Noname manuscript No.
(will be inserted by the editor)

A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance

Shane McIntosh · Meiyappan Nagappan · Bram Adams · Audris Mockus · Ahmed E. Hassan

Author pre-print copy. The final publication is available at Springer via:
http://dx.doi.org/10.1007/s10664-014-9324-x

Abstract Build systems specify how source code is translated into deliverables. They require continual maintenance as the system they build evolves. This build maintenance can become so burdensome that projects switch build technologies, potentially having to rewrite thousands of lines of build code. We aim to understand the prevalence of different build technologies and the relationship between build technology and build maintenance by analyzing version histories in a corpus of 177,039 repositories spread across four software forges, three software ecosystems, and four large-scale projects. We study low-level, abstraction-based, and framework-driven build technologies, as well as tools that automatically manage external dependencies. We find that modern, framework-driven build technologies need to be maintained more often and these build changes are more tightly coupled with the source code than low-level or abstraction-based ones. However, build technology migrations tend to coincide with a shift of build maintenance work to a build-focused team, deferring the cost of build maintenance to them.

Keywords Build systems · Software maintenance · Large-scale analysis · Open source

Shane McIntosh · Meiyappan Nagappan · Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
E-mail: mcintosh@cs.queensu.ca, mei@cs.queensu.ca, ahmed@cs.queensu.ca

Bram Adams
Lab on Maintenance, Construction, and Intelligence of Software (MCIS)
Polytechnique Montréal, Canada
E-mail: bram.adams@polymtl.ca

Audris Mockus
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN, USA
and Avaya Labs Research, NJ, USA
E-mail: audris@avaya.com

1 Introduction

Build systems transform the source code of a software system into deliverables. They describe the process by which software can be incrementally rebuilt by orchestrating compilers, preprocessors, and other tools, allowing developers to focus on making source code changes. A stable build system is crucial to the timely delivery of software, boosting team productivity by: (1) generating versions of the software for integration and feature testing; (2) automating the complex task of packaging and deploying software with the correct versions of required libraries, documentation, and data files; and (3) testing code changes for regression by executing automated tests.

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. Contemporary software development techniques like *continuous integration*, i.e., the practice of routinely downloading the latest source code changes onto dedicated servers to ensure that the code base is free of compilation and test failures would not be possible without a robust build system. The recent practice of *continuous delivery* (Humble and Farley, 2010), where new releases of a software system must be provided within minutes or hours rather than days or months, as used by Facebook or Google would not be possible either. Indeed, Neville-Neal speculates that the build system is one of the most important development tools (Neville-Neal, 2009).

Although modern Integrated Development Environments (IDEs) provide support for building simple applications, complex software systems still require manually maintained build systems (Neitsch et al, 2012; Smith, 2011). There are dozens of build technologies available for developers to select from,[1] each with its own nuances. These technologies adopt various design paradigms. The four most common ones are (Smith, 2011):

1. *Low-level* technologies that require explicitly defined build dependencies between each input and output file [e.g., `make` (Feldman, 1979)].
2. *Abstraction-based* technologies that use high-level project information, such as the project name and the list of files to build, to generate low-level specifications (e.g., CMake[2]).
3. *Framework-driven* technologies that eliminate the "boilerplate" dependency expressions that are typical of low-level technologies in favour of conventions by expecting that input and output files appear in predefined locations (e.g., Maven[3]).
4. *Dependency management* technologies that are used to automatically manage external API dependencies (e.g., Ivy[4]).

Prior research on build systems has shown that: (1) they require non-trivial maintenance effort (McIntosh et al, 2011) in order to stay in sync with the source code that it builds (Adams et al, 2008; McIntosh et al, 2012), and (2) when the maintenance effort associated with the build system grows unwieldy, development teams

[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/`

opt to migrate to a different (perceived to be superior) build technology (Suvorov et al, 2012). Furthermore, anecdotal evidence[5,6] 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 selection 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 each build technology is adopted and its relationship to build maintenance. 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 three dimensions:

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

Build Maintenance: Surprisingly, 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 specifications. Indeed, for systems implemented using Java and Ruby, a large portion of build specification churn is spent on external dependency management.

Build Technology Migration: 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.

