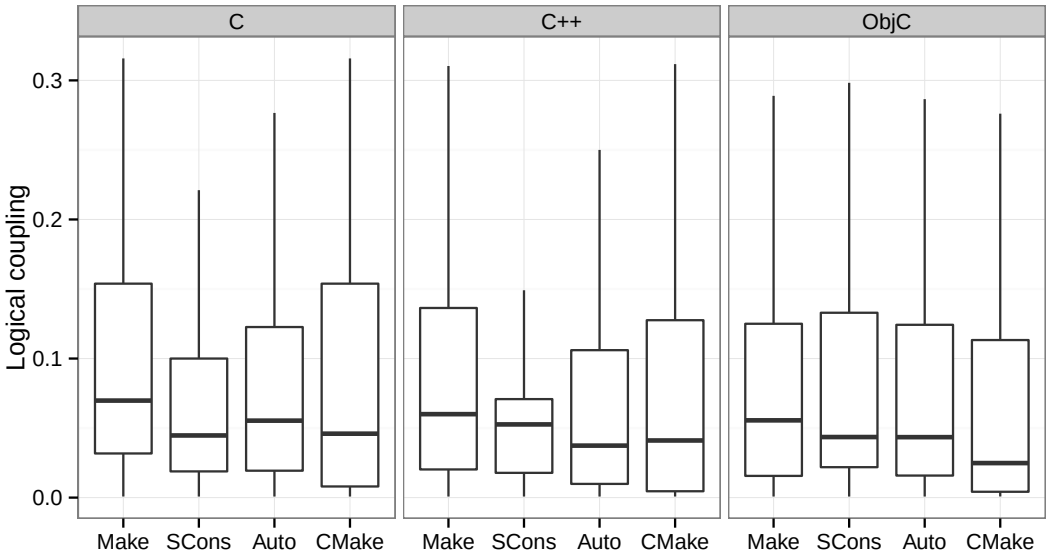
Figure 4.13: [Empirical Study 1] Build commit proportion in the studied forges classified by source languages used.




(a) Java




(b) Ruby




(c) C family

Figure 4.14: [Empirical Study 1] Source-build coupling in the studied forges classified by source languages used.

As shown in Figures 4.13a and 4.13b, the specifications that denote external project dependencies (i.e., Ivy and Bundler) have similar commit proportion as (if not higher than) the specifications that define build behaviour (i.e., Ant and Rake). This indicates that for Java and Ruby systems, a large amount of build maintenance activity is generated by external rather than internal dependency management specifications.

**Language-Specific Source-Build Coupling**

Figure 4.14 shows the source-build coupling between build technologies and programming languages. Figure 4.14a shows that similar to the overall coupling in Figure 4.9a, Maven is tightly coupled to Java code. This reinforces Observation 5, suggesting that Maven changes are indeed tightly coupled with source code.

While Figure 4.13c shows that CMake specifications have a higher commit proportion than the other C family technologies, Figure 4.14c shows that CMake has the lowest median coupling rate for C and Objective C. This finding suggests that C and Objective C projects can reduce source-build coupling by migrating to CMake.

*External dependency management accounts for much of the build maintenance activity in Java and Ruby repositories (Observation 8). Indeed, Maven specifications tend to be tightly coupled to Java source code. CMake tends to be loosely coupled with C family source code changes. Since Ant with Ivy tends to change less frequently than Maven and offers a comparable feature set, it is an option that Java project teams should consider. Furthermore, C and Objective-C projects should consider CMake, since CMake repositories tend to have lower source-build coupling rates than the other C and Objective-C repositories.*

### Discussion

Surprisingly, we find that use of Maven is often accompanied with (1) higher build maintenance activity rates (Observation 4), (2) tighter coupling between source code and build system changes (Observation 5), and (3) a higher dispersion rate of changes among team members (Observation 5). We assert that these rate, size, and authorship measurements of build changes capture relevant dimensions of build maintenance. However, the build system is a means to improve overall maintenance team productivity. In other words, the increases in build maintenance that we observe in Maven may actually be a net benefit to the development team if Maven offers additional features that accelerate the development process. We plan to investigate the complex interplay between build and overall maintenance effort in future work.

## 4.5 Build Technology Migration

In this section, we study whether build technology migration eases the burden of build maintenance by addressing RQ5.

### (RQ5) Does build technology migration reduce the amount of build maintenance?

A recent trend suggests that projects are migrating towards CMake[11,12] [63] and Maven.[13] Hence, we focus our migration study on these technologies. Specifically, we compare

[11] http://www.lenzg.net/archives/291-Building-MySQL-Server-with-CMake-on-LinuxUnix.html

[12] http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4

[13] http://lists.jboss.org/pipermail/hibernate-dev/2007-May/002075.html

median monthly churn rate, source-build coupling, and build author ratios pre- and post-migration using Wilcoxon signed rank tests ($\alpha$ = 0.01). We use Wilcoxon signed rank tests instead of Mann-Whitney U tests because we have paired observations, i.e., the same project pre- and post-migration.

We automatically detect repositories that have migrated to CMake or Maven technologies by checking if CMake or Maven specifications appear in the repository at least one period after another technology. Our approach detects 89 ecosystem project migrations ($\approx 2\%$) and 7,225 forge project migrations ($\approx 4\%$). While prior work has studied build technology migration (e.g., Suvorov *et al.* [100]), the focus has generally been on migration in a few large projects. To the best of our knowledge, this is the first build migration study to focus on a large collection of migrations.

**Observation 9 — Build technology migration often pays off.** Figure 4.15 shows that, despite Maven projects typically having higher source-build coupling rates (Observation 5), migration from Ant to Maven tends to have little impact on churn rate or source-build coupling. In the projects that have migrated, the median monthly churn rate and source-build coupling rate of Maven is almost identical to those of Ant (Figures 4.15a and 4.15b). Also contrary to Observation 5, we find that the build author ratio tends to drop as projects migrate from Ant to Maven (Figure 4.15c). Wilcoxon signed rank tests of build author ratio confirm that the results are significant, while churn rate and source-build coupling results are inconclusive.

When projects migrate from `make` or Autotools to CMake, the source-build coupling also tends to decrease, implying that a migration to CMake eases the burden of build maintenance. Similar to Maven, Figure 4.15c indicates that teams tend to adopt a more concentrated build maintenance style after migrating to CMake. Wilcoxon signed rank

tests confirm that the decreases in source-build coupling and build author ratios are statistically significant.

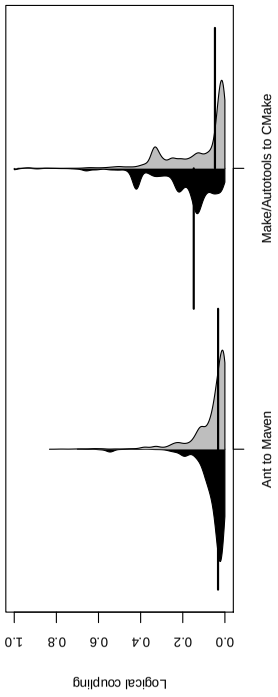
Complementing our software forge findings, Figure 4.16a shows that the median monthly churn rate in the studied ecosystems is rarely impacted by migration projects. Figure 4.16b shows that again source-build coupling tends to drop after a migration to CMake, however is rarely impacted by migration to Maven. Wilcoxon signed rank tests confirm that the CMake migration results in Debian and GNU ecosystems are statistically significant, however the Maven results in Apache are inconclusive. The Wilcoxon signed rank tests also indicate that drops in build author ratios in the studied ecosystems are statistically significant.

### Migration in large-scale projects

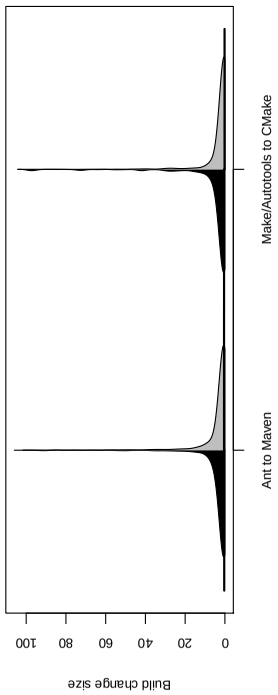
In our study of software forges and ecosystems, we find that build author ratios and source-build coupling tend to decrease. This suggests that technology migration is typically accompanied by a shift of build maintenance from developers to a more specialized build maintenance team. Fewer developers are responsible for build maintenance, freeing them up to focus on making source code changes.

It is unclear whether the decrease in source-build coupling and increase in build team specialization are the result of the migration, perhaps due to the awareness of build maintenance issues raised during migration, or simply due to the trends that we observed as a project ages (Observation 7). To investigate this, we performed a longitudinal study of the large-scale migration from Autotools to CMake in KDE.

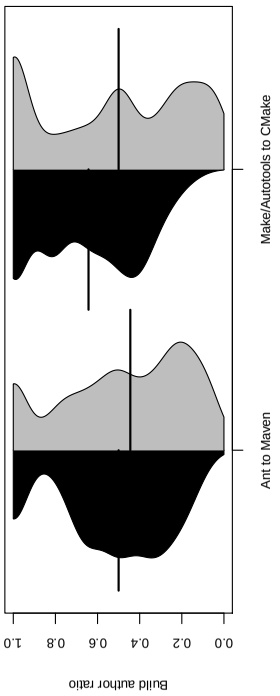
Coupling trends for KDE in Figure 4.12 are decreasing for both Autotools and CMake.




(b) Logical coupling.

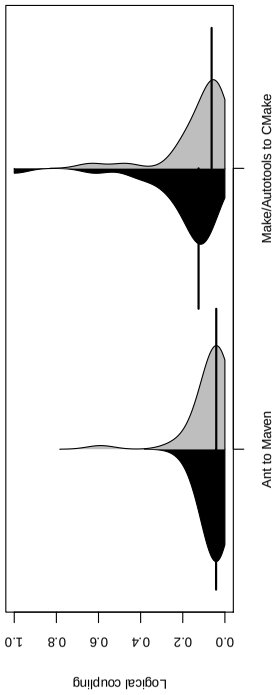



(a) Build churn rates.



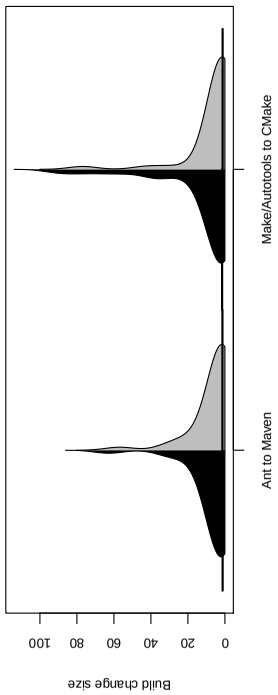

(c) Build author ratio.

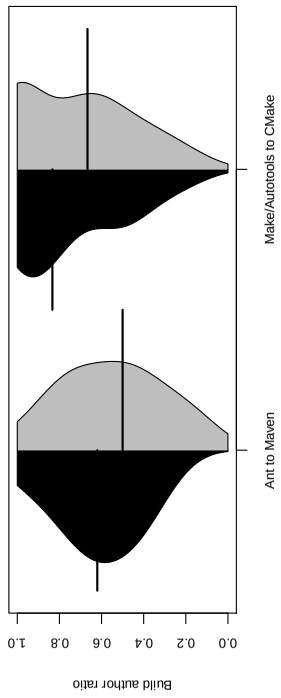
Figure 4.15: [Empirical Study 1] Build technology migration in the studied forges




(a) Build churn rates.

(b) Logical coupling.




(c) Build author ratio.

Figure 4.16: [Empirical Study 1] Build technology migration in the studied ecosystems

After the early development periods in 1997, Autotools follows a slowly decreasing coupling trend from 1998-2004. In 2005, the coupling slowly rises back to 0.1 again, suggesting that it has stabilized. The appearance of the black line in late 2004 indicates that implementation of the new KDE CMake build system has begun. From late 2004 to early 2006, the experimental KDE CMake build system rarely requires coupled changes with the source code, since the Autotools build system is still the official one. The switchover period when the CMake build system became the official one is indicated by the steep slope upwards in CMake and downwards in Autotools in early 2006. There is a brief period when the coupling trend is increasing until it peaks at 0.15 in 2007, but after this the trend begins decreasing again, dipping as low as 0.05 in 2011. The trend does increase again near the end of 2011, which coincides with the KDE team preparing for their 4.8 release. As the KDE project entered 2012, the coupling dropped again to 0.06. The CMake migration has reduced the source-build coupling from a roughly stable 0.1 to 0.05.

Figure 4.12 shows decreasing trends in the KDE build author ratio for both Autotools and CMake build systems. After an early period of dispersed changes, and a trend of growth from 1998 to 1999, a decreasing trend in Autotools authorship begins in 2000. In 2004, the KDE Autotools trend levels off at roughly 50%. After an early growth period in 2007, the KDE CMake authorship drops as low as 24%.

*While changes in monthly build churn rates and source-build coupling prior to and post-migration were inconclusive at times, build author ratio tends to decrease, indicating that more specialized build maintenance teams tend to emerge when performing migrations. The dedication of build experts that we observe during build technology migration can defer build maintenance to a dedicated team, which may help reduce the impact of build maintenance that other software developers must pay.*

## 4.6 Threats to Validity

We now discuss the threats to the validity of our empirical study.

### 4.6.1 Construct Validity

We assume that developers submit related changes using one commit, although our prior work has shown that this may not always be the case [70]. There is a well-documented lack of well-linked data [15, 88] that prevents us from grouping related commits together. Regardless, our analysis draws on comparisons among repositories, not on the absolute values of the metrics.

Our authorship and change analyses rely on the commit data that is recorded in the studied Git repositories. Git repository data may have been imported from other VCS tools that do not: (a) track atomic commits (e.g., CVS), or (b) differentiate between committers and authors (e.g., SVN). In such cases, we rely on the heuristics that are used to recover that information by Git import tools. For example, atomic commits may be approximated using the sliding time window approach [114], which considers all commits that are recorded by one author within a time window (e.g., 300 seconds) as one atomic commit.

Abstraction-based technologies are used to generate low-level specifications. We assume that developers do not commit the generated files, and that projects with commits containing low-level specifications prepared the changes by hand. This assumption may not always hold, creating noise in our dataset. However, if this noise were

heavily influencing our conclusions, we would expect inflated results from the low-level technologies, while we observe that framework-based and abstraction-based technologies tend to induce more build maintenance activity.

### 4.6.2 Internal Validity

We assert that by studying varying trends in the recorded version history of projects using different build technologies, we measure characteristics of build maintenance that are build technology-specific. It may be that the phenomena that we observe are a property of the development cultures of the studied hosts. It may also be that the observations are purely coincidental. However, the large-scale nature of our study of 177,039 repositories spread across four software forges, three software ecosystems, and four large-scale projects, as well as the consistency of our observations across this dataset reduces the likelihood that our observations are purely coincidental.

Counting the number of changes (or the number of lines changed) may not truly reflect the complexity of those changes. For example, while more numerous, Maven changes may be trivial to implement when compared to `make` changes. Moreover, the reliability of the build system may also impact not only the build maintenance effort, but also the overall development as well. For example, `make`-based build systems may be more prone to dependency errors, whereas modern tools automate much of the internal dependency management. As a result, broken builds and other build-related problems may occur more frequently and/or may cause more damage (by slowing build-related feedback for development teams) using traditional `make`-based systems. We plan to investigate these and other topics in future work.

### 4.6.3 Reliability Validity

We use a modified version of the Github Linguist tool[14] to conservatively classify files as source or build files. We have made our extended version available online.[15] While our classification tool is lightweight enough to iterate over all of the changes in our large corpus, we may miss files that are build or source related that do not conform to filename conventions.

### 4.6.4 External Validity

Although we study a large corpus of 177,039 repositories, we focus on a limited number of forges, ecosystems, and projects. Also, we only study open source repositories. As such, our results may not generalize to other open source or proprietary repository hosts. We plan to address this in future work.

There are hundreds of build technologies, and of these, we selected a small subset for study. Our findings are entirely bound to the studied technologies. However, the build technologies that we selected for study cover a considerable portion of the repositories in the corpus.

## 4.7 Chapter Summary

Build systems enable modern development practices such as continuous integration and continuous delivery. However, they require a substantial investment of maintenance effort to remain correct as source files, features, and supported platforms are

[14] https://github.com/github/linguist/
[15] http://sailhome.cs.queensu.ca/replication/shane/PhD/

added and removed. Build maintenance is a nuisance for practitioners, who often refer to it as a "tax."

A wide variety of technologies are available to enable development teams to implement build systems.[16] Although it is of paramount importance for researchers and tool developers, little is known about which build technologies are broadly adopted and whether technology choice is associated with build maintenance activity.

In this chapter, we study the relationship between build technology selection and build maintenance to help practitioners make more informed build technology choices and narrow the scope of future research. Specifically, we set out to address the following question:

**Central Question:** *Is there a relationship between build technology choice and build maintenance activity?*

In performing a large-scale study of 177,039 open source repositories spread across four forges, three ecosystems, and four large projects, we make the following observations according to three themes of study:

**Build Technology Adoption:** Although many projects continue to use traditional technologies like `make`, language-specific technologies like Rake have recently surpassed them in terms of market share. Furthermore, there is indeed a strong relationship between the programming languages used to implement a system and the build technology used to assemble it. Although researchers and service providers should continue to focus on older build technologies like `make` that still account for a large portion of the market share, more modern build technologies are beginning to gain popularity and should also be considered for study.

[16] http://en.wikipedia.org/wiki/List_of_build_automation_software

Knowing this, development service providers can tailor their solutions to fit their target development demographic more appropriately. For example, cloud-based build infrastructure service providers like Travis-CI[17] can tailor their solutions to provide "first-class" service for the more popular, language-specific build technologies in order to stay ahead of the trend.

**Build Maintenance:** Surprisingly, we find that the modern, framework-driven and dependency management technologies tend to induce more churn and be more tightly coupled to source code than low-level and abstraction-based technologies do. Furthermore, we find that much of the Java and Ruby build maintenance effort is spent on external rather than internal dependency management. Yet, irrespective of technology choice, as projects age, the source-build coupling tends to decrease and they tend to adopt a concentrated build maintenance style.

There appear to be additional maintenance activities associated with more modern build technologies, suggesting that while they provide additional features, there is a risk associated with adopting them that development teams should be aware of. Likely due to an inflated source-build coupling rate, changes to framework-driven technologies tend to be more evenly dispersed among developers. Development teams should consider whether this wide dispersion of build changes among the team is an appropriate fit for their development process.

**Build Technology Migration:** Most build technology migration projects successfully reduce the impact that build maintenance has on developers by shifting build maintenance work from typical developers onto a smaller, dedicated team of build maintainers.

[17] http://travis-ci.org/

### 4.7.1 Concluding Remarks

The focus of this chapter is on build maintenance from a "macro" perspective without studying the contents of the build specifications. Just as source code files can contain quality issues, we suspect that build specifications may also suffer from similar quality issues. To that end, in the next chapter, we set out to study *duplication* (a.k.a., cloning), a common source code anti-pattern, in build specifications.

CHAPTER 5

# Cloning in Build Specifications

CENTRAL QUESTION

? *How much cloning is typical of build systems? How can cloning be avoided?*

**An earlier version of the work in this chapter appears in** Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Software Engineering In Practice track (SEIP) [73]

## 5.1 Introduction

To reap the most benefit, build systems must be carefully maintained to ensure that deliverables are assembled correctly. Since build systems tend to grow in terms of size and complexity as they age [4, 67], they also tend to become more difficult to maintain. Indeed, as Munich Re (a large reinsurance company) has shortened development cycles to yield more frequent releases, maintainers have noticed that change requests for

the build system have increased in frequency and difficulty. The increased cost of build maintenance motivated management to contact CQSE (a software quality consultancy group) to investigate the root cause and propose methods of reducing the cost of build maintenance.

Through assessment of the Munich Re build system, we note that cloning (i.e., duplication) of build logic can contribute to an increase in frequency and difficulty of build maintenance. Munich Re maintains roughly 30 custom business information systems implemented using C#, which share a common build system that exploits similarities among the applications. However, over the years, the build system has grown to roughly 1.1 million lines of build logic. Maintenance of the build system has been subcontracted to an external supplier who has allocated a team of three developers to the task. Since clones are scattered throughout the build system, build changes often need to be repeated in as many as 30 locations. Defects may linger in the build system if changes are not propagated to all of the required clones.

Despite the perils of build logic cloning, it is not well understood. Hence, although build maintainers tend to agree that cloning is problematic, selecting a more maintainable solution is non-trivial. For example, it is not clear whether build logic cloning can be avoided, i.e., cloning may be an innate property of build systems. Moreover, it may be that certain technologies are more prone to cloning, which would suggest that migration to a less clone-prone technology could provide some relief. We, therefore, set out to address the following question:

**Central Question:** *How much cloning is typical of build systems? How can cloning be avoided?*

To that end, we collect and analyze a benchmark comprising 3,872 open source

build systems from Apache, GNU, Sourceforge, and Github. Through analysis of the benchmark, we address five research questions and two themes:

### 5.1.1 Deriving Baseline Values

In order to ground a notion of build cloning rates empirically, we *quantitatively* analyze the benchmark, addressing the following three research questions:

(RQ1) *How much cloning is typical of build systems?*

**Motivation:** Little is known about build logic cloning. Hence, we are interested in first exploring what typical cloning rates are within the scope of build systems.

**Results:** Although cloning rates in build systems are typically higher than those of other software artifacts, there are build systems with little cloning, indicating that there are measures one can take to reduce build logic cloning.

(RQ2) *Does technology choice influence cloning in build systems?*

**Motivation:** There are numerous build technologies, each with its own nuances. A better understanding of the influence that technology choice has on build system quality metrics like cloning will allow practitioners to make more informed build technology choices.

**Results:** The more recent CMake (C/C++) and Maven (Java) technologies tend to be more prone to cloning than the older Autotools (C/C++) and Ant (Java) ones.

(RQ3) *Do benchmark-derived cloning thresholds vary among build technologies?*

**Motivation:** If technology-specific cloning benchmarks vary considerably, a single technology-independent benchmark would set a target that is unreasonably

low for clone-prone technologies, and too lax for clone-resistant technologies.

**Results:** We use thresholds derived from quantiles in our benchmark to identify build systems with abnormal cloning characteristics. Technology-specific thresholds vary most for Java build systems with abnormally low amounts of cloning, and between CMake/-Autotools and Ant/Maven for build systems with abnormally high amounts of cloning.

### 5.1.2 Understanding Cloned Information

The build system describes up to five interdependent steps (*cf.* Chapter 2). Build specifications describe how each step must be performed. It is not clear which of these steps are most susceptible to cloning. Through *qualitative* inspection of build logic clones, we address the following two research questions:

(RQ4) *What type of information is typically cloned in build specifications?*

**Motivation:** We set out to better understand what steps of the build process tend to be cloned in each build technology with the intent to discover if cloning rates are affected by limitations of the technology itself or a lack of skill in applying it.

**Results:** The more recent technologies are more susceptible to cloning of configuration details like API dependencies, while the older technologies are more susceptible to cloning of lower-level build logic.

(RQ5) *How do build systems with few clones achieve low clone rates?*

**Motivation:** We compare clone-prone and clone-resistant build systems to elucidate differences in cloning practices.

**Results:** Build systems with little cloning leverage reuse mechanisms beyond those offered by build technologies themselves, suggesting that existing reuse mechanisms offered by build technologies are insufficient for avoiding build logic cloning.

**Chapter organization.** The remainder of this chapter is organized as follows. Section 5.2 provides background detail and definitions used throughout the chapter. Section 5.3 describes the case of build logic cloning at Munich Re. Section 5.4 describes the design of our empirical study. Sections 5.5 and 5.6 present our findings with respect to our five research questions. Section 5.7 discloses the threats to the validity of our empirical study. Finally, Section 5.8 draws conclusions.

## 5.2 Background and Definitions

Clones are duplicated regions in software artifacts, typically created by copying and pasting. Clones tend to hinder maintenance, since changes to an artifact region often need to be performed consistently to all of its clones. Clone detection tools search for clones in software artifacts to support the maintenance of software artifacts that contain clones.

There are various types of clones used in research and practice [57, 94]. To the best of our knowledge, this is the first study to explore build logic cloning. Hence, for our measurements, we focus on *Type I clones*, i.e., exact copies ignoring the variations in whitespace and comments, and leave the exploration of higher level clone types to future work. We measure the extent of build logic cloning using:

**Clone Coverage** — The proportion of build logic lines that are cloned at least once

in the build system. Values range between 0 (i.e., no detected clones) and 1 (i.e., each build logic line is cloned at least once).

**Blow Up** — The degree of inflation in build system size with respect to a hypothetical build system that does not contain any clones, i.e., $\frac{ActualSize}{ReduncancyFreeSize} - 1$. Hence, a blow up value of 0 indicates that the system is not inflated by cloning, while values above 0 indicate the degree of inflation due to cloning.

## 5.3 Build Logic Cloning in Industry

This section provides a motivational example to illustrate the reasons and impact of excessive build logic cloning. First, however, we provide a brief background on cloning, and the metrics that we use to measure its extent.

### 5.3.1 Clone-Based Build System Design

Most Munich Re business applications use a shared company-wide build infrastructure based on Microsoft Team Foundation Server (TFS) specified using MSBuild. Each business application has different build specifications for each build configuration (e.g., debug and release) and each application release (e.g., 2013.1 and 2013.2). For example, one business application has six build specifications representing debug and release configurations for its 2013.1, 2013.2, and 2013.3 releases.

These MSBuild specifications enhance the default TFS build process with unit testing, continuous code quality analysis, and packaging in preparation for automated deployment to testing, pre-production, and production environments. Build specifications range between 1,500-8,000 lines of build logic per file, with an average size of

3,800. The Munich Re build system currently contains more than 1.1 million lines of build logic spread across 295 build specifications.

To add a new release or a new application to the build system, the build specifications of a stable application are duplicated and customized. In the simplest case, the application name, as well as the application-specific directories and source file lists need to be customized. More complex applications have unique packaging requirements or need special interaction with the TFS. Yet, since the core build logic remains unchanged, build specifications are largely the same. Since new releases and new applications must be added to the build system regularly, one can easily see how the Munich Re build system has grown to the size it is today.

### 5.3.2 Clone-Based Build System Maintenance

The effort required to maintain the Munich Re build system has steadily increased over the years. It now requires three full-time employees whose sole responsibility is to maintain the build system. These build maintainers are responsible for configuring new application releases, adding new applications to the build system, fixing build system defects, and adding new build system features.