1.1 Paper Organization

The remainder of the paper is structured as follows. Section 2 introduces the studied build technology paradigms. Section 3 motivates our research questions, while Section 4 describes our approach to mine version histories to address them. The results of our case studies on software forges, ecosystems, and large-scale projects are presented with respect to the studied dimensions in Sections 5, 6, and 7. Threats to the validity of our study are disclosed in Section 8. Section 9 surveys related work. Finally, Section 10 draws conclusions.

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

[6] `http://argouml.tigris.org/ds/viewMessage.do?dsForumId=450&dsMessageId=2618367`

2 Build Technology Paradigms

We use the term *build system* to refer to specifications that outline how a software system is assembled from its sources. The *build process*, i.e., the act of assembling a software system, is typically split into four steps. First, a set of user-selected or environment-dictated build tools and features are selected during the configuration step. Next, the compiler and other commands that produce deliverables are executed in an order such that dependencies among them are not violated by the construction step. The certification step follows, automatically executing tests to ensure that the produced deliverables have not regressed. Finally, the packaging step bundles certified deliverables together with required libraries, documentation, and data files.

Build systems are supported by a variety of technologies that subscribe to different design paradigms. In this paper, we study the four most common paradigms (Smith, 2011). We briefly introduce each of the studied paradigms below. Detailed information about the studied technologies can be found in Appendix A.

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 (Feldman, 1979) 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 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, etc). 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.

2.3 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, 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.

2.4 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.

3 Research Questions

The goal of our study is to better understand: (1) build technology adoption rates, (2) whether build maintenance is influenced by technology choice, and (3) whether technology migration can help to reduce the burden of build maintenance. To do so, we address five research questions using rigourous statistical analysis of 177,039 repositories. We define and motivate our research questions as they relate to the three dimensions of our study below:

3.1 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 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 address this gap, we formulate the following two research questions:

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

(RQ2) *Is choice of build technology impacted by project characteristics?*
The flexibility of build technologies enables use cases beyond those for which they were designed. For example, the `make` technology was not intended for use in large systems (Feldman, 1979), nor for use in the recursive paradigm that is frequently adopted (Miller, 1998). 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.

3.2 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) *Is build technology choice associated with build change activity?*
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 (Hochstein and Jiao, 2011). We are interested in studying whether build technology choice can have an influence on the build "tax".

(RQ4) *Is build technology choice associated with the overhead on source code development?*
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 (McIntosh et al, 2011). We are interested in studying whether the development overhead of build maintenance is influenced by technology choice.

3.3 Build Technology Migration

Any build system requires maintenance, which can quickly become unwieldy (Neundorf, 2010). Software teams take on build migration projects to counteract this, where

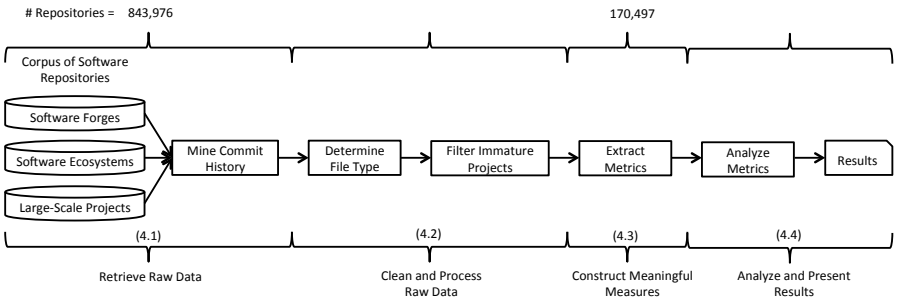


Fig. 1 Overview of our approach.

build specifications are reimplemented, often using different (perceived to be superior) build technologies [e.g., MySQL (Grimmer, 2010) and KDE (Neundorf, 2010)]. 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 (Suvorov et al, 2012). 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 (Suvorov et al, 2012). To assess the impact of build technology migration on build maintenance, we formulate the following research question:

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

4 Approach

Figure 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 (Mockus, 2007). We describe each of the four steps below.

4.1 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

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 (Mockus, 2009). We describe the corpus of repositories used in this study and explain our extraction, stratification, and mining approaches below.

4.1.1 Corpus of Software Repositories

Table 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 (Mockus, 2009). 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).

4.1.2 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.

Table 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 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.

4.2.1 Determine File Type

We mark each commit as changing either source, build, both, or neither. In our prior work, we categorized source and build files semi-automatically (McIntosh et al, 2011), 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.[7] We have made our extended version available online.[8] An overview of the filename conventions that we map to each technology is given in Table 2.

[7] `https://github.com/github/linguist/`

[8] `http://sailhome.cs.queensu.ca/replication/shane/EMSE2013/`

Table 2 The adopted file name conventions for each build technology.

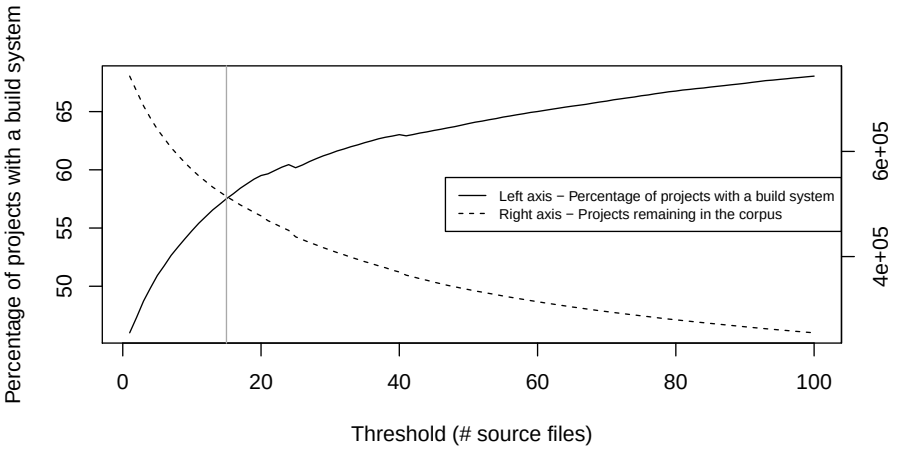
Category	Technology	Conventions
Low-Level	Ant	`build.xml`, `build.properties`
	Jam	`[Jj]amfile`, `*.jam`
	Make	`(GNU)?[Mm]akefile`, `*.mk`, `*.mak`, `*.make`
	Rake	`[Rr]akefile`, `*.rake`,
	SCons	`SConstruct`, `SConscript`, `*.scons`
Abstraction-Based	Autotools	`[Cc]onfigure.(ac|in)`, `ac(local|site).m4`, `[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 Filter Immature Projects

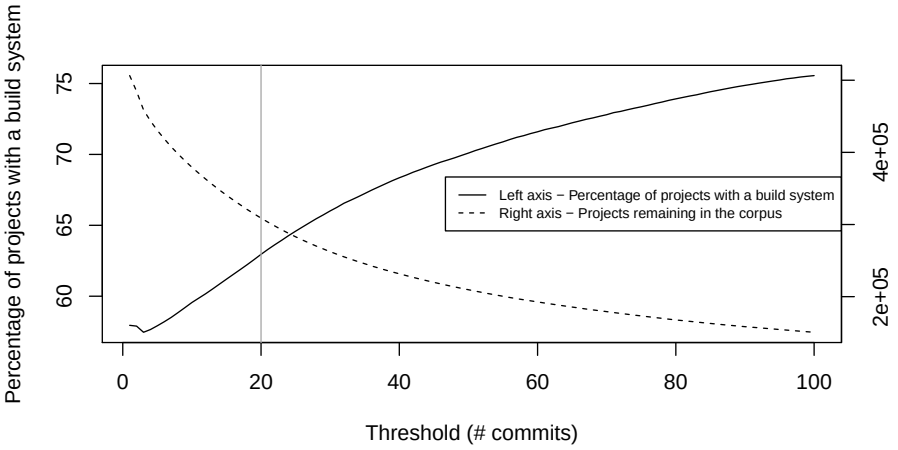
Software forges often contain projects that have not yet reached maturity. We apply three filters to remove repositories that: (1) do not represent software projects, or (2) are too small to require a build system and are hence not of interest in this study.

F1. Select a threshold for project size (measured in number of source files). Figure 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 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 2b shows that selecting a 20 commit cutoff reduces the corpus size to 306,798 repositories, while increasing the number of 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 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 (Savage, 2010). For example, build systems for web applications are necessary to drive continuous delivery (Humble and Farley, 2010), 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 paper, we filter away projects without detected build systems.



(a) Project size.



(b) Development activity.

Fig. 2 Threshold plots for filtering the corpus of repositories.

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.

4.3 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 5, 6, and 7.

4.4 Analyze and Present Results

After extracting metric values, we analyzed them using various visual aids such as line graphs, boxplots, and beanplots (Kampstra, 2008). These figures are also discussed in more detail in Sections 5, 6, and 7.