Even with this dedicated team of build maintainers, defects fixes and new features take a long time to complete. In fact, due to time pressure, some build system changes are never completely propagated to all build specifications. For example, a build maintainer recently added the `ContinueOnError` flag (which prevents the build from failing) to one of three specifications that uninstall the same application. It was not until one week later that the flag was applied to the second of the three specifications. The flag has not yet been applied to a third instance.

Prolonged fixes and inconsistent changes are an often-observed clone-related problem in source code, too [48]. It is a novel observation, however, that build specifications are affected by these problems as well.

### 5.3.3 Shortcomings of the Clone-Based Build System Design

The clone-based build system design has been perceived by build maintainers as one of the fundamental causes of the build maintenance difficulties at Munich Re. Through discussions with the build maintainers, they report that: "You have to alter 15 occurrences [of a defect] and you have to be really careful not to introduce new [defects]" and "With over 270 or more versions [of a build specification], the [build system] is simply not maintainable anymore."

Indeed, with a minimum clone length of twenty lines, clone detection results indicate that the Munich Re build system has a clone coverage of 94.4% and a blow up of 1,023% (clone detector configuration details are found in Section 5.4). With a minimum clone length of five lines, clone coverage and blow up values increase to 99.1% and 2,335% respectively. In other words, the Munich Re build system: (1) is over 23 times larger than it would be without cloning, and (2) only contains roughly 50,000 unique lines of build logic.

Cloning between build specifications has also lead to dead build features. These features were copied when a specification was duplicated, but are not used during the build. This further inflates maintenance effort and increases the likelihood of introducing defects during maintenance, since one must first recognize whether a build feature is active or not before making modifications.

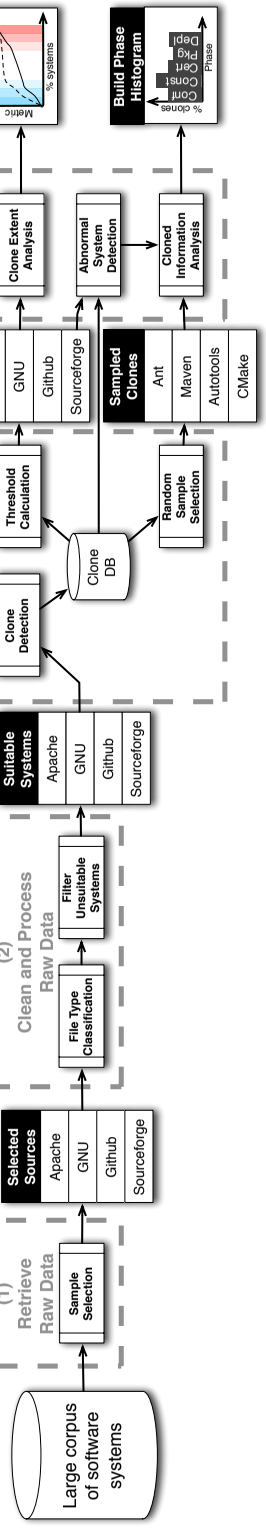



Figure 5.1: [Empirical Study 2] Overview of our data extraction and analysis approach.

Table 5.1: [Empirical Study 2] Overview of the studied systems.

| | Ant | Maven | Autotools | CMake | Total |
|---|---|---|---|---|---|
| **Apache** | 51 | 56 | 18 | 3 | 128 |
| **Github** | 114 | 321 | 521 | 220 | 1,176 |
| **GNU** | 15 | 0 | 243 | 12 | 270 |
| **Sourceforge** | 593 | 125 | 1,517 | 63 | 2,298 |
| **Total** | 773 | 502 | 2,299 | 298 | 3,872 |
| **# w/ Clones** | 664 | 484 | 943 | 162 | 2,253 |
| **% w/ Clones** | 86% | 96% | 41% | 54% | 58% |

## 5.4 Empirical Study Design

In this section, we describe our benchmark collection and analysis approach. Similar to Chapter 4, our approach to extracting and analyzing the build logic cloning benchmark is structured using the four steps suggested by Mockus for analyzing software repositories [76]. Figure 5.1 provides an overview of our approach. We describe each step in the approach below.

### 5.4.1 Retrieve Raw Data

It is important that our benchmark contains a large sample of build systems in order to improve confidence in the conclusions that we draw. Hence, we select a sample of 3,872 build systems from the large corpus of open source systems of varying size, scope, and domain collected by Mockus [77]. We describe the corpus of build systems used in this study and explain our extraction and analysis approaches below.

#### 5.4.1.1 Sample selection

The sample of build systems was obtained from four sources described in Table 5.1. The *Apache Software Foundation* provides organizational, legal, and financial support for a broad range of open source software systems. Savannah (*GNU*) is the software forge for people committed to free software. *Github* and *Sourceforge* are also popular software forges.

We select the build systems of Java and C/C++ systems for our benchmark, since they are among the most broadly adopted programming languages in our corpus [77]. We further narrow our study by selecting the two most frequently used build technologies for each studied language. As shown in Chapter 4, in our corpus, C/C++ systems use GNU Autotools and CMake most frequently, and Ant and Maven are used most frequently to specify Java build systems. We extract the latest version of each software system that meets our selection criteria from the large corpus.

The GNU Autotools and CMake technologies are abstraction-based (*cf.* Chapter 4), and thus, are used to generate low-level build specifications (i.e., `Makefiles`). We configure our clone detection tool to scan the high-level abstractions (e.g., `configure.ac`, `Makefile.am`), rather than the automatically generated build specifications.

Figure 5.2 provides an overview of the benchmark by plotting the number of clones detected against size of the build system using hexbin plots [19]. Hexbin plots are scatterplots that represent several data points with hexagon-shaped bins. The darker the shade of the hexagon, the more data points that fall within the bin. The plot is logarithmically scaled in all dimensions to lessen the influence of outliers.

The relationship between number of clones and build system size is roughly linear on the log scale and quadratic on the linear scale. The hexagons in Figure 5.2 tend to

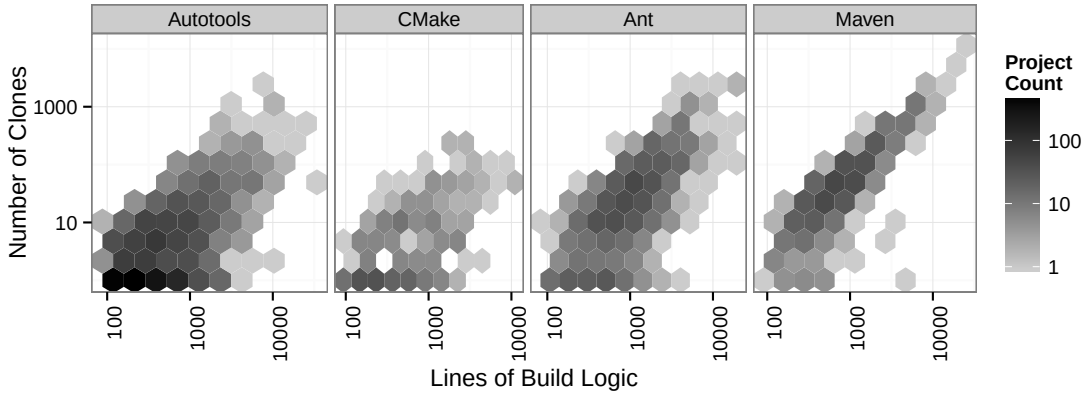



Figure 5.2: [Empirical Study 2] Number of clones detected vs. build system size (in lines of build logic).

appear in a positive upward diagonal direction. Similarly, the hexagons tend to deepen in shade along an upward diagonal trend in the Java build systems. This suggests that as a build system grows, so too does its proneness to cloning.

## 5.4.2 Clean and Process Raw Data

Prior to addressing our research questions, we must first ensure the extracted systems are suitable for analysis. This process is divided into two steps.

### 5.4.2.1 File type classification

Again, similar to Chapter 4, we cannot apply the semi-automatic file type classification of our prior work due to the large-scale nature of this corpus [70]. To address this, we conservatively identify build files based on filename conventions (*cf.* Table 4.2). Although our approach may miss some build specifications that do not follow filename conventions, the approach is lightweight enough to be applied to all files in the corpus.

#### 5.4.2.2 Filter unsuitable systems

Software incubators such as Github and Sourceforge often contain systems that have not yet reached maturity. Neitsch *et al.* conjecture that IDE support for building software is sufficient for small systems [86]. Indeed, Smith suggests that build system maintenance does not become a problem until a system ages, requiring more configurability to expand market presence [99]. To reduce noise in the benchmark, we filter away systems with fewer than five build specification files or 100 lines of build logic.

### 5.4.3 Construct Meaningful Measures

Next, we apply clone detection to the surviving build systems using ConQAT [27]. Then, metric thresholds are derived from the benchmark. Finally, a random sample of clones are selected for detailed analysis.

#### 5.4.3.1 Clone detection

The ConQAT clone detector reads all files of a system that match the pattern of the specified build technology from Table 4.2 into memory. The detection algorithm is configured to be line-based with varying minimum clone lengths of 5, 10, 15, and 20 lines. To handle file formatting differences, we trim the leading and trailing white space of each line. We omit empty lines and comments, since they do not have an impact on the build process. We also omit closing XML tags, since XML-based build specifications are more verbose. Although not strictly necessary for our analyses in this chapter, controlling for XML verbosity helps to make XML and non-XML build logic cloning results more comparable. In this study, we consider only Type I clones. For example, when the minimum clone length is set to five, clones must share at least five consecutive

non-empty lines after applying the normalization described above.

#### 5.4.3.2 Threshold calculation

Thresholds are used to identify entities with metric values that warrant further investigation. For example, build specifications with a blow up value above two may be worth inspection. Yet it is non-trivial to select effective thresholds that pinpoint abnormal entities while retaining low false positive and false negative rates. There are various threshold derivation techniques that can gauge a variable with unknown properties empirically. In order to address RQ3, we adopt the quantile-based technique suggested by Alves *et al.* [9], since (as they point out) other threshold derivation techniques (such as deviation analysis) often make invalid assumptions about the dataset (e.g., normally distributed), or require carefully tuned input parameters (e.g., number of clusters for clustering techniques).

Alves *et al.* suggest that values that fall above the $70^{th}$, $80^{th}$, and $90^{th}$ percentiles are abnormal to a moderately high, high, and very high degree respectively. We extend this concept by arguing that values that fall below the $30^{th}$, $20^{th}$, and $10^{th}$ percentiles are abnormal to a moderately low, low, and very low degree respectively. Values that appear at quantile boundaries are considered thresholds.

#### 5.4.3.3 Random sample selection

To address RQ4, we need to select a representative sample of clones of each studied technology for deeper analysis. We randomly select a sample of clones large enough to achieve a $95\%$ confidence level and a 5% confidence interval.

### 5.4.4 Analyze and Present Results

Finally, we use the derived thresholds to detect and analyze build systems with abnormal amounts of cloning.

#### 5.4.4.1 Clone extent analysis

We use quantile plots to indicate whether the amount of cloning in a system is abnormal. These plots show the cumulative proportion of systems that have clone coverage and blow up metrics below a given value.

#### 5.4.4.2 Abnormal system detection

To better understand good and bad cloning practices, we analyze the most and the least clone-prone systems. We first identify common cloning pitfalls of the most clone-prone systems. Then, we analyze the least clone-prone systems to understand how these pitfalls can be avoided.

#### 5.4.4.3 Cloned information analysis

We manually analyze the information cloned in a random sample of clones for each studied technology (RQ4), and all of the clones in the highly clone-prone build systems (RQ5). To address RQ4, we assess each clone to determine which of the five build steps (*cf.* Chapter 2) are impacted.

The configuration step can be broken down into three subcategories. *Dependency probing* checks for the existence of an appropriate version of a third-party dependency (e.g., build tools, APIs). *Dependency resolution* probes for, downloads, and deploys third-party dependencies in a local cache in preparation for use in later build steps.

*Tool configuration* selects the necessary options to prepare tools for use in later build steps (e.g., compiler flags).

The construction step comprises two subcategories. *Build* either describes: (1) internal source dependencies (e.g., `foo.o` should be compiled before linking it into `foo.so`), or (2) how input files are translated into output files (e.g., `gcc` should be executed on `foo.c` to produce `foo.o`). *Filesystem* logic handles the creation of output directories, or implements so-called "clean" targets that remove intermediate and output files to force the build system to start from scratch.

The certification step most often comprises *Unit testing* logic that configures, compiles, or executes unit tests. Similarly, *Packaging* logic describes how deliverables should be bundled together for end user consumption.

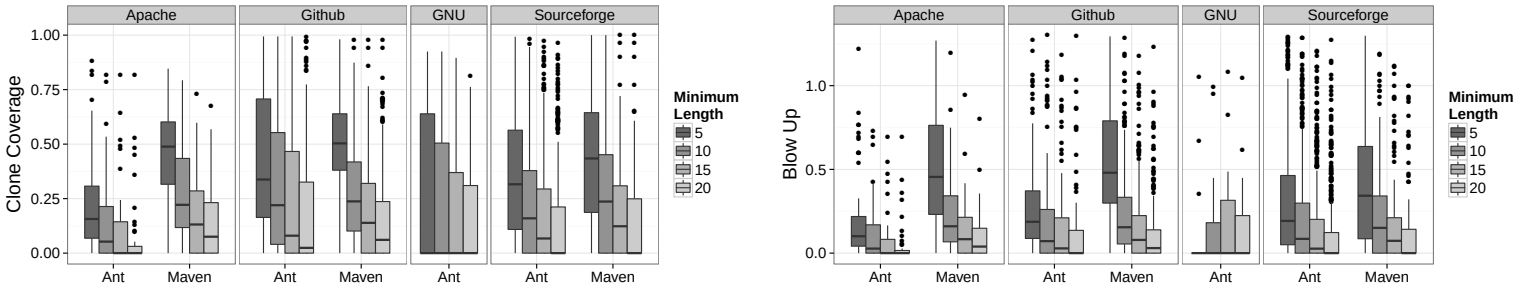
The deployment step not only comprises *Installation* logic that describes how deliverables are deployed on a target machine, but also *Execution* logic that describes how deployed deliverables should be executed in testing environments.

## 5.5 Deriving Baseline Values

In order to ground our intuition about the extent of build cloning, we perform a quantitative analysis of the benchmark with respect to RQ1-RQ3.

### (RQ1) How much cloning is typical of build systems?

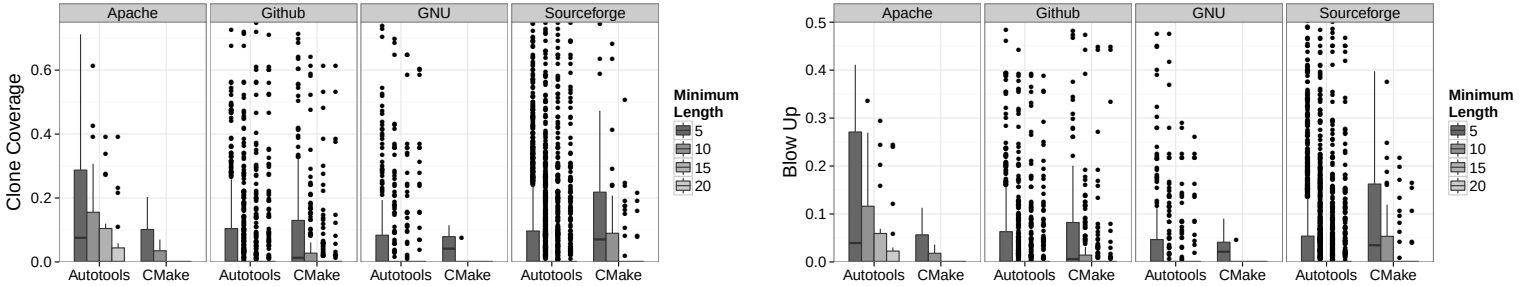
In order to address RQ1, we analyze the distributions of clone coverage and blow up in the benchmark using boxplots.




(a) Clone coverage (Java)




(b) Blow up (Java)




(c) Clone coverage (C/C++)




(d) Blow up (C/C++)

Figure 5.3: [Empirical Study 2] Cloning metrics gathered from the studied systems. Note: scales differ among the plots.

**In general, build logic clones tend to be small.** Figure 5.3 shows that clone coverage and blow up values decrease drastically when the minimum clone length is set to ten or higher, indicating that many of build specification clones cover five to nine lines. This is consistent with clones in other software artifacts, where short clones are also more frequent than long ones [47, 49].

Manual analysis of randomly selected clones with a minimum length of five reveals few false positives. Hence, to simplify the remaining analyses, we only discuss the results with respect to a minimum length of five.

**Cloning is more prevalent in Java build systems than many other software artifacts.** Although it is not abnormal for legacy COBOL systems to have cloning rates of 80% [79], prior work shows that many large software systems contain 7%-23% duplicated code [11, 56, 60], with rare cases reaching 59% [31]. Requirements documents have an average clone coverage of 14%, with one reported case of 72% [47]. Conversely, our benchmark values indicate that a clone coverage of 50% occurs rather frequently for Java build systems. Figure 5.3a shows that the studied Maven build systems have a median clone coverage ranging between 47%-50%. While Ant build systems have medians below 50%, the top of the box (indicating the $75^{th}$ percentile) extends beyond 50% for Github, GNU, and Sourceforge build systems, indicating that more than one quarter of Ant build systems have clone coverage values that exceed 50%.

**On the other hand, cloning in C/C++ build systems is less prevalent.** Figure 5.3c shows that the median clone coverage for Autotools build systems only exceeds 0 in the Apache organization, indicating that half of the studied Autotools build systems in the Github, GNU, and Sourceforge organizations do not contain any clones. In fact, Table 5.1 shows that while 86%-96% of the studied Java build systems contain clones,

only 41%-54% of C/C++ build systems do. Furthermore, Figure 5.3c shows that $75^{th}$ percentile of C/C++ build systems does not exceed 30% clone coverage.

While the magnitude of the observed C/C++ build clone coverage values pale in comparison to the observed Java ones, there are still many C/C++ build systems that have plenty of clones. For example, Figure 5.3c shows that 25% of Autotools build systems in Apache have a clone coverage between 27%-66%. In addition, 25% of CMake build systems in Sourceforge have a clone coverage between 21%-48%.

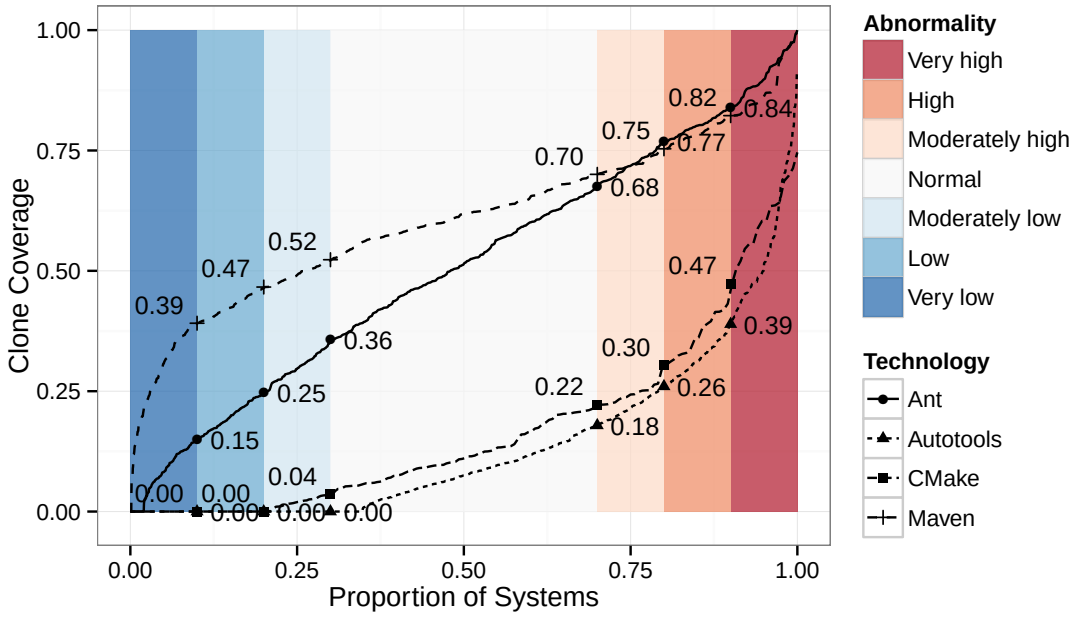
*Java build systems often have 50% clone coverage, rates that have only been observed in legacy systems or in extreme cases when studying other software artifacts. While cloning in C/C++ build systems is less pervasive, there are still several systems that have a substantial number of clones.*

## (RQ2) Does technology choice influence cloning in build systems?

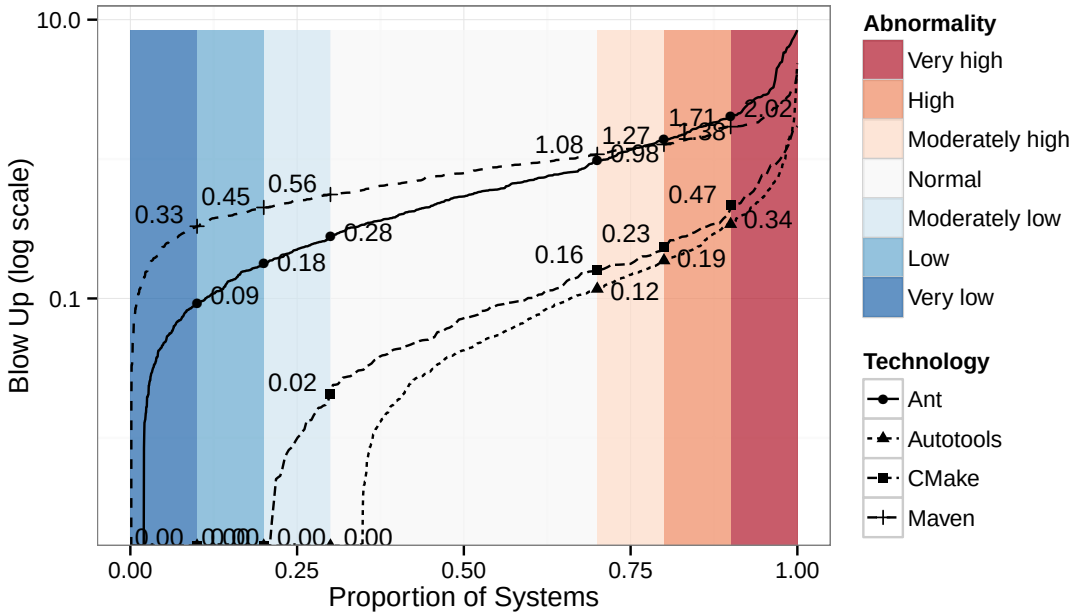
To address RQ2, we compare the distributions of Figure 5.3.

**For Java systems, cloning is more prominent when using the more recent Maven technology than Ant.** Figures 5.3a and 5.3b show that Maven build systems tend to have higher clone coverage and blow up values than Ant ones do. Mann-Whitney U-tests (an alternative to the Student t-test with greater resiliency to non-normal distributions) confirm that the differences in clone coverage and blow up are statistically significant ($p < 0.01$) in Apache, Github, and Sourceforge. Table 5.1 shows that none of the studied GNU systems use Maven, so no comparison can be made.

**For C/C++ systems, cloning is more prominent when using the more recent CMake build technology than Autotools.** Although Figure 5.2 indicates that there are more clones in Autotools than CMake build systems, clone coverage and blow up statistics tend to favour CMake. First, Figure 5.3c shows that clones tend to cover more




(a) Clone coverage




(b) Blow up

Figure 5.4: [Empirical Study 2] Quantile plots of system-level cloning metrics.

CMake lines than Autotools ones do. Second, Figure 5.3d shows that CMake clones tend to inflate build systems more than Autotools clones do. Indeed, with a minimum clone length of five, the median clone coverage and blow up of CMake exceeds that of Autotools in Github, GNU, and Sourceforge.

Mann-Whitney U-tests confirm that the differences in clone coverage and blow up are statistically significant ($p < 0.01$) in GNU and Sourceforge, however they cannot confirm a statistically significant difference in Github ($p = 0.06$). Furthermore, although the median for Autotools build systems exceeds that of CMake in Apache, Table 5.1 shows that our sample of three CMake systems in Apache is too small for statistical comparisons. In general, CMake build systems tend to be covered and inflated more by cloning than Autotools ones are.

*The more recent CMake (C/C++) and Maven (Java) build technologies tend to be more prone to cloning than the older Autotools (C/C++) and Ant (Java) ones are.*

## (RQ3) Do benchmark-derived cloning thresholds vary among build technologies?

To address RQ3, Figure 5.4 shows the clone coverage and blow up quantile plots derived from our benchmark. We discuss the differences in thresholds for the studied Java and C/C++ technologies below.

**Maven build systems have much higher thresholds for low values than Ant ones do.** Complementing our RQ2 findings, Figure 5.4 shows that normal cloning rates in Maven are higher than those of Ant. In fact, Figure 5.4a shows that Maven build systems with a clone coverage below 52%, 47%, or 39% are considered moderately low to very low in our benchmark. On the other hand, Ant build systems with a clone coverage of

36%, 25%, or 15% are considered low. Similarly, Figure 5.4b shows that blow up values of 56%, 45%, and 33% are also considered low for Maven build systems, while values of 28%, 18%, and 9% are considered low for Ant build systems. In other words, clone coverage and blow up values up to and exceeding the $30^{th}$ percentile of Ant would still be beneath the $10^{th}$ percentile of Maven build systems.

On the other hand, there is very little difference between the low thresholds of C/C++ build systems. Figure 5.4 shows that the CMake and Autotools clone coverage and blow up thresholds differ at most by four percentage points.

**High thresholds are similar between Ant and Maven, and between Autotools and CMake.** Figure 5.4a shows that the high clone coverage thresholds for Java build systems differ by two percentage points at the $70^{th}$, $80^{th}$, and $90^{th}$ percentiles. C/C++ systems differ by four to eight percentage points.

Similarly, Figure 5.4b shows that blow up thresholds at the $70^{th}$, $80^{th}$, and $90^{th}$ percentiles of the C/C++ build systems differ by four to seven percentage points. However, blow up thresholds of the studied Java build systems cover a broader range of 10 to 31 percentage points. The largest difference in blow up thresholds for Java build systems (31 percentage points) is at the $90^{th}$ percentile, and is likely due to extreme blow up values in outlier systems.

**Munich Re build system is indeed unusual.** Regardless of the technology, clone coverage or blow up values of the same magnitude as Munich Re are not observed beneath the $90^{th}$ percentile. Hence, our intuition about the Munich Re build system is empirically confirmed by the benchmark.