5 Build Technology Adoption

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

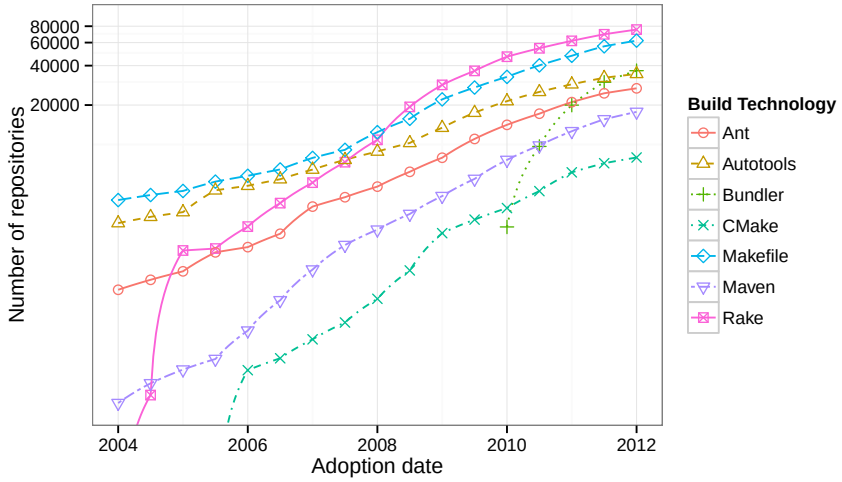
(RQ1) *Which build technologies are broadly adopted?*

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 1 and Figure 3. We discuss our results with respect to the studied forges, ecosystems, and large-scale projects below.

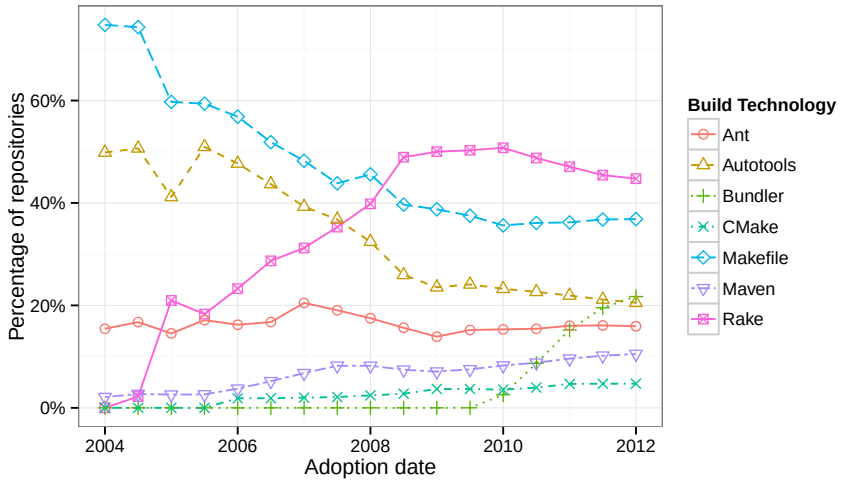
(RQ1-1) Diversity in technology adoption

Software forge repositories are rarely coupled to each other. Hence, we expect diversity in software forge build technology adoption. Table 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 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.



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



(b) Percentage of repositories.

Fig. 3 Build technology adoption over time.

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

and 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 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 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 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.

Summary: *While many projects use traditional technologies like* `make` *and Autotools, language-specific technologies like Rake and Bundler capture more market share (Observation 1).*
Implications: *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 gaining popularity and should also be considered.*

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

To address this 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

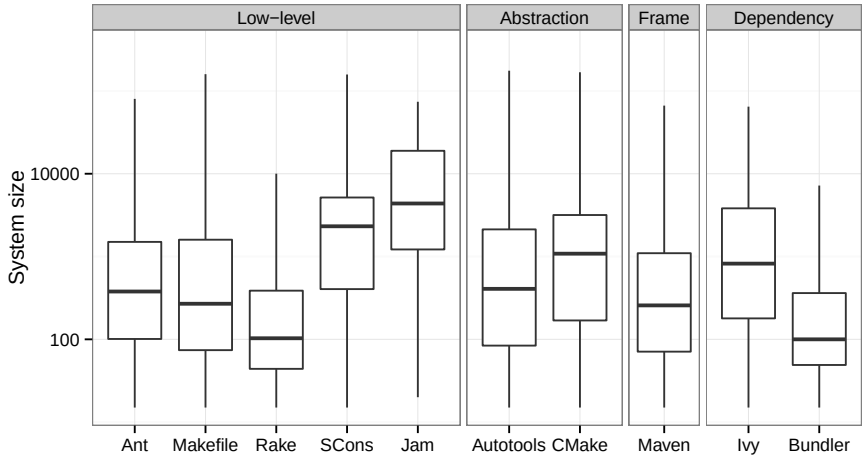


Fig. 4 Size of the source code (# files) per repository in the forges and ecosystems.

that finer-grained metrics, such as SLOC, show similar evolutionary patterns in large datasets (Herraiz et al, 2006).

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 (Miller, 1981) 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.

Observation 2 – Large repositories tend to adopt newer technologies earlier than smaller ones: Figure 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 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

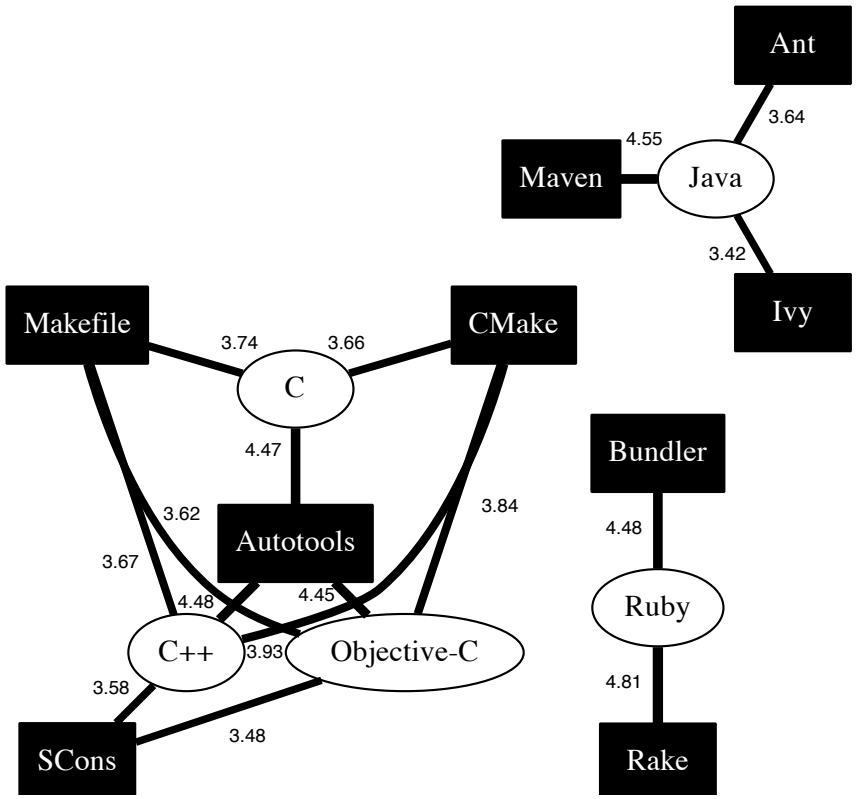


Fig. 5 Statistically significant ($p < 10^{-100}$) co-occurrences of build technology (black boxes) and programming language (white ovals) on a fitted Poisson model. The higher the log odds ratio presented above each edge, the higher the likelihood of a non-coincidental relationship.

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 ~ 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 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.

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

Observation 3 – Programming language choice is related to 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 similar. However, the formation of clustered groups of technologies around programming languages in Figure 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 3 shows that language-specific technologies like Rake, Bundler, Ant, and Maven have grown rapidly in the past few years, while Figure 5 confirms that Rake and Bundler are de facto build technologies for Ruby repositories, and Ant and Maven share the bulk of Java repositories.

Summary: *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).*
Implications: *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.*

5.1 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 language-specific build technologies: Language-specific tools are almost unanimous choices for Java and Ruby systems. Figure 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

Table 3 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 frequently 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.

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 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 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 (Miller, 1998) 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.