*Technology-specific thresholds vary most for Java build systems with abnormally low amounts of cloning, and between CMake/Autotools and Ant/Maven for build systems with abnormally high amounts of cloning.*

## 5.6 Understanding Cloned Information

While our quantitative analysis of Section 5.5 can be used to identify build systems with abnormal amounts of cloning, it does not help us to understand how cloning can be avoided (without migrating to a different technology). To address this, we perform a qualitative inspection of build logic clones. In this section, we present the results of this analysis with respect to RQ4 and RQ5.

### (RQ4) What type of information is typically cloned in build specifications?

In order to address RQ4, we need to analyze a representative sample of clones in each studied technology. As the total number of clones is very large, and there was no automatic means of determining the build step of the clone, we sampled the clones for manual inspection. We obtain the proportion estimates that are within 5% bounds of the actual proportion with 95% confidence level using the sample size calculation of $s = \frac{z^2 p(1-p)}{0.05^2}$, where $p$ is the proportion we want to estimate and $z = 1.96$. Since we did not know the proportion in advance, we use $p = 0.5$. We, further, correct for the finite population of clones to obtain 382 Ant, 382 Maven, 379 Autotools, and 349 CMake clones. Table 5.2 shows the proportion of randomly selected clones that are associated with each build subcategory.

**Cloning focus shifts from construction to configuration in Maven.** Table 5.2 shows that the majority of Ant clones impact the construction step (64%). 47% of clones are associated with the build category and 32% with the filesystem category. The next most frequently cloned step (configuration) appears in half as many clones (32%).

Table 5.2: [Empirical Study 2] A manual analysis of the clones that pertain to each subcategory in a statistically representative subsample (95% confidence level; $\pm 5\%$ confidence interval). Phase totals are not the sum of each subcategory because a clone may pertain to many subcategories.

| **Clone Counts** | | **Ant** | **Maven** | **Autotools** | **CMake** |
|---|---|---|---|---|---|
| All clones | | 56,521 | 71,543 | 23,723 | 3,746 |
| Sample size ($95\% \pm 5\%$) | | 382 | 382 | 378 | 349 |
| **Phase** | **Subcategory** | **Ant** | **Maven** | **Autotools** | **CMake** |
| Config. | Deps. Probing | 3% | 0% | 1% | 8% |
| | Deps. Resolution | 1% | 54% | 0% | 4% |
| | Tool Configuration | 29% | 32% | 21% | 32% |
| | **Phase Total** | **32%** | **79%** | **22%** | **40%** |
| Cons. | Build | 47% | 16% | 48% | 65% |
| | Filesystem | 32% | 1% | 19% | 1% |
| | **Phase Total** | **64%** | **17%** | **56%** | **66%** |
| Cert. | Unit Testing | 12% | 4% | 13% | 11% |
| Pkg. | Packaging | 25% | 21% | 21% | 2% |
| Depl. | Installation | 8% | 1% | 9% | 7% |
| | Execution | 3% | 0% | 0% | 0% |
| | **Phase Total** | **11%** | **1%** | **9%** | **7%** |

Many of these construction clones replicate entire Ant targets that create or delete temporary output directories or compile Java source code. Since a single invocation of the Java compiler will automatically resolve dependencies between input source files [29], Ant targets that compile Java code often invoke the Java compiler specifying all impacted Java files as inputs using wildcards. Hence, the compile target is generic, and often cloned in several Ant specifications.

While construction accounts for many of the clones in Ant build systems (64%), most Maven clones impact configuration (79%). The next most frequently cloned step (packaging) appears in less than a third as many clones (21%).

We observed that many of the Maven clones replicate third-party dependency lists or plugin configuration among subsystems. While this ensures that each subsystem

can be built independently of the others, it imposes a heavy load on maintainers, who will need to update several `pom.xml` files in order to modify third-party dependency lists or update plugin configurations.

**Construction is the most heavily cloned build step in C/C++ build systems.** Table 5.2 shows that the build subcategory represents 48% and 65% of Autotools and CMake clones respectively. The filesystem category is also detected in 19% of Autotools and 1% of CMake clones. All in all, the construction step accounts for 56% of Autotools clones and 66% of CMake clones.

While the Autotools construction step is most frequently cloned (56%), Table 5.2 shows that packaging details are also cloned often (21%). Yet packaging details are rarely cloned in CMake (2%). Many of these Autotools packaging clones have to do with repetition of data file lists among subsystems. Since Autotools build specifications generate recursive `make` build systems, variables are not shared among subsystems [74]. Although Autotools offers developers an `include` directive, we have observed that rather than place shared variables in a header-like file, developers often clone variables that have a shared scope. Similar to Maven, where dependency lists and plugin configuration were replicated, developers likely duplicate shared variables to facilitate subsystem independence. However, this makes system-wide changes more difficult.

Conversely, we observe that CMake packaging details are rarely cloned. We observe that developers leverage built-in CPack functionality of CMake [65], where packaging details are typically specified in a single location: `CPackConfig.cmake`. This eliminates the need to replicate packaging details in subsystem specifications.

**There are more configuration clones in the more recent build technologies.** Table 5.2 shows that there are more than twice as many configuration clones in Maven

```
<!-- Define references to files containing common targets -->
<!DOCTYPE project [
    <!ENTITY modules-common SYSTEM "../modules-common.ent">
]>

...
<project name="bea" default="all">
    <!-- Include the file containing common targets. -->
    &modules-common;
</project>
```

Listing 5.1: [Empirical Study 2] Using XML entity expansion to import common build code in the Keel system.

build systems (79%) than Ant ones (32%). Similarly, there are almost twice as many configuration clones in CMake (40%) than there are in Autotools (22%). In Section 5.5, we report that these more recent technologies are more prone to cloning (RQ2). The shift of cloning tendencies towards configuration likely contributes to the inflated cloning values we observe in the more recent technologies.

*Configuration details are cloned more often in the more recent CMake and Maven build technologies. For Java build systems, Maven clones favour the configuration step, while Ant clones (and clones in C/C++ builds) favour construction. CMake packaging support (CPack) helps to reduce cloning in the packaging step.*

## (RQ5) How do build systems with few clones achieve low clone rates?

To address RQ5, we analyze clones in the systems with the lowest and highest cloning rates for each studied technology.

**Much Java build cloning can be avoided by exploiting the underlying XML representation.** We observe that entire files are duplicated in the Ant and Maven systems with the highest clone coverage. In these cases, development teams duplicate existing

build specifications to rapidly develop new subsystems. However, developers have referred to maintaining such build systems fraught with clones as a "nightmare" (*cf.* Section 5.3). Defect fixes or updates to dependency lists, tool configuration, and packaging details must be carefully replicated among the clones to ensure that builds continue to assemble deliverables correctly.

On the other hand, we have observed that in addition to abstraction mechanisms provided by Ant and Maven (e.g., the `include` and `import` tasks), XML-based build systems avoid cloning by leveraging the underlying XML representation. In prior work, we note that the JBoss build system leverages XML entity expansion in Ant to implement a framework-driven build system referred to as "buildmagic" [67]. Indeed, Listing 5.1 shows how one can use XML entity expansion to avoid duplicating shared build code in subsystem build specifications. We also find that the Ant test suite includes regression tests to ensure that entity expansion continues to work, suggesting that it is not a workaround, but instead is intentionally supported functionality.

**Many C/C++ build logic clones can be avoided by duplicating templates automatically when building.** Many of the studied C/C++ systems provide development APIs. As such, they ship examples of how to use various API functionality with their deliverables. These examples include accompanying build specifications. However, in the C/C++ systems with the highest clone coverage, we find that many of these example build specifications are file clones of each other, which poses maintainability problems.

One of the studied systems with a low clone coverage avoids these clones by duplicating and specializing template build specification automatically using shell scripts

during the construction step. Using this approach, cloning shared build code in example build specifications can be avoided.

*XML entity expansion can be used to avoid cloning shared build code in Java build systems. Cloning of build logic shipped with API usage examples can be avoided by automatically deriving specifications at build-time.*

## 5.7 Threats to Validity

We now discuss the threats to the validity of our analysis.

### 5.7.1 Construct Validity

Our clone detection tool is configured to only detect Type I (exact) clones. Since we do not detect Type II, III, or IV clones, our cloning results should be interpreted as lower bounds rather than exact values.

Our code detection tool is not to confined to the boundaries of coding constructs within build specifications. As such, not all clones that are detected by our approach are refactorable. To mitigate the noise of non-refactorable clones, we set the minimum clone length to be 5, 10, 15, and 20 lines. Nonetheless, an analysis of build logic clones that is confined to the boundaries of coding constructs would make for interesting future work.

### 5.7.2 Internal Validity

We assume that large values of cloning metrics suggest maintenance problems illustrated in our Munich Re example. Yet, recent research suggests that despite the inherent maintainability issues, cloning may not always be harmful [51, 92]. Nonetheless, we find that developers complained about maintainability problems in the heavily cloned build system at Munich Re, suggesting that excessive cloning makes build system maintenance more difficult. Furthermore, prior work shows that unintentional inconsistent changes do occur in large industrial systems [48].

Similar to Chapter 4, we conservatively detect build specifications using filename conventions. Although our classification tool is lightweight enough to iterate over all files in our large corpus, we may miss files that are build-related that do not conform to filename conventions.

### 5.7.3 External Validity

Although our benchmark covers a large corpus of 3,872 systems, a limited number of open source organizations are covered. As such, our results may not generalize to other open source or even proprietary build systems. However, since any build system needs to implement the steps outlined in Chapter 2, we believe that our benchmark is a sound starting point. We plan to extend our benchmark to include proprietary systems in future work.

There are hundreds of build technologies and of these, we only include four in our benchmark. Our findings are entirely bound to the studied technologies. On the other

hand, Chapter 4 shows that the technologies that we have selected are quite popular in open source organizations. Furthermore, Ant shares many similarities with MSBuild. Specifications for both technologies are expressed using abstract targets and tasks specified in an XML format. Hence, we suspect that the characteristics of cloning we observed in Ant will also appear in MSBuild systems. We plan to inspect this suspicion by expanding the scope of our benchmark to include MSBuild in future work.

## 5.8 Chapter Summary

Build systems play a crucial role in software development. They tend to grow in terms of complexity as a software project ages [4, 67]. When build system complexity grows unwieldy, build maintenance becomes difficult, and development teams refactor build systems to restore order.

In order to determine if and where build refactoring should be applied, CQSE performs quality assessments of build systems. In this chapter, we discuss how a benchmark of build logic clones can empirically ground metrics used in these assessments. Specifically, we focus on the following central questions:

**Central Question:** *How much cloning is typical of build systems? How can cloning be avoided?*

Through analysis of the benchmark of 3,872 open source systems, we make the following observations:

- 50% clone coverage rates, which have only been recorded in rare cases in other software artifacts [31], frequently occur in Java build systems.
- The more recent CMake and Maven build technologies tend to be more prone to

cloning, especially of configuration details like API dependencies, than the older Autotools and Ant technologies respectively.

- While build logic cloning can be difficult to avoid, it is not a necessity, i.e., we have observed build systems with little cloning using each studied technology.

- Templating and inclusion mechanisms beyond those provided by build technologies are employed to reduce build logic cloning, suggesting that the mechanisms provided by build technologies are insufficient.

### 5.8.1 Refactoring to Reduce Cloning at Munich Re

The benchmark-derived thresholds confirm that the clone-based build system design at Munich Re is unusual. Munich Re has decided to restructure the build system. To facilitate this, we are creating reusable build logic components that can be shared among build specifications (without cloning).

The analysis we performed in this chapter helped us in designing the solution. First, to work around the limitations of the MSBuild abstraction mechanisms, we adopt a practice that we observed in C/C++ build systems, where common build logic is stored in a template that is copied and specialized automatically during an initial step in the build process. Second, similar to Maven build systems, our solution divides the core build logic that drives the different build steps into individual plugins that enable automated testing, packaging, and deployment.

However, the new build solution also requires a more structured change process. Since changes to shared build components affect all build specifications that rely on them, they must be more carefully maintained than the prior clone-based solution was.

To this end, Munich Re has created a dedicated test bed in which build component changes can be evaluated before they are deployed to production builds. Furthermore, we are creating a dashboard that displays nightly clone detection results as an early-warning system against proliferation of cloning in the new build system.

### 5.8.2 Concluding Remarks

In Chapter 4 and this one, we have explored the overhead introduced by the maintenance of the build system from a high-level in a large sample of systems. To gain a clearer perspective of the drivers of build co-change, in the next chapter, we analyze a sample of four large software systems in detail.

CHAPTER 6

# Drivers of Build Co-Change

CENTRAL QUESTION

? *Can build changes be fully explained using characteristics of co-changed source and test code files?*

**An earlier version of the work in this chapter appears in** Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014) [68]

## 6.1 Introduction

The complex build systems of large software systems require regular maintenance in order to continue functioning correctly. Our prior work shows that, from release to release, source code and build system tend to *co-evolve* [4, 67], i.e., changes to the source code can induce changes in the build system, and vice versa. Indeed, up to 27% of source code changes require accompanying changes to the build system [70].

It is difficult for developers to identify the code changes that require accompanying build system changes. Indeed, Seo *et al.* show that 30%-37% of builds triggered by Google developers on their local copies of the source code are broken, with neglected build maintenance being the most commonly detected root cause [95].

If local build breakage is not fixed before changes are committed to upstream repositories, then their team as a whole will be negatively impacted. For example, Kwan *et al.* find that 31% (60/191) of the studied IBM team builds were broken [59]. Furthermore, Hassan and Zhang find that 15% (209/1,429) of the studied IBM certification builds (i.e., builds that the development team believed were ready for testing) were broken [42]. Kerzazi *et al.* estimate that between 893-2,133 man-hours are wasted due to a build breakage rate of 19% in a large industrial system [52]. These broken team builds prevent quality assurance teams from reproducing and testing actively developed versions of a system in a timely fashion, slowing development progress and the release process.

In order to avoid these costly build breakages, we set out study the code changes that require accompanying changes to the build system. Specifically, we explore the following central question:

**Central Question:** *Can build changes be fully explained using characteristics of co-changed source and test code files?*

To address this question, we construct random forest classifiers using language-agnostic and language-aware characteristics of source and test code changes to understand when build changes are required. Through an empirical study of the Mozilla system (primarily implemented using C++), and three Java systems, we address the following three research questions:

(RQ1) *How often are build changes accompanied by source/test code changes?*

**Motivation:** If the majority of work items containing build changes do not contain accompanying source or test code changes, then code change characteristics would make poor indicators of build change. Hence, before building our classifiers, we want to know how frequently build and source/test code co-change.

**Results:** Although a minority of the source/test code changes require accompanying build changes (4%-26%), the majority of build changes co-occur with source/test code changes (53%-88%).

(RQ2) *Can we accurately explain when build co-changes are necessary using code change characteristics?*

**Motivation:** Prior work has shown that classifiers can be built to accurately explain phenomena in software engineering [42, 46, 54, 93, 96]. We conjecture that such classifiers can be built to accurately explain when a build co-change is necessary using code change characteristics.

**Results:** Yes, our classifiers can explain the source and test code changes that require accompanying build changes with an AUC of 0.60-0.88. Our Java classifiers are less accurate than the C++ classifiers (AUC of 0.60-0.78 vs. 0.88) because 75% ($\pm 10\%$) of Java build changes are not related to changes to system structure.

(RQ3) *What are the most influential code change characteristics for explaining build co-changes?*

**Motivation:** Knowing which code change characteristics are influential indicators of build change could help practitioners to identify code changes that require accompanying build changes.

**Results:** Our Mozilla (C++) classifiers derive much of their explanatory power from indicators of structural changes like adding new source files. On the other hand, since Java build co-changes rarely coincide with these structural changes, our Java classifiers derive most of their explanatory power from the historical co-change tendencies of the modified files and deeper code change characteristics like the addition or removal of `import` statements that reference non-core APIs.

**Chapter organization.** The remainder of the chapter is organized as follows. Section 6.2 describes our empirical study design, while Sections 6.3 and 6.4 present the results. Section 6.5 discloses the threats to the validity of our study. Finally, Section 6.6 draws conclusions.

## 6.2 Empirical Study Design

In this section, we describe the studied systems, and present our data extraction and analysis approaches.

### 6.2.1 Studied Systems

In order to address our research questions, we study one large system primarily implemented using C++ and three systems primarily implemented using Java. The studied systems are of different sizes and domains in order to combat potential bias in our conclusions. More importantly, the studied systems record co-change data at the work item level (see below), which is a critical precondition for our co-change analysis. The scarcity of carefully recorded work item data in practice prevents us from analyzing a larger sample of systems.

Table 6.1: [Empirical Study 3] Characteristics of the studied projects.

| **Project** | **Mozilla** | | **Eclipse-core** | | **Lucene** | | **Jazz** | |
|---|---|---|---|---|---|---|---|---|
| Domain | Internet Suite | | IDE | | Search Indexing Library | | IDE | |
| Timeframe | 1998 – 2010 | | 2001– 2010 | | 2010 – 2013 | | 2007 – 2008 | |
| # Project Files | 123,175 | | 5,490 | | 18,811 | | 67,357 | |
| Source Files (#, % of total) | 43,952 | 35% | 2,391 | 43% | 8,879 | 47% | 45,275 | 67% |
| Test Files (#, % of total) | 30,835 | 25% | 1,211 | 22% | 4,898 | 26% | 14,738 | 22% |
| Build Files (#, % of total) | 10,709 | 9% | 477 | 9% | 421 | 2% | 5,967 | 9% |
| Other Files (#, % of total) | 37,679 | 31% | 1,411 | 26% | 4,613 | 25% | 1,377 | 2% |
| # Transactions | 210,400 | | 6,391 | | 9,856 | | 36,557 | |
| # Work Items | 55,199 | | 2,452 | | 3,280 | | 11,611 | |
| # Transactions with Work Items | 79,242 | 38% | 4,092 | 64% | 6,046 | 61% | 22,485 | 62% |
| Source Work Items (#, % work items) | 45,815 | 83% | 2,130 | 87% | 2,553 | 78% | 9,869 | 85% |
| Test Work Items (#, % work items) | 9,383 | 17% | 765 | 31% | 2,084 | 64% | 2,786 | 24% |
| Build Work Items (#, % work items) | 14,477 | 26% | 427 | 17% | 443 | 14% | 608 | 5% |
| Other Work Items (#, % work items) | 5,275 | 10% | 165 | 7% | 254 | 8% | 973 | 8% |
| Source-Build Co-Change Work Items (#, % source, % build) | 12,450 | 27%, 86% | 350 | 16%, 82% | 194 | 6%, 44% | 437 | 4%, 72% |
| Test-Build Co-Change Work Items (#, % test, % build) | 4,198 | 45%, 29% | 154 | 20%, 36% | 183 | 9%, 41% | 219 | 8%, 36% |
| Source- or Test-Build Co-Change Work Items (#, % source and test) | 12,698 | 26% | 382 | 17% | 234 | 7% | 468 | 4% |
| Build without Source or Test Work Items (#, % build) | 1,779 | 12% | 82 | 19% | 209 | 47% | 140 | 23% |

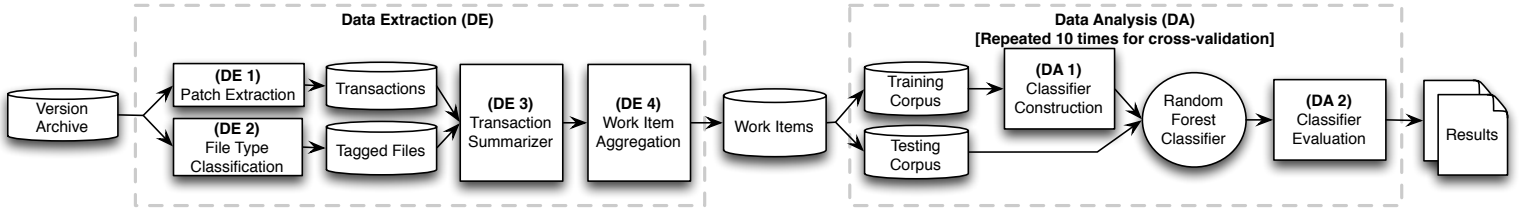



Figure 6.1: [Empirical Study 3] An overview of our data extraction and analysis approaches.

Table 6.1 provides an overview of the studied systems. Mozilla is a suite of internet tools including the Firefox web browser. Eclipse is an Integrated Development Environment (IDE), of which we studied the core subsystem. Lucene is a library offering common search indexing functionality. IBM Jazz™[1] is a proprietary next-generation IDE.

### 6.2.2 Data Extraction

Software projects evolve through continual change in the source code, test code, build system, and other artifacts. Changes to a file are often collected in *file patches* that show the differences between subsequent revisions of a single file. These file patches are typically logged in a VCS. In addition to logging file patches, modern VCSs track *transactions* (a.k.a., *atomic commits*), i.e., collections of file patches that authors commit together.

A *work item* is a development task such as fixing a bug, adding a new feature, or restructuring an existing feature. Several transactions may be required to complete a work item, since developers from different teams may need to collaborate. Work items are often logged in an Issue Tracking System (ITS) like Bugzilla or IBM Jazz and branded with a unique identifier. This ID helps to identify the transactions that are associated with a work item.

We extract work item data from each of the studied systems in order to address our research questions. Figure 6.1 provides an overview of our approach, for which the data extraction component is broken down into four steps. We briefly describe each step of our data extraction approach below.

[1] http://www.jazz.net. IBM and Jazz are trademarks of IBM Corporation in the US, other countries, or both.

#### (DE 1) Patch Extraction

After gathering the VCS archives for each studied project, we extract all transactions as well as authorship, timestamps, and commit message metadata. Although the studied systems use different VCSs (i.e., Git and Mercurial), we wrote scripts to extract transactions and metadata in a common format.

#### (DE 2) File Type Classification

In order to assess whether a transaction (and hence, a work item) impacts the build system, we use the file type classification process from our prior work [70], which tags each file in a project history as either a source, test, or build file. Build system files include helper scripts, as well as construction and configuration layer specifications (such as `make` or `ant` files). Source code files implement software logic. Test code files contain automated tests that check the software for regressions.

The file type classification process was semi-automatic. Table 6.1 lists the number of files classified under each category for the studied systems. Most files could be classified using filename conventions, e.g., file extensions. However, many extensions were ambiguous, e.g., `.xml`. After classifying unambiguous file types, the remaining files were manually classified. For example, of the 123,175 Mozilla files, approximately 20,000 files remained unclassified after all known filename conventions were exhausted. Through manual inspection, we found project-specific extension types that could be classified automatically, further reducing the number of unclassified files to roughly 5,000. The remaining 5,000 or so files were manually classified.

Table 6.2: [Empirical Study 3] A taxonomy of the studied language-agnostic code change characteristics. Each is measured once for source code and once for test code.

| Attribute Name | Type | Definition | Rationale |
|---|---|---|---|
| File added | Boolean | True if a given work item adds new source or test files. | Adding new source files changes the filesystem layout of the codebase, which may require accompanying build changes to include the new file. |
| File deleted | Boolean | True if a given work item deletes old source or test files. | Deleting old source files changes the filesystem layout of the codebase, which may require accompanying build changes to disregard the dead file. |
| File renamed | Boolean | True if a given work item renames source or test files. | Renaming a source file alters the filesystem layout of the codebase, invalidating prior dependencies while creating new ones, which may require accompanying build changes. |
| File modified | Boolean | True if a given work item modifies existing source or test files. | With the exception of the special language-specific cases (see below), modification of source code should rarely require build changes, since modifications do not alter the structure of a system. |
| Prior build co-changes* | Numeric | We compute the proportions of prior work items that were build co-changing for each of the source and test files in a given work item. We select the maximum proportion of the work item's changed files. | Historical co-change tendencies may provide insight into future co-change trends. |
| Number of files* | Numeric | The number of source and test files that were involved in a given work item. | Changes that impact more files may be more likely to require accompanying build changes. |

* Could not be calculated for Jazz due to privacy concerns.

### (DE 3) Transaction summarizer

Next, we produce transaction summaries for all transactions that contain source, test, and/or build file changes, which consist of: (1) measured characteristics that describe the code change, and (2) a boolean value noting whether or not at least one build file was changed. A summary of the measured code change characteristics and the rationale for their use is given in Tables 6.2 and 6.3.

### (DE 4) Work item aggregation

Our prior work has shown that transactions are too fine-grained to accurately depict development tasks [70]. It may take several transactions to resolve a work item. In

Table 6.3: [Empirical Study 3] A taxonomy of the studied language-aware code change characteristics. Each is measured once for source code and once for test code.

| Attribute Name | Type | Definition | Rationale |
|---|---|---|---|
| Changed dependencies | Boolean | True if a given work item adds or removes dependencies on other code through `#include` preprocessor directives for C++ code or `import` statements in Java code. | Dependency changes may need to propagate to the build system. |
| Added/removed dependencies | Boolean | True if the dependency being: (1) added does not appear in any other source or test file, or (2) removed has been completely removed from all source and test files. | Adding or removing dependencies indicates that a new dependency may have been introduced or an old one relaxed. Such changes may need to propagate to the build system. |
| Added/removed non-core dependencies | Boolean | True if the conditions listed for Added/removed dependencies are satisfied by a dependency that is not part of the core language API. | Adding or removing dependencies on core language APIs will not have an impact on the build process, and hence may introduce noise in the Added/removed dependencies metric. |
| Changed conditional compilation (C++ only) | Boolean | True if a given work item adds new or removes old #if[n][def] preprocessor directives. | Conditional compilation is often used to control platform- or feature-specific functionality in the source or test code. The conditions for these blocks of code often depend on configuration layer settings. |

such cases, build changes often appear in different transactions than the corresponding source or test code changes. To avoid missing cases of co-change, we group transactions that address the same work item together by examining the transaction commit messages for work item IDs.