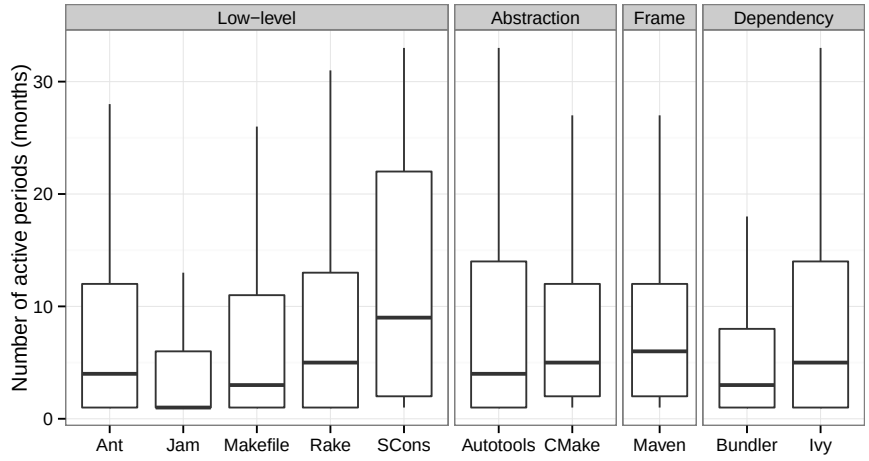
6 Build Maintenance

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

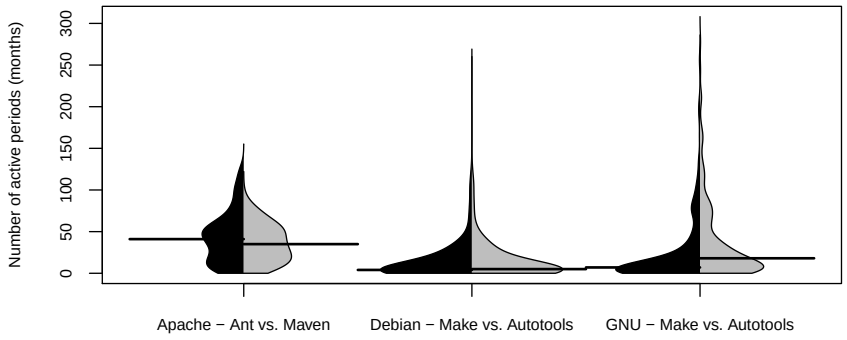
(RQ3) *Is build technology choice associated with build change activity?*

We select metrics that measure three dimensions of build change activity, and calculate them on a monthly basis. Table 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

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



(a) Software forges.



(b) Software ecosystems.

Fig. 6 Number of active periods (months) per repository in the forges and ecosystems.

and rate of change 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 (Graves et al, 2000) and build system domains (McIntosh et al, 2012).

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 6 shows the distribution of commit activity (in number of months) for a specific technology. The upper end of the boxes in Figure 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.

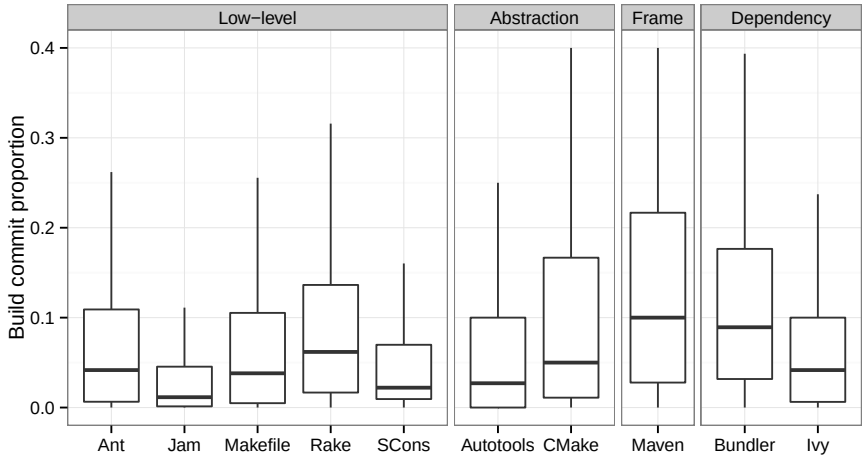
We focus our ecosystem studies on comparing Ant and Maven in Apache, and Make and Autotools in Debian and GNU, since the ecosystems mostly converge on those build technologies. Figure 6b compares the distributions of active months in the studied ecosystems using beanplots (Kampstra, 2008). Beanplots are boxplots in which the vertical curves summarize the distributions of different data sets. The horizontal lines indicate the median values. Figure 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 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 (Bauer, 1972) instead of Tukey HSD tests to compare them ($\alpha = 0.01$).