### Bias Assessment

As shown in Table 6.1, the aggregation to work items is lossy, since it relies heavily on developer behaviour to link transactions to work items. Overall, 38%-64% of the transactions can be connected to work items. The lack of well-linked work item data is a

known problem [15, 88]. Hence, we first evaluate whether the lossy nature of work item aggregation introduces bias in our dataset. We are primarily concerned with two types of bias:

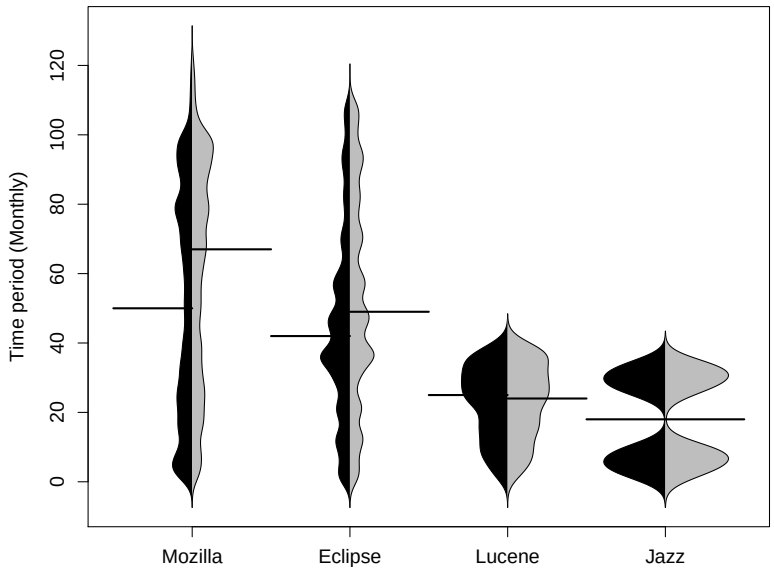
1. Time periods in project history may be missing due to the lossy nature of work item aggregation, i.e., we only have work item data for certain time periods.

2. Work item linkage may be a property of project experience [15], i.e., experienced developers might be more likely to provide the links to work items in their commits.

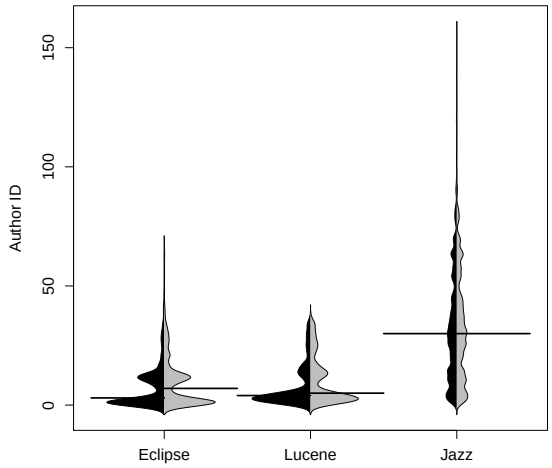
To study the extent of these biases, we compare the number of transactions per month to the number of work items per month and study how these measures evolve over time. We also compare developer contributions in terms of the number of transactions and work items. Figure 6.2 visualizes these distributions using beanplots [50]. Beanplots are boxplots in which the vertical curves summarize the distributions of different datasets. The horizontal black lines indicate the median values. Due to differences in scale, we separate the Java beanplots (Figure 6.2b) from the Mozilla one (Figure 6.2c).

Figures 6.2a and 6.2b show that Eclipse-core, Lucene, and Jazz share highly symmetrical beanplots, indicating that transactions and work items share similar temporal and developer contribution characteristics. The median lines in Jazz and Lucene are almost identical, while the median of the work items is higher than that of the transactions in the Eclipse-core project. The slight difference in medians indicates that the work item granularity introduces minimal skew with respect to the transaction data.

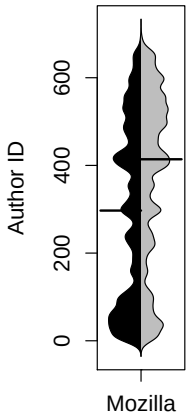
The asymmetrical nature of the Mozilla plot in Figure 6.2a shows that there is skew introduced, i.e., very few transactions could be linked to work items in the initial Mozilla




(a) Monthly changes




(b) Changes made by each developer




(c) Changes made by each developer

Figure 6.2: [Empirical Study 3] Comparison of the time and developer distribution of transactions (black) and work items (grey).

development months. Once the practice of recording the work item ID in the commit message was more firmly established, the symmetry of the beanplot increases, indicating that the temporal characteristics between the two datasets are similar from that point on. To resolve this, we removed the initial 12 development months of Mozilla prior to performing our case study. Figure 6.2c shows that this filtering also makes the distribution of Mozilla developer contributions less skewed, i.e., the bias in our data has been controlled.

### 6.2.3 Data Analysis

Figure 6.1 provides an overview of our data analysis approach. The work items are split into training and testing corpora. Classifiers are constructed using the training corpus, and their performance is evaluated on work items in the testing corpus. We briefly describe each step in our analysis below.

#### (DA 1) Classifier Construction

We use the random forest technique to construct classifiers (one for each studied system) that explain when build changes are necessary. The random forest technique constructs a large number of decision trees at training time [17]. Each node in a decision tree is split using a random subset of all of the attributes. Performing this random split ensures that all of the trees have a low correlation between them [17]. Since each tree in the forest may report a different outcome, the final class of a work item is decided by aggregating the votes from all trees and deciding whether the final score is higher than a chosen threshold.

**(DA 2) Classifier Evaluation**

To evaluate the performance of a classifier, we use it to classify work items in a testing corpus and compare its deduction to the known result. To obtain the testing corpus and evaluate the performance of our classifiers, we use tenfold cross-validation. Cross-validation splits the data into ten equal parts, using nine parts for the training corpus, setting aside one for the testing corpus. The process is repeated ten times, using a different part for the testing corpus each time.

Table 6.4 shows the confusion matrix constructed based on the cross-validation classification results. The performance of the decision tree is measured in terms of recall, precision, F-measure, and AUC. We describe each metric below.

- **Recall**: Of all known build co-changing work items, how many were classified as such, i.e., $\frac{a}{a+b}$.
- **Precision**: Of the work items that are classified as build co-changing, how many actually did co-change, i.e., $\frac{a}{a+c}$.
- **F-measure**: The harmonic mean of precision and recall, i.e., $2 \cdot \frac{precision \cdot recall}{precision + recall}$.
- **Area Under the Curve (AUC)**: The area under the curve that plots true positive rate ($\frac{a}{a+b}$) against the false positive rate ($\frac{c}{c+d}$), for various values of the chosen threshold used to determine whether a work item is classified as build co-changing. Values of AUC range between 0 (worst classifier performance) and 1 (best classifier performance).

We first construct classifiers using only the language-agnostic characteristics from Table 6.2. We then add language-aware characteristics of Table 6.3 to the classifiers.

Table 6.4: [Empirical Study 3] An example confusion matrix.

| | Classified As | |
|---|---|---|
| Actual Category | Change | No Change |
| Change | a | b |
| No Change | c | d |

**Handling imbalanced categories**

Table 6.1 shows that build co-changing work items are the minority category (4%-26%). Classifiers tend to favour the majority category, since it offers more explanatory power, i.e., classification of "no build change needed" will likely be more accurate than classification of "build change needed." To combat the bias of imbalanced categories, we re-balance the training corpus to improve minority category performance [12, 46]. Re-balancing is not applied to the testing corpus.

We chose to re-balance the data using a re-sampling technique, which removes samples from the majority category (under-sampling) and repeats samples in the minority category (over-sampling). We chose to re-sample rather than apply other re-balancing techniques like re-weighing (i.e., assigning more weight to correctly classified minority items) because we found that re-sampling yielded slightly better results, which is consistent with findings reported in the literature [34, 46].

Re-sampling is performed with a given bias $\beta$ towards equally distributed categories ($\beta = 1$). No re-sampling is performed when $\beta = 0$. Values between $0 < \beta < 1$ vary between unmodified categories and equally distributed categories. We report findings for different values of $\beta$.

## 6.3 Mozilla Case Study Results (C++)

In this section, we present the results of our Mozilla case study with respect to our three research questions. For each research question, we present our approach for addressing it followed by the results that we observe.

### (RQ1) How often are build changes accompanied by source/test code changes?

#### Approach

We measure the rate of build and source/test co-change as a percentage of all build changes. Specifically, we report the percentage of build-changing work items that also contain source or test code changes.

#### Results

**Most Mozilla build changes co-occur with source or test code.** While Table 6.1 shows that Mozilla build co-change is the minority category with respect to all source and test changes (27%), source/test co-change is the majority category with respect to all build changes.

Indeed, 86% of Mozilla build-changing work items also change source code, and 29% also change test code. Altogether, 88% of Mozilla build changes co-occur with source/test code changes.

Table 6.5: [Empirical Study 3] The median of the recall, precision, F-measure, and AUC values of the ten classifiers constructed at re-sampling bias ($\beta$) levels of 0, optimal, and 1. The first row shows the raw values while the second row shows the improvement of adding language-specific characteristics to language-agnostic classifiers.

| | **Mozilla** | | | **Eclipse-core** | | | **Lucene** | | | **Jazz** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bias ($\beta$) | 0.0 | 0.40 | 1.0 | 0.0 | 0.59 | 1.0 | 0.0 | 0.73 | 1.0 | 0.0 | 0.32 | 1.0 |
| Recall | 0.57 | 0.63 | 0.67 | 0.30 | 0.39 | 0.43 | 0.14 | 0.31 | 0.39 | 0.24 | 0.31 | 0.40 |
| | +0.05*** | +0.00 | -0.02 | +0.01 | +0.00 | +0.01 | +0.03 | -0.05 | +0.00 | +0.04 | +0.08** | +0.15*** |
| Precision | 0.74 | 0.63 | 0.53 | 0.50 | 0.39 | 0.34 | 0.38 | 0.31 | 0.33 | 0.36 | 0.31 | 0.24 |
| | +0.06*** | +0.10*** | +0.15*** | +0.09* | +0.07** | +0.09** | +0.09 | +0.11** | +0.14*** | -0.12 | -0.06 | -0.13 |
| F-measure | 0.64 | 0.63 | 0.60 | 0.37 | 0.39 | 0.38 | 0.20 | 0.31 | 0.36 | 0.29 | 0.31 | 0.30 |
| | +0.05*** | +0.06*** | +0.10*** | +0.02 | +0.02 | +0.05* | +0.04 | +0.05* | +0.10** | +0.1 | +0.05 | +0.0 |
| AUC | 0.86 | 0.88 | 0.88 | 0.68 | 0.69 | 0.68 | 0.75 | 0.78 | 0.79 | 0.61 | 0.60 | 0.59 |
| | +0.03*** | +0.04*** | +0.05*** | +0.02 | +0.03* | +0.04* | +0.09** | +0.07** | +0.10*** | +0.05** | +0.03* | +0.04* |

Statistical significance of the improvement achieved through language-specific characteristics (One-tailed Mann-Whitney U-test): * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Co-occurrence alone does not indicate that there is a causal relationship between build changes and source/test changes. However, the inflated rates of co-occurrence that we observe suggest that there is likely information in these co-changes that we can leverage to better understand the types of source and test changes that require accompanying build changes.

*While build co-changing work items are the minority category with respect to all source and test changes, source/test co-changing work items are the vast majority of all build changes in Mozilla. This suggests that source and test change characteristics may help to explain when build changes are necessary.*

## (RQ2) Can we accurately explain when build co-changes are necessary using code change characteristics?

### Approach

Table 6.5 shows performance values with $\beta = 0, 1, \theta$, where $\theta$ is the value where recall and precision values are equal. We refer to $\theta$ as the optimal $\beta$ value, since we value precision (are build co-change classifications reliable?) and recall (are we finding all of the build co-changes?) equally.

### Results

**Our Mozilla classifiers vastly outperform random classifiers.** The source- or test-build co-change work items row of Table 6.1 shows that a random classifier would achieve 0.26 precision at best. Table 6.5 shows that our Mozilla classifiers more than double the precision of random classifiers, achieving a recall and precision of 0.63 ($\beta = \theta$). Moreover, since the AUC metric is designed such that a random classifier would achieve an

AUC of 0.5, Table 6.5 shows that our Mozilla classifier outperforms a random classifier by 0.38, achieving an AUC of 0.88.

**Language-aware characteristics improve classifier performance.** Table 6.5 shows that when language-aware characteristics are added to our classifiers, the overall performance improves. Indeed, despite slight decreases in recall, the precision, F-measure, and AUC values improve. To test whether the observed improvement is statistically significant, we performed one-tailed Mann-Whitney U-tests ($\alpha = 0.05$). Test results indicate that the improvements in precision, F-measure, and AUC are statistically significant.

*Using language-aware metrics, we can improve Mozilla classifier performance, achieving an AUC of 0.88.*

## (RQ3) What are the most influential code change characteristics for explaining build co-changes?

### Approach

To study the most influential code change characteristics in our random forest classifiers, we compute Breiman's variable importance score [17] for each studied characteristic. The larger the score, the greater the importance of the code change characteristic.

Figure 6.3 shows the variable importance scores for the studied code change characteristic in each of the ten folds using boxplots. Since analysis of variable importance scores at $\beta = 0$ and $\beta = 1$ show similar trends, Figure 6.3 shows only the variable importance scores for the classifier trained with $\beta = \theta$ to reduce clutter.

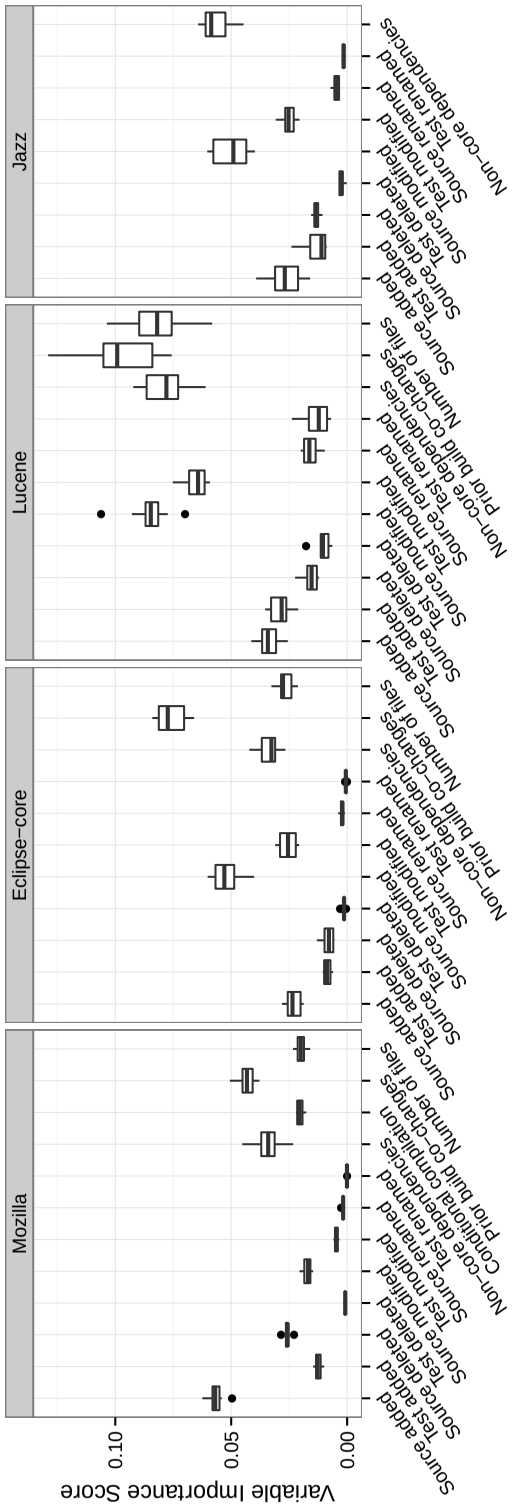



Figure 6.3: [Empirical Study 3] Variable importance scores for the studied code change characteristics ($\beta = \theta$).

### Results

**Source and test changes that modify the structure of a system and prior build co-change are important explanatory factors of build changes in Mozilla.** Figure 6.3 shows that activities that alter the structure of a system like adding/deleting source code and adding/removing non-core libraries through `#include` statements are among the most important variables used by the Mozilla classifiers. Furthermore, prior build co-change is also an important indicator of future build co-changes. While renaming operations also modify the structure of a system, their low importance scores are likely due to the relative infrequency of rename operations in the Mozilla VCS history.

*Our Mozilla classifiers derive much of their explanatory power from frequently occurring structural changes like adding source files, as well as historical co-change tendencies.*

## Discussion

While our classifiers perform well for Mozilla in general, we wondered whether the nature of the programming languages used in a subsystem (i.e., top-level directory) will have an impact on classifier performance. While Mozilla primarily consists of C++ code, it also contains subsystems implemented using several other programming languages (e.g., Javascript and PHP). To evaluate our conjecture, we construct and analyze directory-specific Mozilla classifiers.

### Approach

In order to study classifier performance on a subsystem basis, we mark each work item with a listing of directories that are impacted by source and test changes within the

work item. We then build classifiers for each directory separately. We ignore directories with fewer than 50 work items because we want our tenfold cross validation approach to test on at least five work items (10% of 50 work items).

#### Results

**As we suspected, Mozilla classifier performance is the weakest in subsystems primarily implemented using web technologies.** We find that most Mozilla subsystems have classifier performance that exceeds 0.7 AUC. However, the Mozilla `webtools` subsystem has subpar classifier performance when compared to the other subsystems (0.31-0.45 AUC). We observe similar weak classifier performance in the test subdirectories of the `js` subsystem. The code in these subsystems is written using web technologies, such as Javascript for testing the Mozilla Javascript engine, and PHP and Perl CGI for implementing tools like Bugzilla. Web technologies differ in terms of build tooling from the C++ code for which our classifiers perform well. While C++ code must be compiled and linked by the build system, the web code must only be tested, packaged, and deployed. We constructed special classifiers that detect the PHP `require` keyword as a dependency change, but it did not improve performance. The changes that induce build changes for web technologies are less code-related, and are thus more difficult to explain.

*While coarse-grained file modifications and dependency information explain build changes in C++ subsystems reasonably well, they do not explain build changes in subsystems with web application code.*

## 6.4 Java Case Study Results

Our findings in Section 6.3 show that since Mozilla build changes are frequently accompanied by source and test changes (RQ1), we can derive information from the source and test changes to accurately explain when build changes are necessary (RQ2). This confirms common wisdom among C/C++ developers. However, we find that the programming languages used in a subsystem seem to influence the performance of our classifiers, i.e., it is harder to explain build changes in the subsystems that are implemented using web technologies than those implemented using C++. Furthermore, our prior work has shown that there are differences in the evolution of Java and C build systems, likely due to the built-in dependency management performed by the Java compiler [67].

To further investigate whether those environment changes have an impact on our co-change classifiers, we replicate our Mozilla case study on three Java systems. In this section, we present the results of our Java case study with respect to our three research questions. Since we use the same approaches that were presented in Section 6.3, we only discuss the results that we observe with respect to each research question below.

### (RQ1) How often are build changes accompanied by source/test code changes?

**Similar to Mozilla, Java build systems frequently co-change with source or test code.** Table 6.1 shows that between 44% (Lucene) and 82% (Eclipse-core) of Java work items that contain build changes also change source code. Furthermore, between 36% (Jazz, Eclipse-core) and 41% (Lucene) of work items that change the build also change

test code. Altogether, between 53% (Lucene) and 81% (Eclipse-core) of Java build changes co-occur with source or test changes.

*Similar to C++ build systems, most Java build changes are accompanied by source or test changes, suggesting that Java source and test change characteristics may also help to explain when Java build changes are necessary.*

## (RQ2) Can we accurately explain when build co-changes are necessary using code change characteristics?

**Similar to Mozilla, our Java classifiers outperform random classifiers.** Table 6.5 shows that Eclipse-core, Lucene, and Jazz classifiers achieve recall and precision of 0.31-0.39 ($\beta = \theta$) and AUC values of 0.60-0.78. Our classifiers for Java systems outperform random classifiers by a minimum factor of two, since random classifiers are theoretically constrained to a precision of achieve between 0.04 (Jazz) and 0.16 (Eclipse-core).

We achieve the lowest performance in our Jazz classifiers. Unfortunately, the prior build co-changes and number of files characteristics could not be calculated for Jazz due to limitations of the provided dataset. We suspect that adding these metrics would bring the Jazz classifier performance up to match the performance of the other Java case studies.

Similar to Mozilla, language-aware characteristics improve classifier performance, especially in terms of precision and AUC. Table 6.5 shows that the AUC of our Java classifiers improves by 0.03-0.07 when language-aware characteristics are added ($\beta = \theta$). Mann-Whitney U tests indicate that these AUC improvements are significant.

**On the other hand, our Java classifiers under-perform with respect to our Mozilla**

Table 6.6: [Empirical Study 3] Categories of identified Eclipse-core build changes with a 95% confidence level and a confidence interval of $\pm$ 10%.

| **Category** | **Task** | **Total #** | **%** | **# correctly classified** |
|---|---|---|---|---|
| System structure | Refactorings | 19 | 25% | 8 |
| Build maintenance | Build tool configuration | 15 | 20% | 0 |
| | Build defects | 6 | 8% | 0 |
| Release engineering | Add platform support | 12 | 16% | 2 |
| | Packaging fixes | 12 | 16% | 3 |
| | Library versioning | 8 | 11% | 0 |
| Test maintenance | Test infrastructure | 3 | 4% | 0 |

**classifier.** The difference in performance is substantial — a reduction of roughly 33% in most of the performance metrics. We hypothesize that such a consistent difference in the performance of Mozilla and the Java classifiers is related to fundamental differences in the C++ and Java compile and link tools. For example, when using a C++ compiler, developers often rely on external build tools like `make` to manage dependencies amongst source files, while Java compilers automatically resolve these dependencies [29]. Since Java compilers are more intelligent in this regard, build changes are rarely required to track file-level dependencies.

To evaluate our hypothesis, we selected a representative sample of work items for manual analysis, since the full set of work items is too large to study entirely. Similar to Chapter 5, we obtain proportion estimates that are within 10% bounds of the actual proportion with a 95% confidence level, we use a sample size of $s = \frac{z^2 p(1-p)}{0.1^2}$, where $p$ is the proportion that we want to estimate and $z = 1.96$. Since we did not know the proportion in advance, we use $p = 0.5$. We further correct for the finite population of build co-changing work items in Eclipse-core (i.e., 382, see Table 6.1) using $ss = \frac{s}{1+\frac{s-1}{382}}$ to obtain a sample size of 77. Table 6.6 shows the percentage of randomly selected work items that are associated with each change category.

**The majority of Eclipse-core build changes are unrelated to the structure of the system.** Table 6.6 shows that release engineering tasks (e.g., expanding platform support) and build maintenance tasks (e.g., compiler flag settings) account for $43\% \pm 10\%$ and $28\% \pm 10\%$ of build change respectively, a larger portion than structural changes ($25\% \pm 10\%$). Indeed, $75\% \pm 10\%$ of the studied build-changing work items were unrelated to the structure of the system (i.e., build maintenance, release engineering, and test maintenance).

For example, we studied a defect (ID 226462) where Eclipse was crashing when operating in a specific environment. The source code was fixed to prevent the crash, however the assigned developer discovered that a particular compiler warning could have notified the team of the issue prior to release. The work item fix included the build change to enable the compiler warning to prevent regression. Our classifiers fail to explain these sorts of build changes that do not directly link to source code changes, and in general, most source code changes in the Java systems do not require accompanying build changes (see Table 6.1). Hence, the factors that drive Java build change are more elusive and difficult to isolate based on code change characteristics alone, which might provide an additional difficulty for developers to realize when they need to make a build system change.

Furthermore, a large proportion of source/build co-change requires expertise from different team roles. For example, the source code maintenance tasks require developer expertise, while release engineering and build maintenance tasks require release engineering expertise. This finding complements those of Wolf *et al.*, who find that team communication is a powerful predictor of build outcome [108].

**Nonetheless, our Java classifiers can explain the build changes that are relevant**

**to a developer.** Indeed, Table 6.6 shows that 8 of the 19 work items that alter system structure were identified by our Eclipse-core classifier. In contrast, our classifiers only identified 5 of the 32 release engineering work items and no build or test maintenance work items. Since our classifiers are based on code change characteristics, they cannot assist release engineers, build maintainers, or quality assurance personnel. We plan to expand the scope of our classifiers to assist these practitioners in future work.

*Our Java classifiers outperform random classifiers, achieving an AUC of 0.60-0.78. Yet, they under-perform with respect to the Mozilla classifier (0.88 AUC), since Java build co-changes are mostly related to release engineering activities rather than being purely code-based.*

## (RQ3) What are the most influential code change characteristics for explaining build co-changes?

**Source and test changes that alter system structure are not good indicators of build changes in studied Java systems.** Figure 6.3 shows that source code modifications that do not alter the structure of a system (i.e., Source/Test modified) are more important indicators of build changes in the Java systems than those that do. This finding complements Table 6.6, indicating that structural changes are not very important indicators of build change in Java systems. The relative infrequency of structural co-change for the Java build systems is likely due to the Java compiler's built-in support for dependency resolution.

**Since structural co-changes are of little value for our Java classifiers, these classifiers need to derive co-change indications from other code change characteristics.** Figure 6.3 shows that adding or removing non-core dependencies (heuristically flagged by changes to Java `import` statements) helps to fill the void left by the missing

structural cues. Although omitted from Figure 6.3 due to space constraints, the less detailed versions of the dependency characteristic (see Table 6.3) have lower variable importance scores, suggesting that narrowing the scope of the dependency characteristic to only detect non-core API changes improves its performance in our classifiers. Furthermore, the prior build co-changes characteristic is the most important indicator of build co-change in our Eclipse-core and Lucene classifiers. Prior build co-changes also plays an important role in our Mozilla classifiers, indicating that historical co-change tendencies are consistent indicators of future build co-changes.

*Since Java build changes rarely coincide with changes to the structure of a system, Java build changes are more effectively explained by historical co-change tendencies and changes to non-core Java API import statements.*

## 6.5 Threats to Validity

We now discuss threats to the validity of our empirical study.

### 6.5.1 Construct Validity

We make an implicit assumption that the collected data is correct, i.e., in the data used to build our classifiers, developers always commit related source, test, and build changes under the same work item when necessary. On the other hand, our work item data is robust enough to handle cases where developers did forget to change the build in the same transaction as a corresponding code change.

Our bias analysis in Section 6.2 shows that work item aggregation skews the developer contributions in Mozilla. To combat this bias, we remove the skewed early development period from the dataset prior to performing our case studies.

### 6.5.2 Internal Validity

We use code change characteristics to explain build changes because most of the build changes coincide with code changes. We selected metrics that cover a wide range of change characteristics that we felt would induce build changes. However, other metrics that we have overlooked may also improve the performance of our classifiers.

Although source and build code may appear together in a co-change, there may be no causal link between the changes. Indeed, as Grant *et al.* point out, many co-changes are entirely coincidental [39]. While these coincidental co-changes introduce noise into our analyses, our classifiers are still robust enough to provide a meaningful amount of explanatory power, with AUC values ranging from 0.60-0.88.