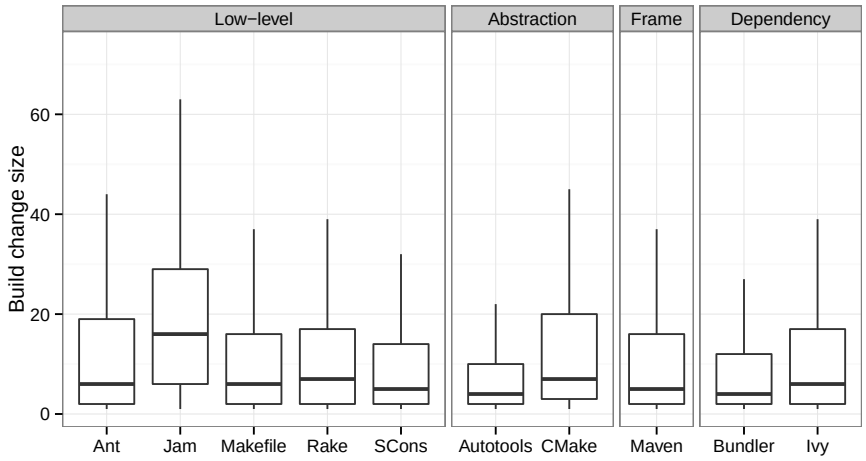
Figure 7 shows the distributions of metric values in the forges. To ensure that each repository is equally considered in our analysis, we select the median value for each metric from each repository. We complement our median-based analysis by performing a longitudinal analysis of each metric in the forges. 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 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).

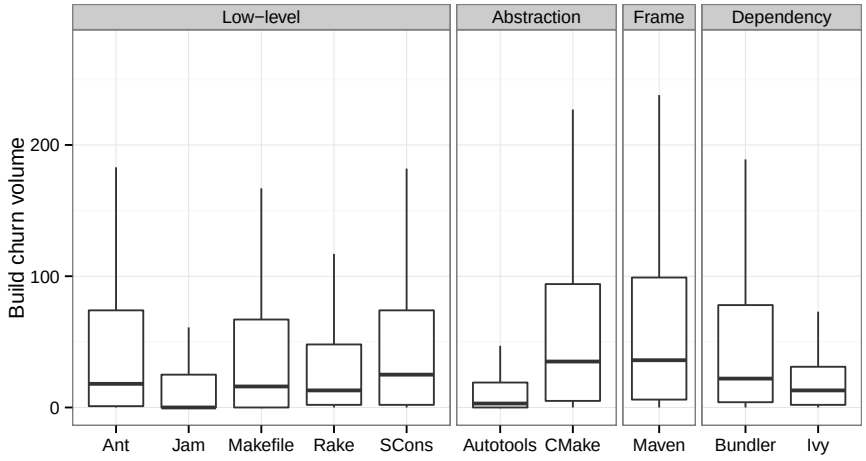
Figure 7a suggest and the Tukey HSD test confirm that the median Autotools build commit proportion tends to be lower than that of the other technologies in the abstraction, low-level, and dependency management categories. Furthermore, Autotools never appears in the top three ranks of the 12-month Tukey 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 imple-



(a) Build commit proportion.



(b) Build change size.



(c) Build churn volume.

Fig. 7 Median build commit proportion, size, and churn in the studied forges.

ment 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 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 7b, while Maven and SCons make more frequent appearances in the top three ranks of the monthly analysis (see Appendix B).

Figure 7c shows that the median build churn volume for framework-driven specifications is higher than that of the other studied technologies. Tukey HSD tests of the median samples confirm that the Maven rates are indeed 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 Appendix B). CMake only drops to the third rank in the seventh month, appearing in the top two ranks for all other months.