Our file classification approach is subject to the authors' opinion and may not be 100% accurate. The authors used their best judgement to classify files that could not be automatically classified using filename conventions. The authors rely on their prior experience with build systems to classify files that may have fit several categories [3, 4, 66, 67, 70]. We have also used this classification approach in our prior work [70] and made the classified files available online[2] to aid in future research.

### 6.5.3 External Validity

Despite the difficulty of collecting linked work item data, we study four software systems. However, our sample size may limit the generalizability of our results. To combat this limitation, we study systems of different sizes and domains. Moreover, we augment our study of three open source systems with the proprietary IBM Jazz system.

We suspect that the differences in the C++ and Java build change classifiers are due

[2] http://sailhome.cs.queensu.ca/replication/shane/PhD/

to differences in the dependency support of C++ and Java build tools. However, there are likely several confounding factors that we could not control for in such a small sample. For example, we observed variability in the performance of the three studied Java systems. Thus, the differences that we observe among C++ and Java systems may simply be due to natural variability among the systems rather than indicative of differences between the C++ and Java build systems. While deeper manual analysis seems to support the latter case, further replication of our results in other (particularly C++) systems could prove fruitful.

## 6.6 Chapter Summary

Build systems age in tandem with the software systems that they are tasked with building. Changes in source and test code often require accompanying changes in the build system. Developers may not be aware of changes that require build maintenance, since build systems are large and complex. Neglecting such build changes can cause build breakages that slow development progress, or worse can cause the build system to produce incorrect deliverables, impacting end users. Hence, we set out to answer this question:

**Central Question:** *Can build changes be fully explained using characteristics of co-changed source and test code files?*

Through an empirical study of four large software systems, we found that the answer is no:

- While 4%-26% of work items that change source/test code also change the build system, 53%-88% of build-changing work items also contain source/test changes,

suggesting that there is a strong co-change relationship between the build system and source/test code.

- Our Mozilla build co-change classifiers achieve an AUC of 0.88, with these co-changes being most effectively indicated by structural changes to a system and historical build co-change tendencies of the modified files.

- However, classifier performance suffers in systems composed of Java and web application code due to a shift in the usage and design of build technology from requiring build changes for structural code changes (e.g., adding a file) to enabling cross-disciplinary activities related to release engineering and general build maintenance.

Our results suggest that there are differences in the way that the maintenance of the build system materializes in systems primarily implemented using different programming languages (i.e., C++ and Java).

### 6.6.1 Concluding Remarks

In the past three chapters, we have focused on the overhead introduced by the maintenance of the build system. In the following chapter, we turn our attention to the overhead introduced by the execution of the build system.

CHAPTER 7

# Build Hotspots

CENTRAL QUESTION

? *Which files should development teams optimize first to improve build performance the most? Which properties of hotspot files should development teams focus optimization effort on?*

**An earlier version of the work in this chapter appears in** the Springer Journal of Automated Software Engineering [69]

## 7.1 Introduction

Build systems specify how source code, libraries, and data files are transformed into deliverables, such as executables that are ready for deployment. Build tools (e.g., `make` [35]) orchestrate thousands of order-dependent commands, such as those that compile and

test source code, to ensure that deliverables are rebuilt correctly. Such a build tool needs to be executed every time developers modify source code, and want to test or deploy the new version of the system on their machine. Similarly, continuous integration and release engineering infrastructures on build servers rely on a fast build system to provide a quick feedback loop.

Since large software systems are made up of thousands of files that contain millions of lines of code, executing a full build can be prohibitively expensive, often taking hours, if not days to complete. For example, builds of the Firefox web browser for the Windows operating system take more than 2.5 hours on dedicated build machines.[1] In a recent survey of 250 C++ developers, more than 60% of respondents report that build speeds are a significant issue.[2] Indeed, while developers wait for build tools to execute the set of commands necessary to synchronize source code with deliverables, they are effectively idle [45].

To avoid incurring such a large build performance penalty for each build performed by a developer, build tools such as `make` [35] provide *incremental builds*, i.e., builds that calculate and execute the minimal set of commands necessary to synchronize the built deliverables with any changes made to the source code. Humble and Farley suggest that incrementally building and testing a change to the source code should take no more than 1.5 minutes [45]. Developers have even scrutinized 5-minute long incremental build processes,[3] calling the process "abysmally slow."[4] Again, the slower the incremental build process, the longer the idle period, frustrating developers and slowing down development progress.

[1] http://tbpl.mozilla.org/
[2] http://mathiasdm.com/2014/01/24/a-c-questionnaire-on-build-speed-the-results-are-in/
[3] https://bugs.webkit.org/show_bug.cgi?id=32921
[4] https://bugs.webkit.org/show_bug.cgi?id=33556

To assess build performance bottlenecks in the real world, we asked developers of the GLib and PostgreSQL systems to list the files that slowed them down the most when rebuilding them incrementally. While the reported bottlenecks were often the files that triggered a relatively long rebuild process (since many source code files depend on them), paradoxically, there were other files that took a longer time to rebuild, but were not pointed out by the developers. Many of these slower files were not perceived to be build bottlenecks because they rarely changed over time (and hence, rarely needed to be rebuilt by the developers). Indeed, although often overlooked by build optimization approaches, the frequency of change that a file undergoes influences how developers perceive build performance issues. Indeed, we set out to answer the following question:

**Central Question:** *Which files should development teams optimize first to improve build performance the most? Which properties of hotspot files should development teams focus optimization effort on?*

In order to address this central question, we make two main contributions:

1. We propose an approach to detect hotspots by analyzing the build dependency graph and the change history of a system (Section 7.3). We evaluate our approach by simulating the build time improvement of build hotspots for a developer by using historical data (Section 7.5). We find that optimization of the files identified by the hotspot approach would lower the total future rebuild cost more than optimization of the files that trigger the slowest rebuild processes, change the most frequently, or are used the most throughout the codebase.

2. We study the characteristics of build hotspots in the studied systems (Section 7.6). We find that logistic regression models can explain 32%-57% of the identified

build hotspots using the architectural and code properties of files. Furthermore, our GLib and Qt models identify hotspot-prone subsystems that would benefit most from architectural refinement.

**Chapter organization.** The remainder of this chapter is organized as follows. Section 7.2 describes incremental builds and build hotspots in more detail. Section 7.3 presents the hotspot detection approach. Section 7.4 describes the setup of our empirical study of four open source systems. Section 7.5 presents the results of our simulation experiment. Section 7.6 presents the results of our study of the characteristics of build hotspots. Section 7.7 discloses the threats to the validity of our empirical study. Finally, Section 7.8 draws our conclusions.

## 7.2 Build Hotspots

### 7.2.1 Incremental Builds

Developers who make source code changes would like to quickly produce modified deliverables in order to test their changes. Hence, the cornerstone feature of a build system is the incremental build, which can reduce the cost of a full build dramatically. After performing a full build that produces initial copies of the necessary deliverables, incremental builds only execute the commands necessary to update the deliverables ("build targets") impacted by source code changes.

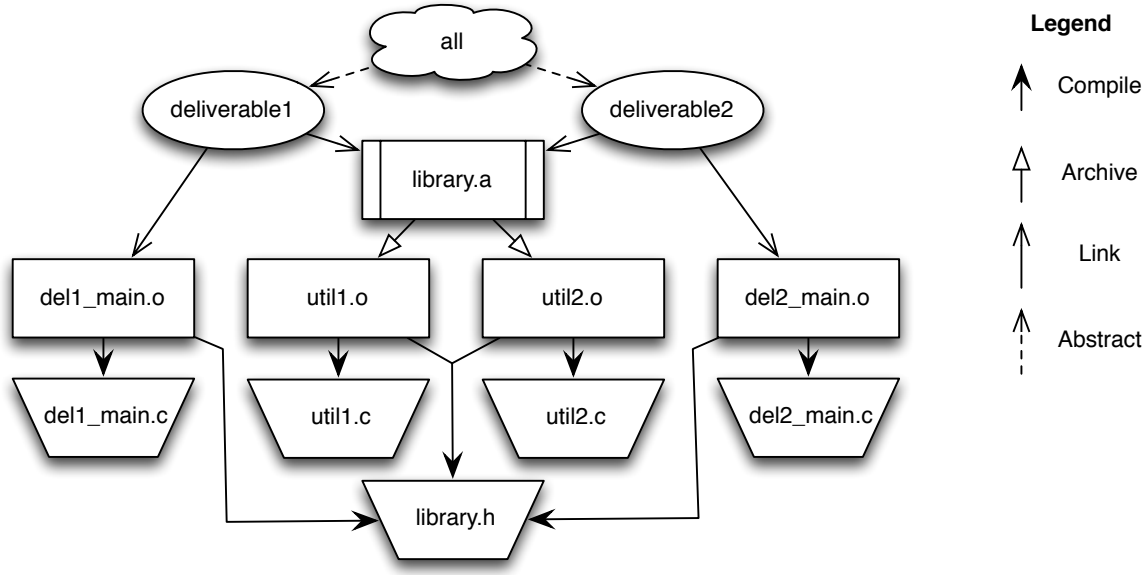
For example, consider the build dependency graph depicted in Figure 7.1a, which represents the dependencies in the `make` specification of Figure 7.1b. The `all` node in the graph is *phony*, i.e., a node used to group deliverables together into abstract

build phases rather than to represent a file in the filesystem. The full build will execute four compilation commands (recipe 4) to produce build targets `del1_main.o`, `util1.o`, `util2.o`, and `del2_main.o`, as well as an archive command (recipe 3) to produce `library.a`, and finally, two link commands (recipes 1 and 2) to produce `deliverable1` and `deliverable2`. If `del1_main.c` is modified after a full build has been performed, an incremental build only needs to recompile `del1_main.o` and re-link `deliverable1`. As software systems (and build dependency graphs) grow, the minimizing behaviour of incremental builds saves developers time.

### 7.2.2 Build Hotspots

Although incremental builds tend to save time, changes to header files often trigger slow rebuild processes [61]. For example, Figure 7.1a shows that changes to `library.h` will trigger the equivalent of a full build, since all four `.c` files reference `library.h`, and will thus need to be recompiled when it changes. In turn, `library.a` will be re-archived and the two deliverables will be re-linked.

To better understand how developers are impacted by such build performance bottlenecks (e.g., files that trigger slow rebuild processes), we asked the three most active contributors to GLib and PostgreSQL (two long-lived and rapidly evolving open source systems) to pick five files that they believe slow them down the most when rebuilding. Surprisingly, the files that were reported as bottlenecks were not the ones with the worst raw build performance. In fact, of the bottlenecks reported by the three developers, the files with the worst performance appear $61^{st}$ (GLib) and $32^{nd}$ (PostgreSQL) in the lists of files ordered by actual rebuild cost (i.e., the time taken to incrementally build the system after a change to one of those files). Indeed, the respondents seemed




(a) Build dependency graph.

```
1  CC = gcc
2  LIBTOOL = libtool
3
4  .PHONY: all
5  all: deliverable1 deliverable2
6
7  deliverable1: del1_main.o library.a
8          $(CC) -o $@ $^ # recipe 1
9
10 deliverable2: del2_main.o library.a
11         $(CC) -o $@ $^ # recipe 2
12
13 library.a: util1.o util2.o
14         $(LIBTOOL) -static -o $@ $^ # recipe 3
15
16 %.o: %.c library.h
17         $(CC) -c $< # recipe 4
```

(b) `make` implementation

Figure 7.1: [Empirical Study 4] An illustrative build dependency graph and its `make` implementation.

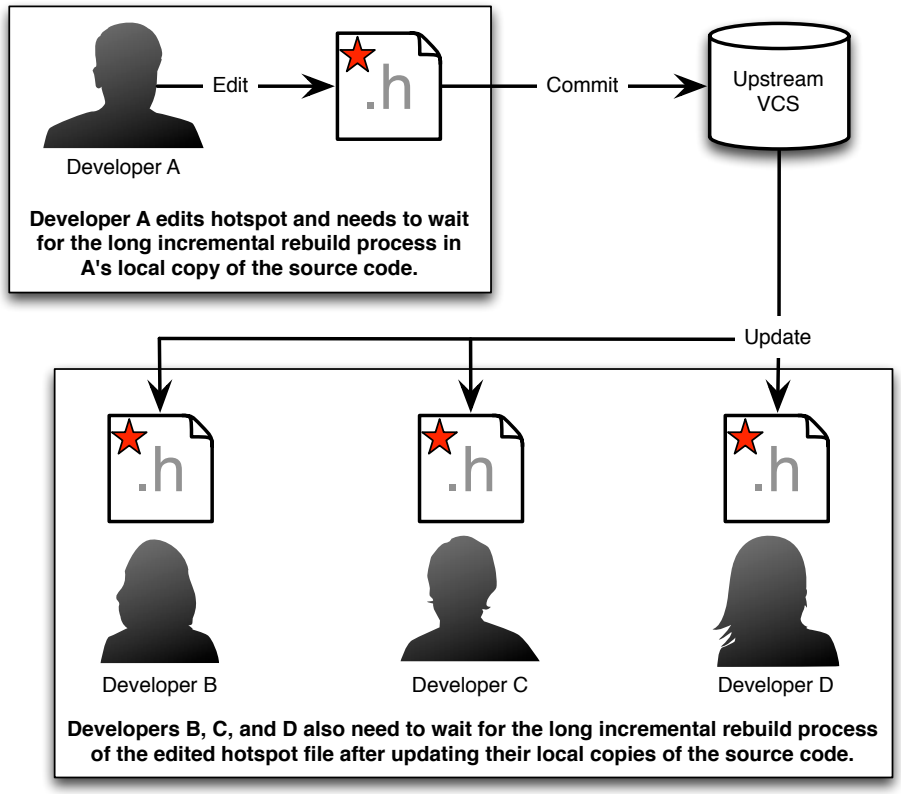



Figure 7.2: [Empirical Study 4] An example scenario of the impact that a header file hotspot can have on a development team.

to have most of their build performance issues with files that we measured to be relatively fast to rebuild. When asked why they did not select the slower files, one GLib developer responded: "because none of these [files] change often."

At first glance, this insight might seem counterintuitive. However, consider the scenario depicted in Figure 7.2 with a build hotspot and a team of four developers: A, B, C, and D. First, changing the hotspot file impacts the original developer. For example, if developer A modifies H, the change would trigger the slow rebuild process of H in A's copy of the source code. Next, the change to the hotspot impacts other team members. When developers B, C, and D update their copies of the source code and receive

A's change to H, it will also trigger the slow rebuild process of H on their machines.

Based on this insight, this chapter analyzes whether the build hotspots (i.e., files that not only trigger long rebuild processes, but also tend to change frequently) are better indicators of files that will slow the rebuild process in the future, and hence should be optimized now to save developers time. In order to understand how such reduction of rebuild cost can be achieved, we use logistic regression models to study actionable factors that impact build hotspot likelihood. Such factors correspond to common source code (e.g., file fan-in) and code layout properties (e.g., the subsystem that a file belongs to). Of course, making fewer changes to the code is not a feasible option for reducing build activity, since after all, the software needs to evolve to implement changing requirements.

## 7.3 Hotspot Analysis Approach

In order to identify build hotspots, we analyze the Build Dependency Graph (BDG) and the change history of a software system. Figure 7.3 provides an overview of our approach, which is divided into the three steps that are described below. In this section, we describe our approach in abstract terms, while details of the prototype implementation used in our case studies are provided in Section 7.4.

### 7.3.1 Dependency Graph Construction

We first extract the build dependency graph of the main build target of a software system (e.g., `all` in Figure 7.1a), which is a directed acyclic graph $BDG = (T, D)$ with the following properties:

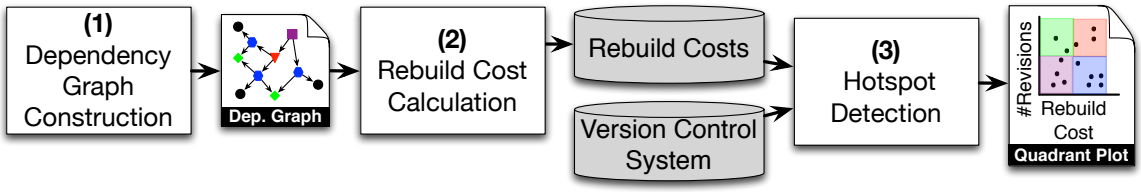



Figure 7.3: [Empirical Study 4] Overview of our hotspot analysis approach.

- Graph nodes represent build targets $T = T_f \cup T_p$, where $T_f$ is the set of concrete files produced or consumed by the build process, $T_p$ is the set of phony targets in the build process, and $T_f \cap T_p = \emptyset$.

- Directed edges denote dependencies $d(t, t') \in D$ from target $t$ to target $t'$. A dependency exists between targets $t$ and $t'$ if $t$ must be updated when $t'$ changes. Figure 7.1a shows an example BDG.

### 7.3.2 Rebuild Cost Calculation

In order to calculate the rebuild cost of a source file, we build a cost map $CM = (D_r, C)$ with the following properties:

- The set of BDG dependencies $D = D_r \cup D_g$, where $D_r$ is the set of $d(t, t')$ with *recipes* (i.e., build commands that must execute in order to update $t$ when $t'$ changes), $D_g$ is the set of $d(t, t')$ used to order dependencies (i.e., dependencies without recipes), and $D_r \cap D_g = \emptyset$.

- There is a cost $C(d(t, t'))$ associated with each $d(t, t') \in D_r$, which is used to give a weight to each directed edge. This cost may be measured in terms of number of triggered commands, elapsed time, etc.

- $CM$ contains an entry that maps each $d(t, t') \in D_r$ to its cost $C(d(t, t'))$.

The rebuild cost of a source file then is calculated by combining the file's dependencies in the BDG with the edge costs from the CM. The process is split into four steps as described below.

#### 7.3.2.1 Detect Source Files

Using the BDG, we detect the set of source files $S = \{s \in T_f \mid |\mathit{in}(s)| > 0 \wedge |\mathit{out}(s)| = 0\}$, where $\mathit{in}(s) = \{d(t, s) \in D\}$ (i.e., dependencies that must be regenerated when $s$ changes) and $\mathit{out}(s) = \{d(s, t) \in D\}$ (i.e., dependencies that regenerate $s$), and $|X|$ is the cardinality of the set $X$. In other words, $S$ is the set of non-generated files (no outgoing edges) that are the initial inputs for the main build target.

#### 7.3.2.2 Detect Triggered Edges

For each source file node $s \in S$, we identify the set of edges $E(s)$ that will be triggered should $s$ change by selecting all edges that transitively depend on $s$ in the $BDG$. In other words, we perform a transitive closure of $d(s, t)$ on the $BDG$, and filter away edges that are not present in the $BDG$.

#### 7.3.2.3 Filter Duplicate Edges

Since the same recipe may be attached to multiple outgoing edges of a given build target $t$, we count each such recipe only once by filtering out all but one of the corresponding edges $d(t, t')$ from $E(s)$. We apply this filter to all dependencies $d(t, t') \in E(s)$ to obtain $E'(s)$.

For example, Figure 7.1a shows that when either `util1.o` or `util2.o` is updated, `library.a` must be re-archived. The `make` implementation in Figure A.1a shows that in such a case, the re-archiving of `library.a` only needs to be performed once. In this case, we would filter the edge between `library.a` and `util2.o` out of $E(s)$ to obtain $E'(s)$.

#### 7.3.2.4 Aggregate Cost of Triggered Edges

Finally, to calculate the rebuild cost of a source file $s$, we begin by looking up each edge $d(t,t') \in E'(s)$ in the CM. Any edge that appears in $E'(s)$, but does not appear in CM (e.g., $d(t,t') \in D_g$) is assumed to have no cost. The rebuild cost is then calculated by summing up the costs of the edges in $E'(s)$ that were found in the CM.

### 7.3.3 Hotspot Detection

Software systems evolve through continual change in the source code, build system, and other artifacts. Changes to files are logged in a Version Control System (VCS), such as Git. To identify hotspots, we need to calculate the rate of change of each source file, i.e., the number of revisions of the file that are recorded in the VCS, then plot this against the rebuild cost for each file. Similar to Khomh *et al.* [53], we divide the plot into four quadrants:

**Inactive** — Files that rarely change and that trigger quick rebuild processes. Optimizing the build for these files is unnecessary.

**High churn** — Files that frequently change, but trigger quick rebuild processes. These files are low-yield build optimization candidates because although they endure heavy maintenance, they do not cost much to rebuild.

Table 7.1: [Empirical Study 4] Characteristics of the studied systems. For each studied system, we extract two years of historical data just prior to the release dates.

| | **GLib** | **PostgreSQL** | **Qt** | **Ruby** |
|---|---|---|---|---|
| Domain | Development library | DBMS | UI framework | Programming language |
| Build Technology | Autotools | Autoconf, `make` | QMake | Autoconf, `make` |
| Version | 2.36.0 | 9.2.4 | 5.0.2 | 1.9.3 |
| Release Date | 2013-03-25 | 2013-04-04 | 2013-07-03 | 2011-10-31 |
| System Size (kSLOC) | 401 | 658 | 5,132 | 1,098 |
| # BDG Nodes | 3,375 | 4,637 | 38,235 | 1,560 |
| # BDG Edges | 121,710 | 59,676 | 2,752,226 | 6,240 |

**Slow build** — Files that rarely change, but trigger slow rebuild processes. These files are low-yield build optimization candidates.

**Hotspot** — Files that frequently change and trigger slow rebuild processes. These files are high-yield build optimization candidates.

The quadrant thresholds can be dynamically configured to suit the needs of the development team. Initially, thresholds may be selected using intuition, however later on, nonfunctional requirements could specify a maximum rebuild cost according to a system's common rate of file change.

## 7.4 Empirical Study Design

We perform a case study on four open source systems in order to: (1) evaluate our build hotspot detection approach, and (2) study the characteristics of real-world build hotspots. Hence, our case study is divided into two sections accordingly, which we motivate below:

**Evaluation of our hotspot detection approach (Section 7.5)** — Since rebuild cost, rate of change, and impact on other files individually can also be used to prioritize

files for build optimization, we want to evaluate whether the hotspot heuristic truly identifies the most costly files.

**Analysis of build hotspot characteristics (Section 7.6)** — Since code changes are required to address defects or add new features, one cannot simply avoid changing the code. Instead, build optimization effort must focus on controllable properties that influence build hotspot likelihood. Hence, we set out to study the relationship between controllable source file properties and hotspot likelihood.

The remainder of this section introduces the studied systems, provides additional detail about our implementation of the hotspot detection approach proposed in Section 7.3, and compares the build performance of header files to other files in the studied systems.

### 7.4.1 Studied Systems

We select four long-lived, rapidly evolving open source systems in order to perform our case study. We select systems of different sizes and domains to combat potential bias in our conclusions. Table 7.1 provides an overview of the studied systems.

*GLib* is a core library used in several GNOME applications.[5] *PostgreSQL* is an object-relational database system.[6] *Qt* is a cross-platform application and user interface framework whose development is supported by the Digia corporation, however welcomes contributions from the community-at-large.[7] Ruby is an open source programming language.[8]

[5] https://developer.gnome.org/glib/
[6] http://www.postgresql.org/
[7] http://qt.digia.com/
[8] https://www.ruby-lang.org/

The studied systems use different build technologies (e.g., GNU Autotools and QMake). However, each studied build technology eventually generates `make` specifications from higher level build specifications. The choice of studying `make`-based build systems is not a coincidence, since such build systems are the de facto standard for C/C++-based software projects (*cf.* Chapter 4), which are the projects that typically use header files.

### 7.4.2 Implementation Details

#### 7.4.2.1 Dependency Graph Construction and Rebuild Cost Calculation

We first perform a full build of each studied system on the Linux x64 platform with GNU `make` tracing enabled to generate the necessary trace logs. Such a trace log carefully records all of the decisions made by the build tool (e.g., is input file X newer than output file Y?). The generated trace is then fed to the MAKAO tool [3], which parses it to produce the BDG and CM. Finally, we implemented the four steps of Section 7.3.2 in a script and applied it to the BDG and CM to calculate the rebuild cost of each source code file $s \in S$.

#### 7.4.2.2 Edge Weight Metric

To give the edge weighing function $C(d(t,t'))$ a meaningful concrete value, we use *elapsed time*, i.e., the time spent executing build recipes. For this, we measure the time consumed by each recipe during a full build by instrumenting the shell spawned by the build tool for each recipe's execution. Since varying load on our experimental machines may influence the elapsed time measurements, we repeated the full build process (from scratch) ten times and select the median elapsed time for each recipe.

After ten repetitions, we find that the standard deviation of the elapsed time for

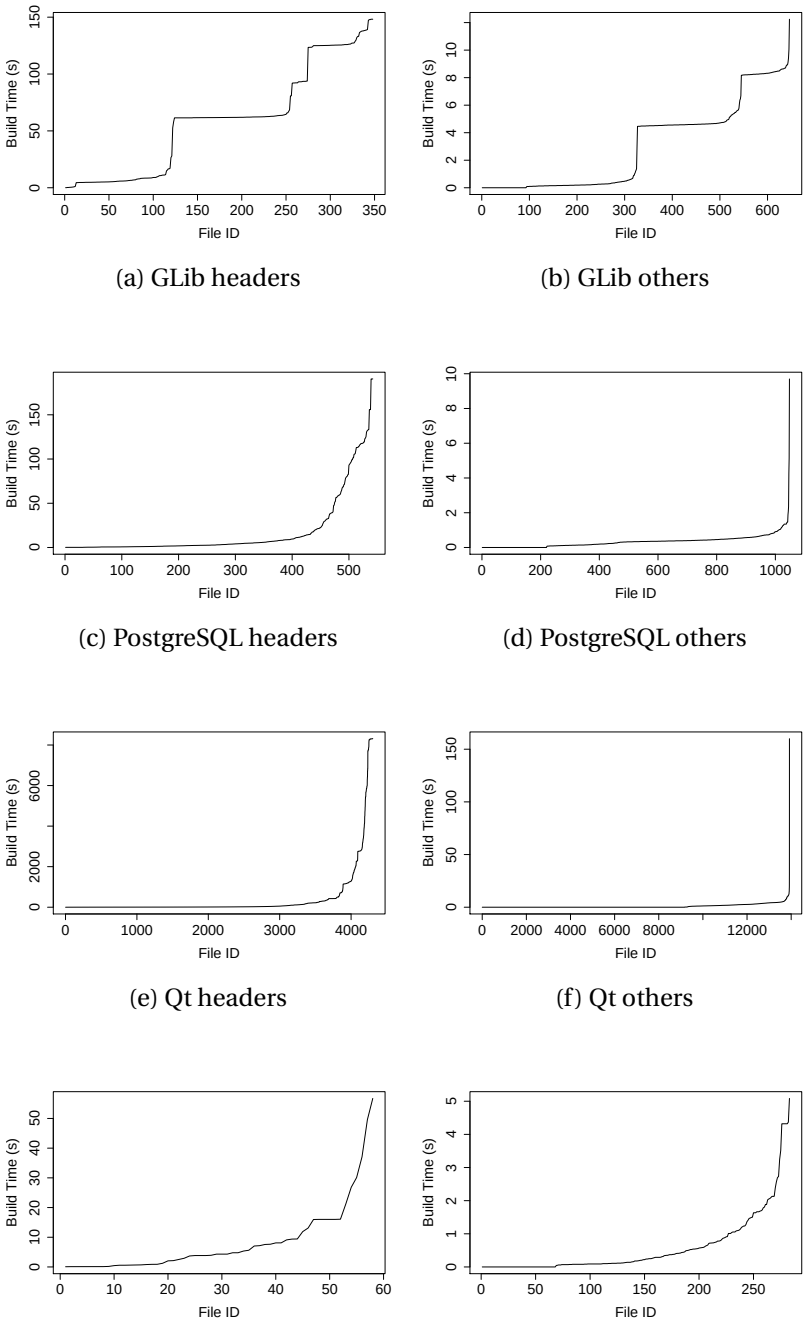
any given command does not exceed 0.5 seconds and the median standard deviation among the ten repetitions does not exceed 0.02 seconds. Thus, the variability in the elapsed time consumed by a recipe will not skew our results severely.

#### 7.4.2.3 Quadrant Threshold Selection

For the purposes of our case study, we use 90 seconds as the threshold for rebuild cost, since Humble and Farley suggest this as an upper-bound on the time spent on an incremental build [45]. For the rate of change threshold, we select the median number of revisions across all files of a system. Furthermore, to reduce the impact that outliers have on the quadrant plots, we apply the logarithm on both rebuild cost and rate of change values. We normalize rebuild cost and rate of change by dividing each logarithmic value by the maximum so that the quadrant plots of different systems can be compared.

### 7.4.3 Preliminary Analysis of Header File Build Performance

Prior to performing our case studies, we first perform a preliminary analysis to evaluate whether header files are the source of the most problematic build hotspots in the studied systems. Indeed, while prior work has focused on header file optimization [24, 111, 112], it is unclear whether they are truly the largest build hotspots. Since header files represent interfaces (which ought to be more stable over time), they may not necessarily change as frequently as regular source code files. It is conceivable that core implementation files that change often and generate a substantial amount of link-time build activity may also be hotspots that are worthy of optimization effort [61].



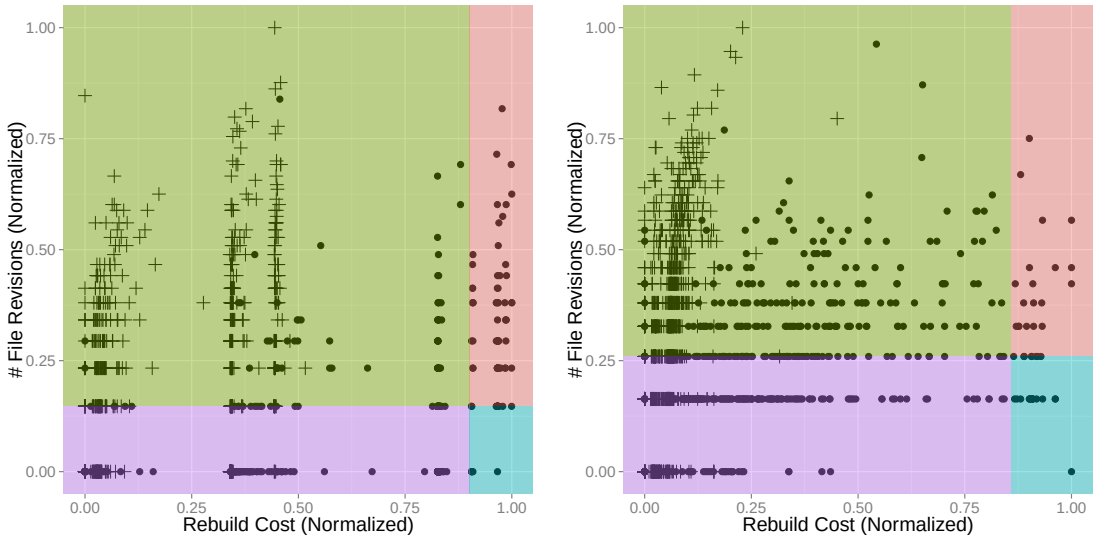

(a) GLib headers (b) GLib others

(c) PostgreSQL headers (d) PostgreSQL others

(e) Qt headers (f) Qt others

(g) Ruby headers (h) Ruby others

Figure 7.4: [Empirical Study 4] The rebuild cost of the header and other (primarily source) files in the studied systems.




(a) GLib

(b) PostgreSQL

(c) Qt

(d) Ruby

Figure 7.5: [Empirical Study 4] Quadrant plot of rate of change and rebuild cost. Hotspots are shown in the top-right (red) quadrant. The shaded circles indicate header files, while plus (+) symbols indicate non-header files. Non-header file hotspots are circled in red.

#### 7.4.3.1 Approach

Figure 7.4 plots the rebuild cost of each source file $s \in S$ in increasing order for each of the studied systems. The figures in the column on the left show the rebuild costs of header files, while the figures in the column on the right show the rebuild costs of the other source files in each of the studied systems. In addition, we show quadrant plots of the rebuild costs versus the number of revisions of each source file $s \in S$ in Figure 7.5.

#### 7.4.3.2 Results

**Figure 7.4 shows that, as expected, almost all header files trigger a longer rebuild process than other file types do.** This is primarily because when a header file is changed, all files that `#include` it must be recompiled. The majority of GLib header files trigger rebuild processes of more than 60 seconds (Figure 7.4a). Several Qt header files trigger rebuild processes of more than 15 minutes (900 seconds), with extreme cases reaching over two hours (Figure 7.4e). In all of the studied systems, the median rebuild cost for header files is at least 10 times larger than the median rebuild cost for the other types of files. Our findings support the argument of Yu *et al.* [111], that (false) dependencies in header files can indeed substantially slow down the build process.

Interestingly, header files are not the only source of spikes in rebuild cost. Figures 7.4b, 7.4f and 7.4h show that a small set of other files can trigger rebuild processes of several seconds. Many of these files are `.c` files, for which one would normally expect that several subsequent linker commands may be triggered by updating the object code, however only one compile command should be triggered.

Deeper inspection of the GLib system shows that 89 `.c` files in GLib in fact trigger

multiple compile commands. We found that 1 of the 89 `.c` files is imported through the preprocessor into several `.c` files, similar to a header file. Hence, changes to the imported `.c` file trigger compile commands for each `.c` file that includes the file. Another 4 of the 89 multi-compiling `.c` files contain test code that is linked into several test executables. However, each test binary requires the object code of the common files to be generated with different compiler flag settings, which means that the same `.c` file must be compiled once for each compiler flag setting. The remaining 84 of the 89 multi-compiling `.c` files are used to implement a source code generator. The generator produces code that is linked to several test executables. When any of the code generator source files are changed, the tool must be rebuilt, then the generated code must be reproduced, recompiled, and re-linked to the test executables. The GLib code generator is an example of a "build code robot," as was identified for GCC by Tu and Godfrey [104].

Figures 7.4g and 7.4h show that the Ruby project has no file that exceeds the 90-second threshold that we selected for header file hotspots. This is likely due to the size of the system and its build dependency graph, which, as shown in Table 7.1, has almost an order of magnitude fewer edges than the next smallest system (PostgreSQL). Although Ruby may be free of 90-second header file hotspots, developers of such a small system may be accustomed to a very quick rebuild cycle, and may have a lower threshold for frustration. In a study of time delay, Fischer *et al.* find that user satisfaction degrades linearly as delay increases from 0-10 seconds [36]. We, therefore, set the threshold for Ruby hotspots to 10 seconds for the remainder of the chapter.

**Non-header files do not generate enough build activity to be of concern for hotspot detection.** Figure 7.5 shows the source files that land in each of the four quadrants.

Header files are plotted using shaded circles, while other files are plotted using plus (+) signs. Files that land on quadrant borders are conservatively mapped to the lower quadrant.

The quadrant plots in Figure 7.5 show that only two non-header file appear in the hotspot quadrant. The first is a Bison grammar file `parse.y` in the Ruby system. Changing the grammar file causes both an implementation and a header file to be regenerated, which in turn triggers several recompilations. The second non-header file hotspot is a Qt file `qtdeclarative/tests/auto/shared/util.cpp`, which contains the testing utility code that causes several test binaries to be re-linked. Although each change to the test utility implementation triggers a rebuild process of 159 seconds, Figure 7.4e shows that there are several Qt header files that, when they change, trigger rebuild processes that take hours.

*Although some implementation files take a long time to rebuild, the median rebuild cost for header files is at least ten times larger than the median rebuild cost for the other types of files. While our hotspot detection approach is generic enough to be applied to any source file, since the overwhelming majority of hotspots are header files, the remainder of this chapter focuses on header file hotspots.*

## 7.5 Evaluation of the Hotspot Detection Approach

Since we find in Section 7.4.3 that header files dominate the hotspot quadrant for the studied systems, we focus on *header file hotspots* throughout the remainder of the chapter.

While our quadrant plots can identify header file hotspots, it is not clear whether build optimization effort that is focused on such hotspots would yield a larger reduction in future build cost than other hotspot detection approaches. In this section, we

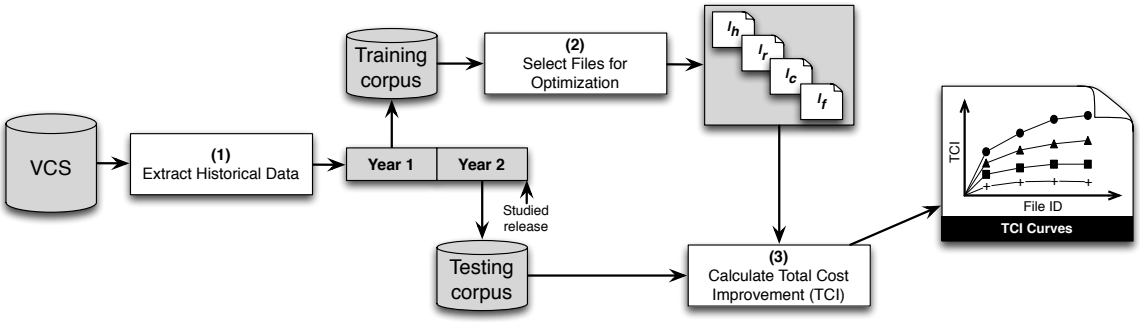



Figure 7.6: [Empirical Study 4] Overview of our simulation exercise.

discuss a case study that we have performed in order to evaluate our hotspot heuristic.

### 7.5.1 Approach

In order to evaluate our hotspot heuristic, we compare the decrease in future rebuild cost when prioritizing files for build optimization using the header file hotspot heuristic to the decrease in future rebuild cost when using heuristics based on the following:

**Individual rebuild cost** — Header files that trigger slow rebuild processes are likely to be costly.

**Rate of change** — Header files that are changing frequently are likely to be costly.

**File fan-in** — Prior header file optimization approaches focus on header files that have the highest file fan-in [111, 112], i.e., number of modules that use the functionality defined or declared in the header. These approaches implicitly assume that header files with the highest file fan-in trigger the most build activity.

In order to perform this comparison, we perform a simulation exercise using two years of historical data from each studied system, which we divide into training and

testing corpora. Figure 7.6 provides an overview of the steps in our simulation exercise. We describe each step in the simulation below.

#### 7.5.1.1 Extract Historical Data

We allocate the year of historical data just prior to the studied releases shown in Table 7.1 to the testing corpus. Then, we allocate the year prior to the testing corpus to the training corpus. We do so because we want to evaluate the approaches on a time period where we are sure that developers would be rebuilding the system frequently. The period leading up to a release is guaranteed to require frequent rebuilds due to active development.

The build cost and build changes are calculated differently. Build change is measured twice — once in the training corpus, and again in the testing corpus. This makes the data representative for changes in each corpus. On the other hand, we measure rebuild cost on the actual release instead of on an intermediate version released in between the training and testing period. We do so out of convenience, since measuring rebuild cost and extracting the build dependency graph requires a buildable version of the whole system, whereas intermediate versions more often than not are broken in several subsystems (especially in large systems like Qt). Although our evaluation hence combines rebuild cost measurements with change data that is one year older, in practice the mismatch is quite limited, as shown by our results.

#### 7.5.1.2 Select Files for Optimization

For each of the four approaches, we identify the top $N$ header files that should be optimized for build speed based on analysis of the training corpus. Depending on the

heuristic, the top $N$ corresponds to the header files that occupy the hotspot quadrant ($l_h$), or the $N$ files with the highest individual rebuild cost ($l_r$), rate of change ($l_c$), or file fan-in ($l_f$).

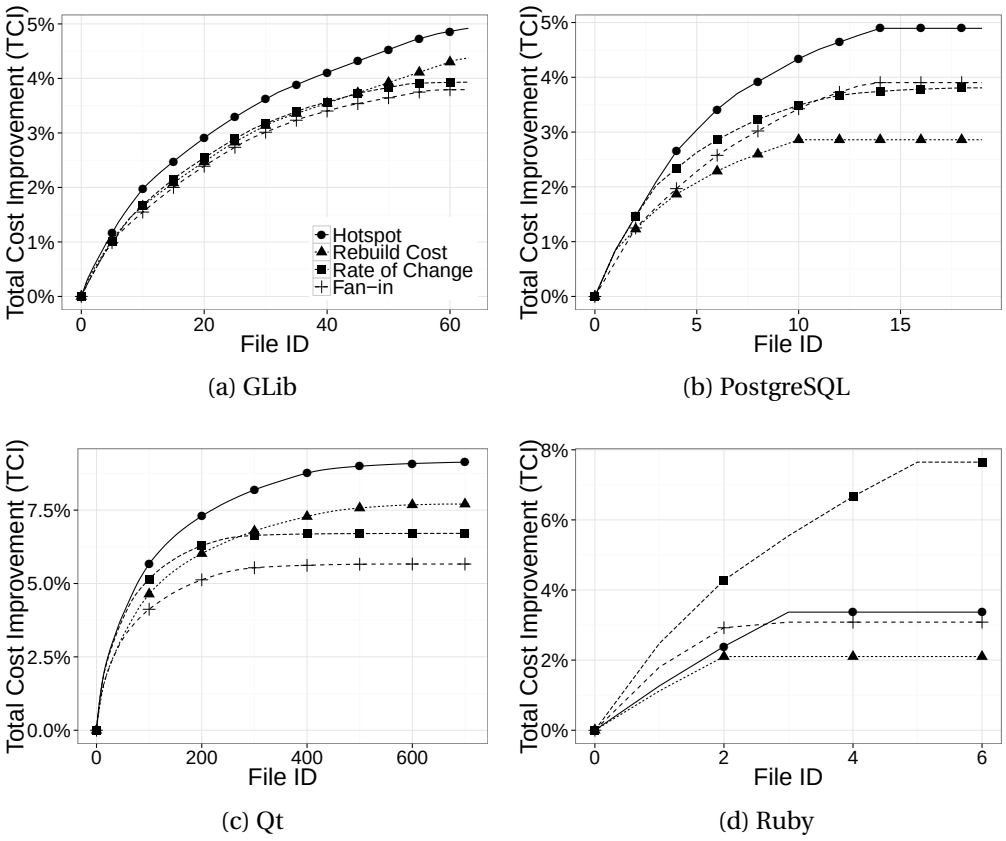
#### 7.5.1.3 Calculate Total Cost Improvement (TCI)

To evaluate the impact of improving the top $N$ header files in the testing corpus suggested by a particular heuristic calculated in the training corpus, we calculate the *Total Cost Improvement* (TCI). This is the percentage of reduction in the *Total Rebuild Cost* (TRC, i.e., the sum of the rebuild cost of all header files $h$ and all changes in the testing corpus) that would be achieved when replaying all required builds of the testing corpus if the individual rebuild costs of each header file in $l_h$, $l_r$, $l_c$, or $l_f$ (i.e., the top $N$ header files of each of the four approaches) were reduced by 10%[9] prior to entering the testing corpus.

For example, we first calculate the total rebuild cost in the testing corpus, and then, if a particular heuristic suggests that we optimize header files A, B, and C based on the training corpus, we recalculate the hypothetical total rebuild cost assuming that the individual rebuild costs of header files A, B, and C were reduced by 10%. Then, the TCI for the heuristic is calculated as: $TCI = \frac{TRC_{actual} - TRC_{hypothetical}}{TRC_{actual}}$. Note that in this chapter, we consider a reduction of 10% in individual build cost, independent of specific approaches (e.g., refactorings) that can be used to obtain a 10% reduction [24].

Figure 7.7 compares the TCI of $l_h$, $l_r$, $l_c$, and $l_f$ for each of the studied systems. These curves are cumulative, meaning that, for instance, the TCI shown for file #2 in each curve is the TCI in which the top two files (#1 and #2) have been improved. To determine the value of $N$, we select $N$ header files such that $l_h$ contains a list of all of the header

[9] The simulation was repeated for 20% and 50% improvements, which yielded similar results.




(a) GLib (b) PostgreSQL

(c) Qt (d) Ruby

Figure 7.7: [Empirical Study 4] Cumulative curves comparing the four approaches for selecting header files for build performance optimization. The Total Cost Improvement (TCI) measures the reduction of time spent rebuilding in the future (testing corpus) when the performance of the selected header files are improved by 10%.

files that occupy the hotspot quadrant.

### 7.5.2 Results

**The TCI of the hotspot heuristic exceeds the TCI of files with the highest individual rebuild cost, rate of change, or file fan-in.** Indeed, Figure 7.7 shows that the TCI of $l_h$ exceeds those of $l_r$, $l_c$ and $l_f$ in three of the four studied systems. In the Ruby system,

simply ordering files by their change frequency yields the highest TCI. Individual rebuild cost does not help to select the files that will yield a high TCI because most Ruby header files rebuild quickly (*cf.* Section 7.4).

We performed a Kruskal-Wallis rank sum test to check whether there is a statistically significant difference between the TCI values of $l_h$, $l_r$, $l_f$ and $l_c$ ($\alpha = 0.05$). The test results for the GLib, PostgreSQL, and Qt systems show that the differences in TCI are significant ($p \ll 0.05$), indicating that the hotspot analysis selects more costly header files than just considering the file fan-in, the individual rebuild cost, or the rate of change of a header file separately. In the case of Ruby, a Wilcoxon signed rank test indicates that the hotspot analysis ($l_h$) performs significantly worse than the rate of change ($l_c$).

**Figure 7.7 shows that Qt achieves the largest TCI rates, with $l_h$ reaching a peak of 9.3% and $l_r$ reaching 8.7%.** Note that TCI values in this simulation are theoretically constrained to a maximum of 10%, since this is the amount that the individual rebuild cost of the selected header files were improved by. Moreover, TCI values of 8.7% and 9.3% equate to a total rebuild cost savings of 8.4 and 8.9 days respectively — on average, a savings of 3.3 and 3.7 minutes of rebuild cost per change. GLib and PostgreSQL both achieve maximum TCI values of 4.9%, which equate to a savings of 49.0 and 7.4 minutes, or 4.0 and 1.4 seconds per change respectively. Although Ruby achieves a high TCI value of 7.6% by optimizing the most frequently changing files, due to the rapid speed of the Ruby build process, this only equates to a total savings of 38.7 seconds or 0.16 seconds per change. Large and complex systems like Qt can really lower rebuild costs with carefully focused build optimization.

*The hotspot heuristic tends to select more costly header files that yield a higher total cost improvement than other header file selection heuristics, especially in larger systems with more complex build dependency graphs.*

### 7.5.3 Discussion

It is important to note that TCI is an under-approximation for the true total rebuild cost for two reasons. First, TCI assumes that each change will only be rebuilt once, when in reality, a build will run several times by the developer making the change, and by the other developers of the system. Since the number of times a build was executed for a particular change is not typically recorded, we assume the minimum case (i.e., just once). Second, TCI assumes that rebuild cost improvements to header files are independent, i.e., by improving the rebuild cost of a header file A, we do not improve the rebuild cost of another header file B in our simulation. In reality, due to overlapping dependencies, this assumption likely will not hold. In both cases, our reported TCI values correspond to a lower bound of the actual TCI.

## 7.6 Hotspot Characteristic Analysis

To help practitioners avoid creating header file hotspots or find opportunities for refactoring, we analyze whether header file properties offer any explanatory power for hotspot likelihood. To do so, we build logistic regression models and measure the effect that each property has on hotspot likelihood.

When selecting explanatory variables for our models, we discarded change frequency, since it is not an actionable factor, i.e., changes that fix defects and add features cannot be avoided. Instead, build optimization effort must focus on reducing the rebuild cost of a header file by either: (1) shrinking the set of triggered edges $E'(s)$, or (2) reducing the complexity of the header file itself (to reduce its individual compilation time). The latter suggests that header file size and complexity metrics should be added to our

models, while the former suggests that we should consider code layout as well.

### 7.6.1 Approach

We build *logistic regression models* to check for a relationship between header file properties and hotspot likelihood. A logistic regression model will predict a binary outcome variable (whether or not a header file appears in the hotspot quadrant of Figure 7.5) based on the values of a given set of explanatory variables.

Table 7.2 lists the code content and layout properties that we considered, and provides our rationale for selecting them. Each metric is measured using the released versions of the studied systems presented in Table 7.1. Code layout metrics are derived from the pathname of each source file. Directory and file fan-in are calculated based on code dependency information extracted using the Understand static code analysis tool.[10] Source lines of code are counted using the SLOCCount tool.[11] Number of includes is calculated using common UNIX tools to select `#include` lines that refer to files within each software system.

Since our goal is to understand the relationship between the explanatory variables (code layout and content properties) and the dependent variable (hotspot likelihood), which is similar to prior work of Mockus *et al.* [78] and others [20, 97], we adopt a similar model construction approach. Moreover, since we do not intend to use the models for prediction, but rather for explanation, we do not split the datasets into training and testing corpora as was done in Section 7.5, but rather build models using both years of the historical data together. To lessen the impact of skew, we log-transform the SLOC, number of includes, depth, and file and directory fan-in variables. We build models for

[10] http://www.scitools.com/index.php
[11] http://www.dwheeler.com/sloccount/

Table 7.2: [Empirical Study 4] Source code properties used to build logistic regression models that explain header file hotspot likelihood.

| | **Property** | **Description** | **Rationale** |
|---|---|---|---|
| Code layout | Subsystem | The top-level directory in the path of a header file. | Certain subsystems have a more central role and thus may be more susceptible to header file hotspots. |
| | Depth | The number of subdirectories in the header file's path relative to the top directory of the system. | Since header files that appear in deeper directory paths are likely more specialized and hence impact fewer deliverables than header files at shallower directory paths, they likely have a smaller impact on the build process, and are thus less likely to be hotspots. |
| | Directory fan-in | The number of other directories whose source files refer to code within this header file. | Header files with code dependencies that are more broadly used throughout the codebase are likely to have a higher rebuild cost, and thus are more likely to be hotspots. |
| Code content | File fan-in | The number of other source files referring to code within this header file. | The more source files that rely on a header file, the more likely it is to be a hotspot. |
| | Source lines of code | Non-whitespace, non-comment lines. | Larger header files are more likely hotspots. |
| | Number of includes | The number of distinct imported interface files, i.e., `#include` statements. | Yu *et al.* suggest that including several interface files bloats the build process [111]. Hence, we suspect that importing many other header files will increase hotspot likelihood. |

each studied system separately.

#### 7.6.1.1 Data Preparation and Model Construction

We check for variables that are highly correlated with one another prior to building our models, and also check for variables that introduce multicollinearity into preliminary models. We use Spearman rank correlation ($\rho$) to check for highly correlated variables instead of other types of correlation (e.g., Pearson) because rank correlation does not require normally distributed data. After constructing preliminary models, we check them for multicollinearity using the Variance Inflation Factor (VIF) score. A VIF score is calculated for each explanatory variable used by a preliminary model. A VIF score of one indicates that no collinearity is introduced by the variable, while values greater than one indicate the ratio of inflation in the variance explained due to collinearity. As suggested by Fox [37], we select a VIF score threshold of five. Neither correlation nor VIF analysis identified any variables that are problematic for our models ($|\rho| \geq 0.6$ or $\mathit{VIF} \geq 5$).

Finally, to decide whether an explanatory variable is a significant contributor to the fit of our models, we perform drop-one tests [21] using the implementation provided by the core `stats` package of R [91]. The test measures the impact of an explanatory variable on the model by measuring the *deviance explained* (i.e., the percentage of deviance that the model covers) of models consisting of: (1) all explanatory variables (the full model), and (2) all explanatory variables except for the one under test (the dropped model). A $\chi^2$ likelihood ratio test is applied to the resulting values to indicate whether each explanatory variable improves the deviance explained by the full model to a statistically significant degree ($\alpha = 0.05$). We discard those variables that do not improve

the deviance explained by a significant amount.

#### 7.6.1.2 Model Analysis

In order to compare the effect that each statistically significant header file property has on hotspot likelihood, we extend the approach of Cataldo *et al.* [20]. First, a typical header file is imitated by setting all explanatory numeric variables to their median values and categorical/binary values to their mode (most frequently occurring) category. The model is then applied to the typical header file to calculate its *predicted probability*, i.e., the likelihood that the typical header file is a hotspot, which we call the Standard Median Model (SMM). One by one, we then modify each explanatory variable of the typical header file in one of two ways:

**Numeric variable** — We add one standard deviation to the median value and recalculate the predicted probability.

**Categorical/binary variable** — The predicted probability is recalculated for each category except for the mode.

The recalculated predicted probabilities are referred to as the Increased Median Model (IMM) values. Note that the SMM is a fixed value while IMM is calculated for each explanatory variable. The Effect Size $ES$ of an explanatory variable $X$ is then calculated as:

$$ES(X) = \frac{IMM(X) - SMM}{SMM} \tag{7.1}$$

Variables can have positive or negative *ES* values indicating an increase or decrease in hotspot likelihood relative to SMM respectively. The farther the value of $ES(X)$ is

Table 7.3: [Empirical Study 4] Logistic regression model statistics for the larger studied systems (i.e., GLib and Qt). Deviance Explained (DE) indicates how well the model explains the build hotspot data. $\Delta DE$ measures the impact of dropping a variable from the model, while $ES(X)$ measures the effect size (see equation 7.1), i.e., the impact of explanatory variables on model prediction.

| **GLib** | | | | **Qt** | | | |
|---|---|---|---|---|---|---|---|
| Deviance explained | | 57% | | Deviance explained | | 49% | |
| Metric | Subdir. | $\Delta DE$ | $ES(X)$ | Metric | Subdir. | $\Delta DE$ | $ES(X)$ |
| Subsystem | gio | 28%$^{***}$ | † | Subsystem | qtwebkit | 23%$^{***}$ | † |
| | tests | | 2.91 | | qtimageformats | | -0.69 |
| | gmodule | | 4.73 | | qtactiveqt | | -0.52 |
| | build | | 49.94 | | qtsvg | | -0.07 |
| | ./ | | 158.62 | | qtdoc | | 0.43 |
| | gobject | | >1,000 | | qtgraphicaleffects | | 0.43 |
| | glib | | >1,000 | | qtscript (+7 others) | | >1,000 |
| Depth | | $\diamond$ | | Depth | | 5%$^{***}$ | -0.58 |
| Directory fan-in | | 1%$^{*}$ | 4.11 | Directory fan-in | | 1%$^{***}$ | 0.98 |
| File fan-in | | 5%$^{***}$ | 1.10 | File fan-in | | 13%$^{***}$ | 2.53 |
| SLOC | | 1%$^{*}$ | 0.39 | SLOC | | 1%$^{***}$ | 0.44 |
| Includes | | $\diamond$ | | Includes | | 1%$^{***}$ | -1.0 |

† Mode values of categorical variables are part of the SMM calculation and hence cannot be calculated using IMM.
Statistical significance of explanatory power according to Drop One analysis:
$\diamond\ p \geq 0.05$; $^{*}\ p < 0.05$; $^{**}\ p < 0.01$; $^{***}\ p < 0.001$

from zero, the larger the impact that $X$ has on our models. For example, an $ES$ value of 0.51 means that the IMM is 51% higher than the SMM.

### 7.6.2 Results

**Tables 7.3 and 7.4 shows that our complete models achieve between 32% (Ruby) and 57% (GLib) deviance explained.** Our models could be likely improved by adding additional header file properties, or even properties extracted from the build system. However, we believe that these models provide a sound starting point for explaining header file hotspot likelihood.

Table 7.4: [Empirical Study 4] Logistic regression model statistics for the smaller studied systems (i.e., PostgreSQL and Ruby). Deviance Explained (DE) indicates how well the model explains the build hotspot data. $\Delta DE$ measures the impact of dropping a variable from the model, while $ES(X)$ measures the effect size (see equation 7.1), i.e., the impact of explanatory variables on model prediction.

| | **PostgreSQL** | | **Ruby** | |
|---|---|---|---|---|
| Deviance explained | 47% | | 32% | |
| Metric | $\Delta DE$ | $ES(X)$ | $\Delta DE$ | $ES(X)$ |
| Subsystem | ◇ | | ◇ | |
| Depth | ◇ | | ◇ | |
| Directory fan-in | ◇ | | ◇ | |
| File fan-in | 47%$^{***}$ | 8.75 | 32%$^{***}$ | 3.38 |
| SLOC | ◇ | | ◇ | |
| Includes | ◇ | | ◇ | |

† Mode values of categorical variables are part of the SMM calculation and hence cannot be calculated using IMM.
Statistical significance of explanatory power according to Drop One analysis:
◇ $p \geq 0.05$; $^{*}$ $p < 0.05$; $^{**}$ $p < 0.01$; $^{***}$ $p < 0.001$

Tables 7.3 and 7.4 also shows the change in deviance explained reported by the drop-one test ($\Delta DE$) for each variable. The $\Delta DE$ measures the percentage of the deviance explained by the model that can only be explained by a particular variable. Since multiple variables may explain the same header file hotspots, the $\Delta DE$ values do not sum up to the total deviance explained by the full model.

**In the larger studied systems, header file hotspots are more closely related to code layout than to the content of a file.** Table 7.3 shows that there is a drop in deviance explained of 28% and 23% in the GLib and Qt systems respectively when the subsystem explanatory variable is excluded from the model. Furthermore, although the impact is small, directory fan-in explains a statistically significant amount of deviance in the GLib and Qt systems. On the other hand, in those systems, file size and number of

includes offer little explanatory power, and although GLib and Qt models without file fan-in drop in explanatory power by 5% and 13% respectively, the decrease is smaller than that of the subsystem variable. Hence, most of the explanatory power of the GLib and Qt models is derived from code layout properties, such as the subsystem of a file.

Filesystem depth provides additional explanatory power (5%) in Qt, which is the largest studied system. The negative effect size indicates that as we move deeper into the filesystem hierarchy, hotspot likelihood decreases, suggesting that more central header files (located at shallower filesystem locations) are more prone to build performance issues.

**On the other hand, for the smaller systems, code layout properties offer little hotspot explanatory power.** Table 7.4 shows that the subsystem variable does not contribute a significant amount of explanatory power to the PostgreSQL and Ruby models. Moreover, despite the fact that 93% of the PostgreSQL hotspots reside in the `include` subsystem, this corresponds to 79% of all PostgreSQL header files, making this a less distinguishing variable.

Furthermore, filesystem depth is not a significant contributor to our PostgreSQL and Ruby models. The vast majority of hotspots in the PostgreSQL and Ruby systems are found in their `include` and top directory subsystems respectively, which do not have complex subdirectory structures within them. On the other hand, hotspots are more evenly distributed among subsystems in the Qt system, and hence the depth metric provides additional explanatory power there.

**File fan-in provides all of the explanatory power in the smaller studied systems.** Although file fan-in provides a significant amount of explanatory power to all of the

models, file fan-in impacts the performance of the smaller PostgreSQL and Ruby system models the most. File fan-in calculates the source files that are directly depending on the functionality provided within the header file. In this sense, file fan-in provides an optimistic (minimal) perspective of dependencies among code files. In smaller systems, this optimistic perspective is sufficient, whereas in larger systems with many subsystems (and a complex interplay between them) where architectural decay has introduced false dependencies among files [111], file fan-in no longer accurately estimates the actual set of build dependencies. Spearman rank correlation tests indicate that there is a stronger correlation between the file fan-in and individual rebuild cost of header files in the smaller studied systems ($\rho_{PostgreSQL} = 0.87, \rho_{Ruby} = 0.62$) than in the larger ones ($\rho_{Qt} = 0.28, \rho_{GLib} = 0.48$). Indeed, in larger systems with a more complex interplay between system components, most of the files including a header file will likely be other header files. Since this additional layer of header file indirection introduces an additional set of unpredictable, but necessary compile dependencies, file fan-in by itself tends to lose its hotspot explanatory power as systems grow.

Yet, even in the smaller PostgreSQL and Ruby systems where file fan-in provides all of the explanatory power, it does not provide a highly accurate estimate of hotspot likelihood. The reason for this discrepancy is twofold. First, while file fan-in provides an optimistic estimate of the compile commands that would be required should the header file change, it can offer little information about the link commands that would be required. For example, two header files with an identical amount of file fan-in may trigger very different amounts of link activity, which would greatly impact the rebuild cost of the header files. Second, being a metric that is calculated based at the code level, file fan-in cannot estimate how frequently a header file will change.

*Our models explain 32%-57% of identified hotspots. Architectural properties offer much of the explanatory power in the larger systems, suggesting that as systems grow, header file hotspots have more to do with code layout than with code content properties like file fan-in.*

### 7.6.3 Discussion

The explanatory power of the code layout metrics indicates that there are areas of the larger studied systems that are especially susceptible to header file hotspots (Table 7.3). Although our regression models seem to suggest that one should simply redistribute header files from subsystems with high hotspot likelihood to the subsystems with lower hotspot likelihood, such a course of action is impractical. Instead, our results should be interpreted as pinpointing the problematic subsystems that would benefit most from architectural refinement, such as a focused refactoring impetus. Moreover, the impact that code metrics have on our regression models suggests that optimization of the header files in the hotspot-prone subsystems will yield better results if the focus of such optimization is on the reduction of file fan-in, rather than of header file size or the number of includes.

## 7.7 Limitations and Threats to Validity

In this section, we discuss the limitations and the threats to the validity of our study.

### 7.7.1 Limitations

Our approach focuses on the detection and prioritization of header file hotspots, but does not suggest automatic hotspot refactorings. In this respect, our approach is similar to defect prediction, which is used to focus quality assurance effort on the most defect-prone modules. Furthermore, automatically proposing fixes for hotspots requires domain-specific expertise. For example, an automatically generated build dependency graph refactoring may fix hotspots in theory, but in practice may require an infeasibly complex restructuring of the system, reducing other desirable properties of a software system like understandability and maintainability. Further work is needed to find a balance between these forces.

An experienced developer may have an intuition about which header files are hotspots, but such a view would be coloured by his or her individual perspective. Moreover, our hotspot detection approach provides automated support to ground developers' intuition with data and through routine reapplication of our approach, a development team can monitor improvement or deterioration of hotspots over time.

### 7.7.2 Construct Validity

Since build systems evolve [4, 67], the BDG itself will change as a software system ages, which may cause the rebuild cost of each file to fluctuate. For simplicity sake, our simulation experiment in Section 7.5 projects a constant build cost for each change. Nonetheless, our technique is lightweight enough to recalculate rebuild costs after each change to the build system.

### 7.7.3 Internal Validity

Since one can only execute a build system for a concrete configuration, we only study a single configuration for each studied system. Unfortunately, once a target configuration is selected, areas of the code that are not related to the selected software features will not be exercised by the build process. For example, since we focus on the Linux environment, Windows-specific code will be omitted during the build process. A static analysis of build specifications, such as that of Tamrawi *et al.* [101] could be used to derive BDGs (and could easily be plugged into our approach), however appropriate edge weights need to be defined and calculated for them.

The header file hotspot heuristic assumes that header files that have a high rebuild cost only become build performance problems when they change frequently. Poor build performance in infrequently changing header files still poses a lingering threat to build performance. However, the approach allows practitioners to configure the hotspot quadrant thresholds to match their build performance requirements.

### 7.7.4 External Validity

We focus our case study on four open source systems, which threatens the generalizability of our case study results. However, we studied a variety of systems with different sizes and domain to combat potential bias.

The build systems of the studied systems rely on `make` specifications, which may bias our case study results towards such technologies. However, our approach is agnostic of the underlying build system, operating on a build dependency graph, which can be extracted from any build system. Furthermore, our study focuses on header file

hotspots, which are a property of C/C++ systems for which `make`-based build systems are the de facto standard (*cf.* Chapter 4).

The thresholds that we selected for the quadrant plots threaten the reliability of our case study results. Use of different thresholds will produce different quadrants, and thus, different header file hotspots. However, we believe that our selected elapsed time thresholds are representative, since the values were derived from the literature [36, 45]. Moreover, we use the median for the rate of change threshold — a metric that is resilient to outliers.

## 7.8 Chapter Summary

Developers rely on the build system to produce testable deliverables in a timely fashion. A fast build system is at the heart of modern software development. However, software systems are large and complex, often being composed of thousands of source code files that must be carefully translated into deliverables in a timely fashion by the build system. As software projects age, their build systems tend to grow in size and complexity, making build profiling and performance analysis challenging.

In this chapter, we propose an approach for pinpointing build hotspots by analyzing the build dependency graph and the change history of a software system. Our approach can be used to prioritize build optimization effort, allowing teams to focus effort on the files that will deliver the most value in return. The empirical study of this chapter sets out to explore the following question:

**Central Question:** *Which files should development teams optimize first to improve build performance the most? Which properties of hotspot files should development teams focus optimization effort on?*

Through a case study on four open source systems, we show that:

- The build hotspot approach highlights header files that, if optimized, yield more improvement in the future total rebuild cost than just the header files that trigger the slowest rebuild processes, change the most frequently, or are used the most throughout the codebase (Section 7.5).
- Regression models are capable of explaining between 32%-57% of the detected hotspots using code layout and content properties of the header files (Section 7.6).
- In large projects, build optimization benefits more from architectural refinement than from acting on code properties like header file fan-in alone (Section 7.6).

CHAPTER 8

# Conclusions and Future Work

KEY CONCEPT

*The benefits provided by the build system come at a cost — build systems introduce overhead on the software development process.*

The software build system has long been at the heart of the software development process. The rapid pace at which modern software is developed has raised the profile of the build system. A quick and robust build system has become critical infrastructure that organizations need in order to keep pace with market competitors.

On the other hand, the benefits provided by the build system come at a cost — build systems introduce overhead on the software development process. In this thesis, we empirically study the software development overhead introduced by the maintenance and execution of build systems. In the remainder of this section, we outline the contributions of this thesis and lay out promising avenues for future research.

## 8.1 Contributions and Findings

The overarching goal of this thesis is to better understand the factors that influence the overhead introduced by the build system. To do so, we leverage data stored in software repositories and within the build system itself. Broadly speaking, we find that:

> **Thesis Statement:** *The overhead introduced by the build system is an important issue that development teams need to manage. Historical data extracted from software repositories and facts extracted from the build system itself can inform organizational decisions that aim to mitigate this overhead.*

We investigate the overhead introduced by build systems in four empirical studies. Below, we reiterate the main findings of this thesis:

1. **Build Technology Choice**: Although modern technologies like Maven provide additional features that older technologies like `make` do not, they tend to require additional maintenance activity that teams should be aware of when making technology choices (Chapter 4).

2. **Cloning in Build Specifications**: Build cloning is a commonly used solution to build system maintenance problems. Typical cloning rates in build systems are much higher than those of other software artifacts (Chapter 5). Nonetheless, there are commonly-adopted patterns of build system implementation that leverage creative means of abstraction that can keep build cloning rates under control.

3. **Drivers of Build Co-Change**: Accurate classifiers can be trained to identify the source and test code changes that will require accompanying build changes (Chapter 6). These classifiers are especially accurate for changes to C/C++ code.

4. **Identifying and Understanding Build Hotspots**: We propose an approach to detect *build hotspots*, i.e., files that rebuild slowly and change often (Chapter 7). Through a simulation exercise, we demonstrate that if these hotspot files were optimized, they would yield more improvement in time spent building than optimizing the files that rebuild the slowest, change the most frequently, or are used the most throughout the codebase. Moreover, logistic regression models can accurately explain the incidence of build hotspots using only code and layout properties of files (Chapter 7). Furthermore, in large systems, build optimization would benefit more from focusing on architectural refinement than acting on code properties like fan-in.

Surprisingly, we find that the more modern build technologies tend to require additional maintenance activity (Chapter 4) and tend be more prone to cloning (Chapter 5) than the older studied technologies. At first glance, this may seem like a grim observation. However, we believe that this indicates that the additional features provided by more modern technologies are not free of cost. Organizations should consider these additional costs when determining which build technologies are appropriate for their software systems.

## 8.2 Opportunities for Future Research

Although we believe that this thesis has made a positive contribution toward understanding the overhead introduced by build systems, there is plenty of room for future research. Below, we outline several promising avenues for future work.

### 8.2.1 The Impact of Cloning on Build Maintenance

While we have seen the shortcomings of a clone-based build system design at Munich Re, we do not know to what extent it can be generalized. For example, it could be that different types of build specification information have differences in change-proneness. Clones in some areas (e.g. construction) could be more problematic than in others (e.g. configuration). Analysis of the evolution of clones in build systems could help to further our knowledge.

### 8.2.2 Understanding Build Co-Change

Our findings in Chapter 6 suggest that most C++ build changes and at least the code-related Java build changes can indeed be predicted using characteristics of corresponding changes to source and test code. However, we find that much more of the build co-change in Java and web-driven code tends to be related to other roles in the development process (e.g., release engineering, quality assurance). Hence, to improve the performance of our co-change classifiers for release engineers, build maintainers, and quality assurance personnel, future work should explore metrics related to build structure and platform configuration.

### 8.2.3 Combining Hotspot Detection with Automated Refactoring

Our approach focuses on the detection and prioritization of header file hotspots, but does not suggest automatic hotspot refactorings. In this respect, our approach is similar to defect prediction, which is used to focus quality assurance effort on the most

defect-prone modules. Automatically proposing fixes for hotspots may require domain-specific expertise. For example, an automatically generated build dependency graph refactoring may fix hotspots in theory, but in practice may require an infeasibly complex restructuring of the system, reducing other properties of a software system like comprehension and maintenance. Further work is needed to find a balance between these forces.

### 8.2.4 Build Parallelism Bottlenecks

As the speed of computer processors stagnate, developers rely more and more on parallel processing in order to optimize software system performance. The same is true for build processes, which increasingly rely on distributed and parallel build architectures in order to improve build performance. Yet the speedup achieved by parallel build processes is limited by the dependency structure of the system being constructed. Thus, files in the build process may act like bottlenecks that slow the build process down. In future work, we plan to analyze the graph of dependencies described by the build system in order to identify bottlenecks that prevent parallel builds from achieving larger speedups.

### 8.2.5 Enhancing Software Analyses using Build Data

While this thesis has focused on the various ways that the build system introduces overhead on the software development process, the build system contains plenty of useful information that researchers can leverage to complement analyses of software systems. For example, in recent work, we mine traces of build executions to identify potential software licensing violations [106]. The potential violations identified by our approach

are of immense practical value, generating rapid reactionary measures in several open source systems.

The data contained in the build system can also be used to enhance other types of software analysis. For example, impact analysis of defects is of growing importance in the field of software defect prediction. The build system can be used to aid in impact analysis by not only identifying the deliverables that are impacted by a defect, but also by identifying the impacted configurations of the software (e.g., Windows vs. Mac OS X). The intuition behind this approach would be that a defect that impacts many deliverables in many configurations has a higher impact than one that impacts few deliverables and configurations.

# Bibliography

[1] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir. The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers. *IEEE Software*, 32(2):42–49, 2015. (Cited on page 2)

[2] B. Adams, C. Bird, F. Khomh, and K. Moir. 1st International Workshop on Release Engineering. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)*, pages 1545–1546, 2013. (Cited on page 2)

[3] B. Adams, K. De Schutter, H. Tromp, and W. Meuter. Design recovery and maintenance of build systems. In *Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM)*, pages 114–123, 2007. (Cited on pages 21, 146, and 162)

[4] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter. The Evolution of the Linux Build System. *Electronic Communications of the ECEASST*, 8, 2008. (Cited on pages 5, 24, 30, 87, 116, 119, 146, and 184)

[5] R. Adams, W. Tichy, and A. Weinert. The Cost of Selective Recompilation and Environment Processing. *Transactions On Software Engineering and Methodology (TOSEM)*, 3(1):3–28, January 1994. (Cited on page 27)

[6] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. Fault Localization for Build Code

Errors in Makefiles. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 600–601, 2014. (Cited on page 21)

[7] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. Fault Localization for Make-Based Build Crashes. In *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 526–530, 2014. (Cited on page 21)

[8] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Detecting Semantic Changes in Makefile Build Code. In *Proc. of the 28th Int'l Conf. on Software Maintenance (ICSM)*, pages 150–159, 2012. (Cited on page 21)

[9] T. L. Alves, C. Ypma, and J. Visser. Deriving Metric Thresholds from Benchmark Data. In *Proc. of the 26th Int'l Conf. on Software Maintenance (ICSM)*, pages 1–10, 2010. (Cited on page 100)

[10] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an Open Bug Repository. In *Proc. of the OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005. (Cited on page 1)

[11] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of the 2nd Working Conf. on Reverse Engineering (WCRE)*, pages 86–95, 1995. (Cited on page 104)

[12] R. Barandela, J. Sánchez, V. García, and F. Ferri. Learning from Imbalanced sets through resampling and weighting. In *Proc. of the 1st Iberian Conf. on Pattern Recognition and Image Analysis (IbPRIA)*, 2003. (Cited on page 132)

[13] D. F. Bauer. Constructing Confidence Sets Using Rank Statistics. *Journal of the American Statistical Association*, 67(339):687–690, 1972. (Cited on page 56)

[14] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proc. of the 25th Int'l Conf. on Automated Software Engineering (ASE)*, 2010. (Cited on page 23)

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proc. of the 7th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, 2009. (Cited on pages 81 and 128)

[16] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The Promises and Perils of Mining Git. In *Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)*, 2009. (Cited on page 62)

[17] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001. (Cited on pages 130 and 136)

[18] J. Buffenbarger. Adding Automatic Dependency Processing to Makefile-Based Build Systems with Amake. In *Proc. of the 1st Int'l Workshop on RELease ENGineering (RELENG)*, pages 1–4, 2013. (Cited on page 21)

[19] D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. Scatterplot Matrix Techniques for Large N. *Journal of the Amarican Statistical Association*, 82(398):424–436, 1987. (Cited on page 97)

[20] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *Transactions on Software Engineering (TSE)*, 35(6):864–878, 2009. (Cited on pages 175 and 178)

[21] J. M. Chambers and T. J. Hastie, editors. *Statistical Models in S*, chapter 4. Wadsworth and Brooks/Cole, 1992. (Cited on page 177)

[22] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. (Cited on page 23)

[23] W. Cunningham. The WyCash Portfolio Management System. In *Addendum to the Proc. of the 7th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 29–30, 1992. (Cited on page 26)

[24] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos. Improving the Build Architecture of Legacy C/C++ Software Systems. In *Proc. of the 8th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 96–110, 2005. (Cited on pages 27, 163, and 171)

[25] M. de Jonge. Decoupling source trees into build-level components. In J. Bosch and C. Krueger, editors, *Proc of the 8th Int'l Conf. on Software Reuse (ICSR)*, volume 3107 of *LNCS*, pages 215–231. Springer-Verlag, July 2004. (Cited on page 23)

[26] M. de Jonge. Build-level components. *Transactions on Software Engineering (TSE)*, 31(7):588–600, 2005. (Cited on page 23)

[27] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008. (Cited on page 99)

[28] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proc. of the 16th*

*Int'l Software Product Line Conference (SPLC)*, pages 21–30, 2012. (Cited on page 24)

[29] M. Dmitriev. Language-Specific Make Technology for the Java Programming Language. In *Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. ACM, 2002. (Cited on pages 110 and 142)

[30] J. Downs, B. Plimmer, and J. G. Hosking. Ambient Awareness of Build Status in Collocated Software Teams. In *Proc. of the 34th Int'l Conf. on Software Engineering (ICSE)*, pages 507–517, 2012. (Cited on page 25)

[31] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of the 15th Int'l Conf. on Software Maintenance (ICSM)*, 1999. (Cited on pages 104 and 116)

[32] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007. (Cited on page 2)

[33] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. An Infrastructure for Composing Build Systems of Software Product Lines. In *Proc. of the 15th Int'l Software Product Line Conference (SPLC)*, volume 2, pages 18:1–18:8, 2011. (Cited on page 23)

[34] A. Estabrooks and N. Japkowicz. A mixture-of-experts framework for learning from imbalanced data sets. In *Proc. of the 4th Int. Conf. on Advances in Intelligent Data Analysis (IDA)*, pages 34–43, 2001. (Cited on page 132)

[35] S. Feldman. Make - a program for maintaining computer programs. *Software -*

*Practice and Experience*, 9(4):255–265, 1979. (Cited on pages 14, 22, 27, 32, 149, 150, and 210)

[36] A. R. H. Fischer, F. J. J. Blommaert, and C. J. H. Midden. Monitoring and evaluation of time delay. *Int'l Journal of Human-Computer Interaction*, 19(2):163–180, 2005. (Cited on pages 167 and 186)

[37] J. Fox. *Applied Regression Analysis and Generalized Linear Models*. Sage Publications, $2^{nd}$ edition, 2008. (Cited on page 177)

[38] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. of the 14th Int'l Conf. on Software Maintenance (ICSM)*, pages 190–198, 1998. (Cited on page 62)

[39] S. Grant, J. R. Cordy, and D. B. Skillicorn. Reverse Engineering Co-maintenance Relationships Using Conceptual Analysis of Source Code. In *Proc. of the 18th Working Conf. on Reverse Engineering (WCRE)*, pages 87–91, 2011. (Cited on page 146)

[40] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting Fault Incidence using Software Change History. *Transactions on Software Engineering (TSE)*, 26(7):653–661, 2000. (Cited on page 55)

[41] R. Hardt and E. V. Munson. Ant Build Maintence with Formiga. In *Proc. of the 1st Int'l Workshop on RELease ENGineering (RELENG)*, pages 13–16, 2013. (Cited on page 21)

[42] A. E. Hassan and K. Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proc. of the 21st Int'l Conf. on Automated Software Engineering (ASE)*, pages 189–198, 2006. (Cited on pages 7, 25, 26, 120, and 121)

[43] I. Herraiz, G. Robles, J. Gonzalez-Barahona, A. Capiluppi, and J. Ramil. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *Proc. of the 10th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 213–221, 2006. (Cited on page 47)

[44] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In *Proc. of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 384–387, 2011. (Cited on pages 5 and 33)

[45] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010. (Cited on pages 2, 18, 27, 42, 150, 163, and 186)

[46] W. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan. Should I contribute to this discussion? In *Proc. of the 7th working conf. on Mining Software Repositories (MSR)*, 2010. (Cited on pages 121 and 132)

[47] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can Clone Detection Support Quality Assessments of Requirements Specifications? In *Proc. of the 32nd Int'l Conf. on Software Engineering (ICSE)*, volume 2, pages 79–88, 2010. (Cited on page 104)

[48] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter?

In *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, pages 485–495, 2009. (Cited on pages 6, 24, 94, and 115)

[49] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *Transactions on Software Engineering (TSE)*, 28(7):654–670, 2002. (Cited on page 104)

[50] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 2008. (Cited on page 128)

[51] C. J. Kasper and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008. (Cited on pages 24, 25, and 115)

[52] N. Kerzazi, F. Khomh, and B. Adams. Why do Automated Builds Break? An Empirical Study. In *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 41–50, 2014. (Cited on pages 25 and 120)

[53] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan. An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox. In *Proc. of the 18th Working Conf. on Reverse Engineering (WCRE)*, pages 261–270, 2011. (Cited on page 159)

[54] P. Knab, M. Pinzger, and A. Bernstein. Predicting Defect Densities in Source Code Files with Decision Tree Learners. In *Proc. of the 3rd Int'l Workshop on Mining Software Repositories (MSR)*, pages 119–125, 2006. (Cited on page 121)

[55] A. Koenig. *Patterns and antipatterns*, pages 383–389. Cambridge University, 1998. (Cited on page 6)

[56] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Automated Software Engineering*, 3(1-2):77–108, 1996. (Cited on page 104)

[57] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007. (Cited on pages 24 and 91)

[58] G. Kumfert and T. Epperly. Software in the DOE: The Hidden Overhead of "The Build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA, 2002. (Cited on page 5)

[59] I. Kwan, A. Schröter, and D. Damian. Does Socio-Technical Congruence Have An Effect on Software Build Success? A Study of Coordination in a Software Project? *Transactions on Software Engineering (TSE)*, 37(3):307–324, May/June 2011. (Cited on pages 25 and 120)

[60] B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proc. of the 13th Int'l Conf. on Software Maintenance (ICSM)*, pages 314–321, 1997. (Cited on page 104)

[61] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996. (Cited on pages 153 and 163)

[62] R. Lawrence. The space efficience of XML. *Information and Software Technology (IST)*, 46(11):753–759, 2004. (Cited on page 61)

[63] Linden Labs. CMake. http://wiki.secondlife.com/wiki/CMake, July 2010. Last viewed: 20-Aug-2010. (Cited on page 75)

[64] D. H. Martin, J. R. Cordy, B. Adams, and G. Antoniol. Make It Simple — An Empirical Analysis of GNU Make Feature Use in Open Source Projects. In *Proc. of the 23rd Int'l Conf. on Program Comprehension (ICPC)*, 2015. To appear. (Cited on page 21)

[65] K. Martin and B. Hoffman. *Mastering CMake, 5th Edition*. Kitware Inc., Clifton Park, NY, USA, 2009. (Cited on page 111)

[66] S. McIntosh, B. Adams, and A. E. Hassan. The Evolution of ANT Build Systems. In *Proc. of the 7th Working Conf. on Mining Software Repositories (MSR)*, pages 42–51, 2010. (Cited on pages 5 and 146)

[67] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, August 2012. (Cited on pages 5, 30, 55, 87, 113, 116, 119, 140, 146, and 184)

[68] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining Co-Change Information to Understand when Build Changes are Necessary. In *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 241–250, 2014. (Cited on page 119)

[69] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Identifying and Understanding Header File Hotspots in C/C++ Build Processes. *Automated Software Engineering*, In press, 2015. (Cited on page 149)

[70] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, pages 141–150, 2011. (Cited on pages 3, 5, 28, 30, 34, 39, 62, 81, 98, 119, 125, 126, and 146)

[71] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the QT, VTK, and ITK Projects. In *Proc. of the 11th Working Conf. on Mining Software Repositories (MSR)*, pages 192–201, 2014. (Cited on page 18)

[72] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering*, In press, 2015. (Cited on page 30)

[73] S. McIntosh, M. Poehlmann, E. Juergens, A. Mockus, B. Adams, A. E. Hassan, B. Haupt, and C. Wagner. Collecting and Leveraging a Benchmark of Build System Clones to Aid in Quality Assessments. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 115–124, 2014. (Cited on page 87)

[74] P. Miller. Recursive make considered harmful. In *Australian Unix User Group Newsletter*, volume 19, pages 14–25, 1998. (Cited on pages 24, 32, 53, and 111)

[75] R. G. Miller. *Simultaneous Statistical Inference*. Springer, 1981. (Cited on page 47)

[76] A. Mockus. Software support tools and experimental work. In *Proc. of the Int'l Conf. on Empirical Software Engineering Issues: Critical Assessment and Future Directions*, pages 91–99, 2007. (Cited on pages 35 and 96)

[77] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)*, pages 11–20, 2009. (Cited on pages 23, 36, 96, and 97)

[78] A. Mockus. Organizational Volatility and its Effects on Software Defects. In *Proc. of the 18th Symposium on the Foundations of Software Engineering (FSE)*, pages 117–126, 2010. (Cited on page 175)

[79] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proc. of the 8th Int'l Symposium on Software Metrics*, pages 87–94, 2002. (Cited on page 104)

[80] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for Build Debt: Experiences Managing Technical Debt at Google. In *Proc. of the 3rd Int'l Workshop on Managing Technical Debt (MTD)*, pages 1–6, 2012. (Cited on page 26)

[81] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 140–151, 2014. (Cited on page 24)

[82] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. Linux Variability Anomalies: What Causes Them and How Do They Get Fixed? In *Proc. of the 10th*

*Working Conf. on Mining Software Repositories (MSR)*, pages 111–120, 2013. (Cited on page 26)

[83] S. Nadi and R. Holt. Make it or Break it: Mining Anomalies in Linux Kbuild. In *Proc. of the 18th Working Conf. on Reverse Engineering (WCRE)*, pages 315–324, 2011. (Cited on page 21)

[84] S. Nadi and R. Holt. Mining Kbuild to Detect Variability Anomalies in Linux. In *Proc. of the 16th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 107–116, 2012. (Cited on page 21)

[85] S. Nadi and R. Holt. The Linux kernel: a case study of build system variability. *Journal of Software: Evolution and Practice (JSEP)*, 26(8):730–746, 2014. (Cited on page 21)

[86] A. Neitsch, K. Wong, and M. W. Godfrey. Build System Issues in Multilanguage Software. In *Proc. of the 28th Int'l Conf. on Software Maintenance*, pages 140–149, 2012. (Cited on pages 2, 21, 25, and 99)

[87] G. V. Neville-Neal. Kode vicious: System changes and side effects. *Communications of the ACM*, 52(4):25–26, April 2009. (Cited on page 2)

[88] T. H. D. Nguyen, B. Adams, and A. E. Hassan. A Case Study of Bias in Bug-Fix Datasets. In *Proc. of the 17th Working Conf. on Reverse Engineering (WCRE)*, pages 259–268, 2010. (Cited on pages 81 and 128)

[89] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel.

In *Proc. of the 17th Software Product Line Conference (SPLC)*, pages 91–100, 2013. (Cited on page 24)

[90] S. Phillips, T. Zimmermann, and C. Bird. Understanding and Improving Software Build Teams. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 735–744, 2014. (Cited on page 5)

[91] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. (Cited on page 177)

[92] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012. (Cited on pages 24 and 115)

[93] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining Software Evolution to Predict Refactoring. In *Proc. of the 1st Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 354–363, 2007. (Cited on page 121)

[94] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007. (Cited on pages 24 and 91)

[95] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' Build Errors: A Case Study (at Google). In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 724–734, 2014. (Cited on pages 25, 26, 27, and 120)

[96] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. ichi Matsumoto. Predicting Re-opened Bugs: A Case Study on the Eclipse Project. In *Proc. of the 17th Working Conf. on Reverse Engineering (WCRE)*, pages 249–258, 2010. (Cited on page 121)

[97] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the Impact of Code and Process Metrics on Post-Release Defects: A Case Study on the Eclipse Project. In *Proc. of the 4th Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2010. (Cited on page 175)

[98] M. Shridhar, B. Adams, and F. Khomh. A Qualitative Analysis of Software Build System Changes and Build Ownership Styles. In *Proc. of the 8th Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 29:1–29:10, 2014. (Cited on page 26)

[99] P. Smith. *Software Build Systems: Principles and Experience*. Addison-Wesley, 1st edition, March 2011. (Cited on pages 2, 14, 21, 22, and 99)

[100] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams. An Empirical Study of Build System Migrations in Practice: Case Studies on KDE and the Linux Kernel. In *Proc. of the 28th Int'l Conf. on Software Maintenance (ICSM)*, pages 160–169, 2012. (Cited on pages 5, 22, 24, 31, 34, 69, and 76)

[101] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen. Build Code Analysis with Symbolic Evaluation. In *Proc. of the 34th Int'l Conf. on Software Engineering (ICSE)*, pages 650–660, 2012. (Cited on pages 21 and 185)

[102] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. SYMake: A Build Code Analysis and Refactoring Tool for Makefiles. In *Proc. of the 27th Int'l Conf. on Automated Software Engineering (ASE)*, pages 366–369, 2012. (Cited on page 21)

[103] A. Telea and L. Voinea. A Tool for Optimizing the Build Performance of Large Software Code Bases. In *Proc. of the 12th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 323–325, 2008. (Cited on page 27)

[104] Q. Tu and M. W. Godfrey. The Build-Time Software Architecture View. In *Proc. of the 17th Int'l Conf. on Software Maintenance (ICSM)*, pages 398–407, 2001. (Cited on pages 23 and 167)

[105] H. Unphon. Making Use of Architecture throughout the Software Life Cycle - How the Build Hierarchy can Facilitate Product Line Development. In *Proc. of the 4th Int'l Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, pages 41–48, 2009. (Cited on page 23)

[106] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel. Tracing Software Build Processes to Uncover License Compliance Inconsistencies. In *Proc. of the 29th Int'l Conf. on Automated Software Engineering (ASE)*, pages 731–741, 2014. (Cited on page 192)

[107] T. van der Storm. Backtracking Incremental Continuous Integration. In *Proc. of the 12th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 233–242, 2008. (Cited on page 25)

[108] T. Wolf, A. Schröter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, pages 1–11, Washington, DC, USA, 2009. (Cited on page 143)

[109] X. Xia, X. Zhou, D. Lo, and X. Zhao. An Empirical Study of Bugs in Software Build

Systems. In *Proc. of the 13th Int'l Conf. on Quality Software (QSIC)*, pages 200–203, 2013. (Cited on page 21)

[110] X. Xia, X. Zhou, D. Lo, X. Zhao, and Y. Wang. An Empirical Study of Bugs in Software Build System. *IEICE Transactions on Information and Systems*, E97-D(7):1769–1780, 2014. (Cited on page 21)

[111] Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing False Code Dependencies to Speedup Software Build Processes. In *Proc. of the 13th IBM Centre for Advanced Studies Conference (CASCON)*, pages 343–352, 2003. (Cited on pages 27, 163, 166, 169, 176, and 182)

[112] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos. Reducing Build Time Through Precompilations for Evolving Large Software. In *Proc. of the 21st Int'l Conf. on Software Maintenance (ICSM)*, pages 59–68, 2005. (Cited on pages 27, 163, and 169)

[113] E. Zadok. Overhauling Amd for the '00s: A Case Study of GNU Autotools. In *Proc. of the FREENIX Track on the USENIX Technical Conf.*, pages 287–297. USENIX Association, 2002. (Cited on page 22)

[114] T. Zimmermann and P. Weissgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proc. of the 1st Int'l Workshop on Mining Software Repositories (MSR)*, pages 2–6, 2004. (Cited on page 81)

Appendix A

# Build Technology Examples

In this appendix, we briefly describe how each of the studied technologies can be used to specify a simple build system.

## A.1 Low-Level

Figure A.1 provides working examples of the five studied low-level build technologies.

### A.1.1 Make

One of the earliest build technologies on record is Feldman's `make` tool [35], which automatically synchronizes program sources with deliverables. `Make` specifications outline target-dependency-recipe tuples. *Targets* specify files created by a *recipe*, i.e., a shell script that is executed when the target either: (1) does not exist, or (2) is older than one or more of its *dependencies*, i.e., a list of other files and targets.

The `make` specification snippet in Figure A.1a describes three target-dependency-recipe tuples. Lines 2, 4, and 7 list targets to the left of the colons and dependency lists to the right. Recipes are specified for the `main.o` and `example` targets on lines 5 and

```
1 .PHONY: all
2 all: example
3
4 example: main.o
5         gcc -o example main.o
6
7 main.o: main.c
8         gcc -c main.c
```

(a) Make

```
1  rule LinkRule {
2          Depends $(1) : $(2) ;
3          Link $(1) : $(2) ;
4  }
5
6  actions Link {
7          gcc -o $(1) $(2)
8  }
9
10 rule CompileRule {
11         Depends $(1) : $(2) ;
12         Compile $(1) : $(2) ;
13 }
14
15 actions Compile {
16         gcc -c -o $(1) $(2)
17 }
18
19 LinkRule example : main.o ;
20 CompileRule main.o : main.c ;
```

(b) Jam

```
1  <project name="example">
2    <target name="compile">
3      <javac
4        destdir="classes"
5        srcdir="src"
6        includes="**/*.java"
7      />
8    </target>
9
10   <target
11     name="link"
12     depends="compile"
13   >
14     <jar
15       jarfile="example.jar"
16       basedir="classes"
17     />
18   </target>
19 </project>
```

(c) Ant

```
1 env = Environment(CXX = "g++")
2
3 srcs = Split("main.cc")
4
5 objects = env.Object(source = srcs)
6
7 t = env.Program(target="example", source=objects)
8 Default(t)
```

(d) SCons

```
1 task :default => [:utest]
2
3 task :utest do
4   ruby utest.rb
5 end
```

(e) Rake

Figure A.1: Example low-level technology specifications.

8. Line 1 of Figure A.1a specifies that the `all` target is *phony*, representing an abstract phase in the build process rather than a concrete file in the filesystem.

### A.1.2 Jam

Jam provides a more procedural-style structure for target-dependency-recipe tuples. Figure A.1b shows how *rules* (the equivalent of `make` tuples) can be specified (lines 1-4 and 10-13). Dependencies are expressed by invoking the built-in *Depends* rule on lines 2 and 11. Jam *actions* (the equivalent of `make` recipes) for C compilation and object code linking are defined on lines 6-8 and 15-17 respectively.

### A.1.3 Ant

Ant borrows the (phony) target-dependency-recipe concept from `make`, however all Ant targets are abstract. When an Ant target is triggered, a list of specified *tasks* (the equivalent of `make` recipes) are invoked. Ant tasks execute Java code rather than shell scripts to synchronize sources with deliverables.

Figure A.1c shows an Ant specification that describes two targets, i.e., `compile` (lines 2-8) and `link` (lines 10-18). The `compile` target invokes the `javac` task (lines 3-7), which executes the `javac` compiler. The `link` target invokes the `jar` task (lines 14-17), which executes the `jar` command. The dependency between the `link` and `compile` targets is expressed on line 12 using the *depends* target attribute.

### A.1.4 SCons

SCons provides several advanced build system features (e.g., implicit dependency tracking for header files in popular programming languages) and allows maintainers to write highly portable build specifications using Python. Line 7 of Figure A.1d shows how a binary *example* can be assembled from object code. Line 5 shows how object code can be generated using SCons built-in support for C++ compilation. Environmental settings (e.g., compilers, linkers, and flags) are automatically detected, however parameters passed to the `Environment()` function call will override the detected settings, as shown on line 1.

```
1 AC_INIT([example], [1.0])
2 AM_INIT_AUTOMAKE
3 AC_PROG_CC
4 AC_CONFIG_HEADERS([config.h])
5 AC_CONFIG_FILES([Makefile])
6 AC_OUTPUT
```

(a) Autotools (Autoconf)

```
1 bin_PROGRAMS = example
2 example_SOURCES = main.c
```

(b) Autotools (Automake)

```
1 cmake_minimum_required(VERSION 2.6)
2 project(Example)
3
4 add_executable(example main.cc)
```

(c) CMake

Figure A.2: Example abstraction-based technology specifications.

### A.1.5 Rake

Rake is a modern build tool with advanced support for building Ruby applications. Similar to SCons, Rake specifications are written in a high-level scripting language (i.e., Ruby), to give build maintainers the power to express complex relationships and transformations in a highly portable language. Similar to Ant, Rake *tasks* (the equivalent of targets in `make`) are abstract.

The example snippet in Figure A.1e shows how a unit testing task `utest` can be specified (lines 3-5). Line 4 describes the recipe that is executed when `utest` is triggered. Line 1 specifies that the `default` target depends upon the `utest` target.

## A.2 Abstraction-Based

Figure A.2 provides working examples of the two studied abstraction-based technologies (*cf.* Section 2.2.2).

### A.2.1 Autotools

GNU Autotools specifications describe external and internal dependencies, configurable compile-time features, and platform requirements. These specifications are parsed to generate `make` specifications that satisfy the described constraints.

Autotools is actually a large collection of build tools that work together to generate build systems according to specifications. Two of the most commonly used tools are `autoconf` and `automake`, for which we provide example specifications in Figures A.2a and A.2b respectively. Lines 1 and 2 of Figure A.2a initialize the `autoconf` environment, specifying that our project name is *example* version *1.0* and that automake is also necessary. Line 3 specifies an environment dependency on a C compiler, while lines 4 and 5 request that the configuration step store preprocessor directives in a file named `config.h`, and store the build system implementation in a file called `Makefile`. Line 1 of Figure A.2b specifies that a deliverable called *example* should be constructed during the build process and that it should be deployed in the *bin* directory. Line 2 states that main.c is a source file that should be compiled and linked into the *example* binary.

### A.2.2 CMake

Similar to Autotools, CMake abstractions can be used to generate `make` specifications, but can also generate Microsoft Visual Studio and Apple Xcode project files. Figure A.2c specifies that a build system should be generated to produce a binary called *example* by compiling and linking `main.cc` (line 4) as a part of a project called *Example* (line 2). Line 1 denotes that CMake version 2.6 (or later) should be used to parse the specification.

## A.3 Framework-Driven

Below we describe the studied Maven framework-driven technology (*cf.* Section 2.2.3).

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>
    an.example.application
  </groupId>
  <artifactId>example</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>example</name>
  <build>
    <plugins>
      <plugin>
        <groupId>
          org.apache.maven.plugins
        </groupId>
        <artifactId>
          maven-compiler-plugin
        </artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.5</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>
</project>
```

(a) Maven

```xml
<ivy-module version="2.0">
  <info
    organisation="example"
    module="application"
  />
  <dependencies>
    <dependency
      org="junit"
      name="junit"
      rev="3.8.1"
    />
  </dependencies>
</ivy-module>
```

(b) Ivy

```
source "https://rubygems.org"

gem "rake", ">=10.0.3"
gem "rspec", "2.13.0"
```

(c) Bundler

Figure A.3: Example Framework-driven and dependency management technology specifications.

### A.3.1 Maven

Maven assumes that source and test files are placed in default locations and that projects adhere to a typical Java dependency policy, unless otherwise specified. If projects abide by the conventions, Maven can infer build behaviour automatically without any explicit specification. For example, Figure A.3a does not specify a location for source or output files. Convention specifies that source and unit test code appear under `src/main/java` and `src/test/java`, respectively.

Lines 10-18 of Figure A.3a show how the Maven convention can be overridden through configuration. The Java compiler is instructed to operate in Java 1.5 source mode (line 15), and generate bytecode that is compatible with the Java 1.7 runtime environment

(line 16).

## A.4 Dependency Management

Figure A.3 provides working examples of dependency management in Maven and the two studied dependency management technologies (*cf.* Section 2.2.4).

### A.4.1 Maven

In addition to providing a framework-driven build environment, Maven doubles as a dependency management technology. Lines 22-26 of Figure A.3a provide an example dependency declaration on the JUnit tool, version 3.8.1.

### A.4.2 Ivy

Ivy provides dependency management features that are most notably leveraged by Ant. Figure A.3b shows an Ivy specification for the same JUnit dependency as depicted in Figure A.3a.

### A.4.3 Bundler

Bundler provides packaging and dependency management for Ruby applications. Line 1 of Figure A.3c specifies that bundler should download *gems*, i.e., Ruby packages, from the given host. Lines 2 and 3 specify dependencies on Rake version 10.0.3 (at least) and rspec version 2.13.0 (exact).

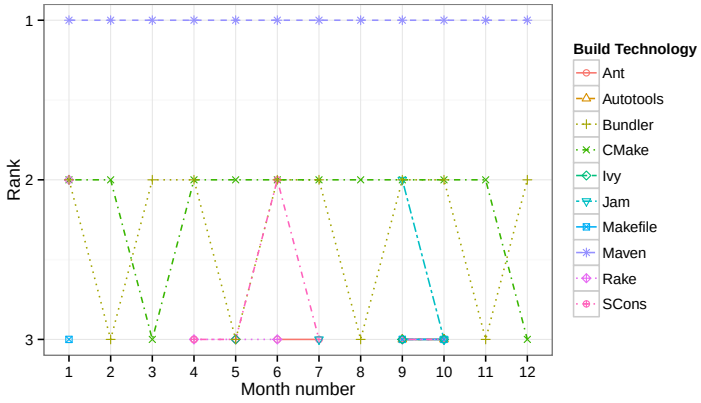
Appendix B

# Additional Figures

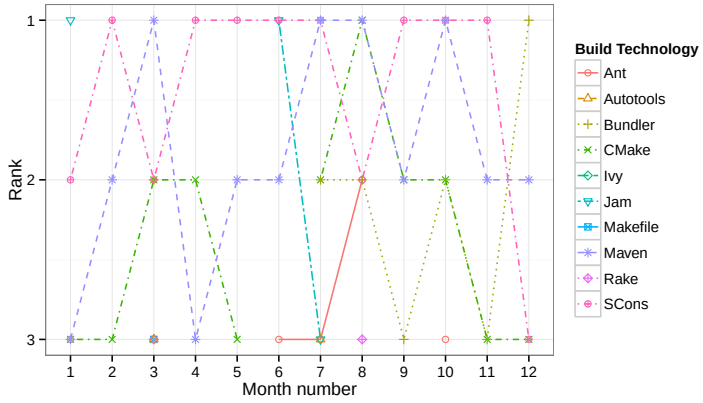
## B.1 Build Technology Choice

We perform longitudinal analyses of the Tukey HSD ranks for each metric in the forges to complement our median-based analyses in Chapter 4. Figures B.1 and B.2 show only the first twelve months of history and the top three ranks to improve the readability of the figures. Unfiltered figures are available online.[1]

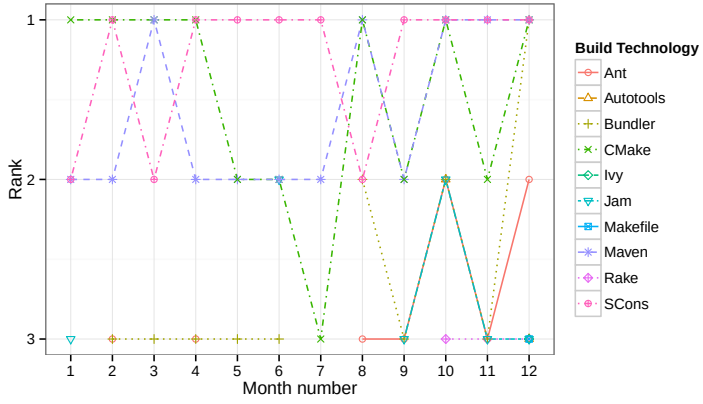
[1] http://sailhome.cs.queensu.ca/replication/shane/PhD/




(a) Build commit proportion.

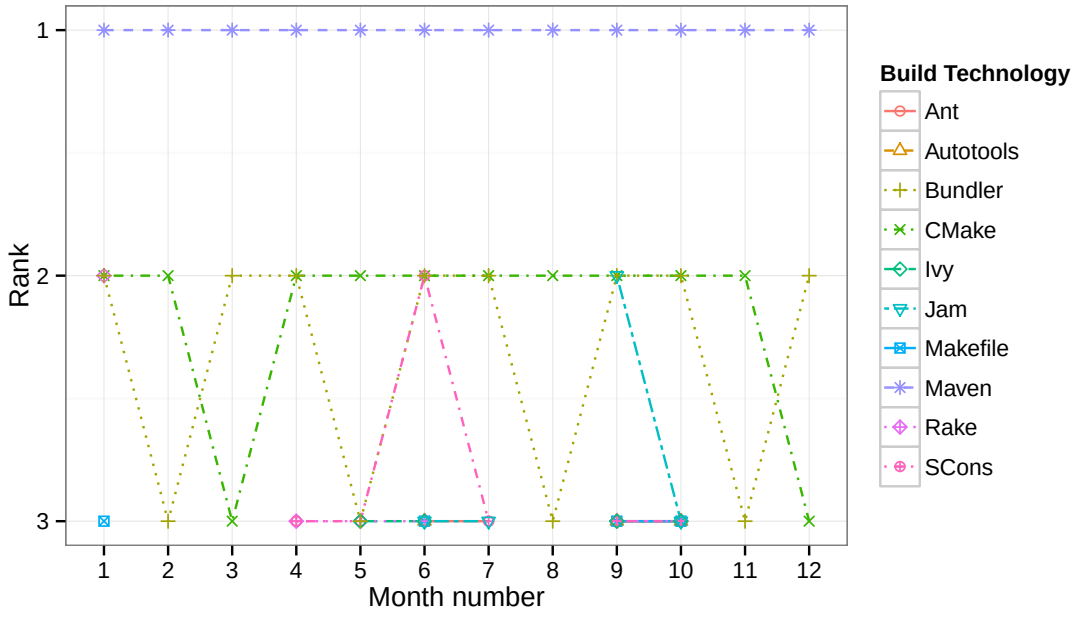



(b) Build change size.

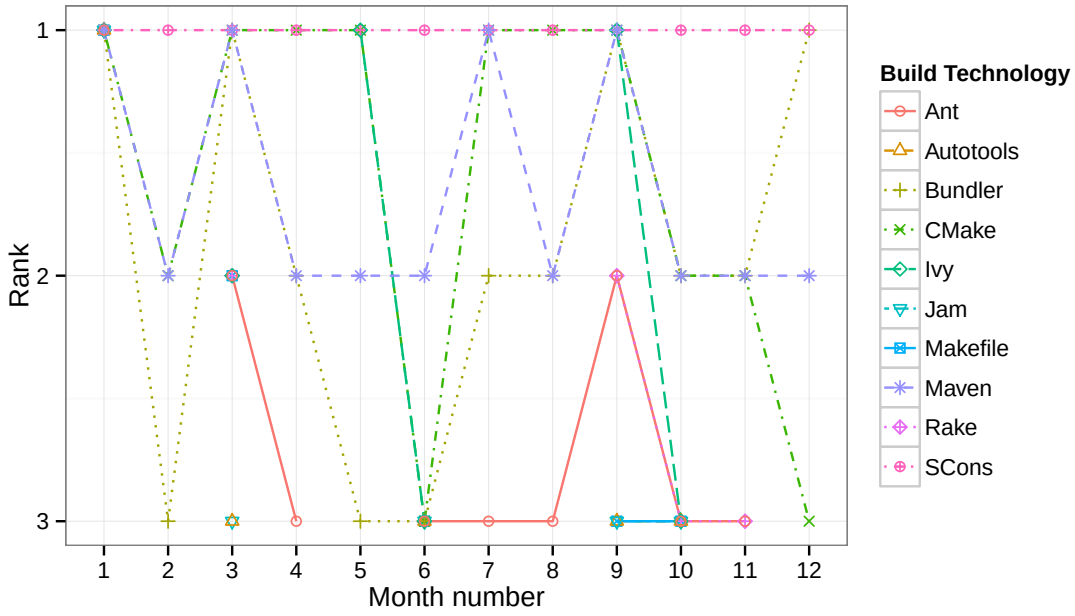



(c) Build churn volume.

Figure B.1: Monthly build commit proportion, sizes, and churn volume in the studied forges.




(a) Logical coupling.




(b) Build author ratio.

Figure B.2: Monthly source-build coupling and build author ratios in the studied forges.