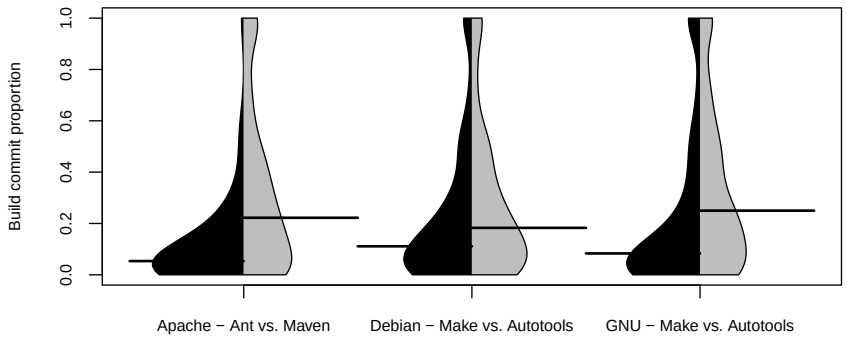
Corroborating our findings in the software forges, Figure 8a shows that Maven tends to require a larger proportion of monthly build changes than Ant in the Apache ecosystem. Indeed, while Figure 8b suggests that Maven commits tend to be smaller than Ant commits in the Apache ecosystem, Figure 8c shows 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.

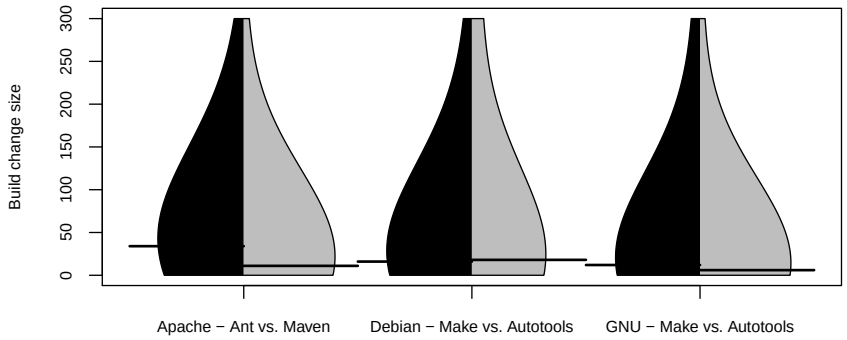
Summary: *Framework-driven technologies like Maven tend to have a higher build commit proportion and induce more build churn than low-level or abstraction-based technologies (Observation 4).*
Implications: *While modern build technologies provide additional features, the development teams adopting them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

6.1 Discussion

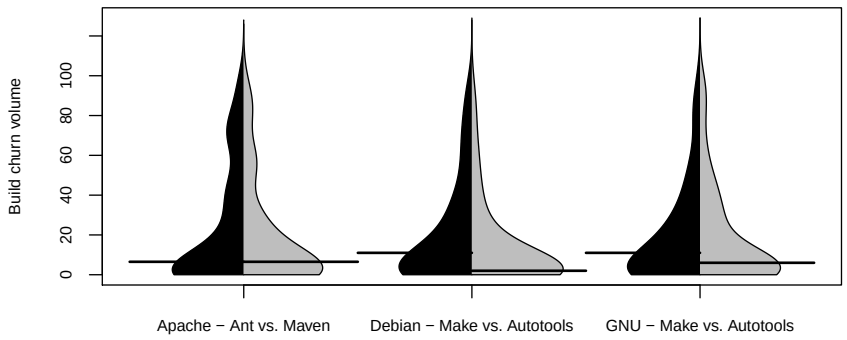
While the sizes of Jam and SCons changes are noteworthy, in addition to the tendency of being used in larger systems (Observation 2), they are also low-level technologies, and are therefore expected to be more verbose than the other technologies. Ant and Maven change sizes may be inflated because of the verbose nature of the XML markup (Lawrence, 2004). The verbosity of CMake changes is surprising,



(a) Build commit proportion.



(b) Build commit sizes.



(c) Build churn volume.

Fig. 8 Median build commit proportion, size, and churn in the studied ecosystems.

Table 4 Build maintenance overhead metrics.

Metric	Description	Rationale
Source-build coupling	The logical coupling (Equation 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 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.

since CMake is an abstraction-based technology – a quality that one would expect to decrease change size.

As described in the discussion of Section 5, 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) *Is build technology choice associated with the overhead on source code development?*

Similar to our prior work (McIntosh et al, 2011), we select metrics that measure build maintenance overhead using *logical coupling* (Gall et al, 1998), which is calculated as shown below:

$$LC(source \Rightarrow build) = \frac{Support(source \cap build)}{Support(source)} \quad (1)$$

Note that $Support(X)$ in Equation 1 is the number of commits that satisfy the clause X.

Table 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 (Bird et al, 2009b). 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. Insofar as developers

use this feature, our build author ratio analysis does not lose the original authorship information.

Figure 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 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.

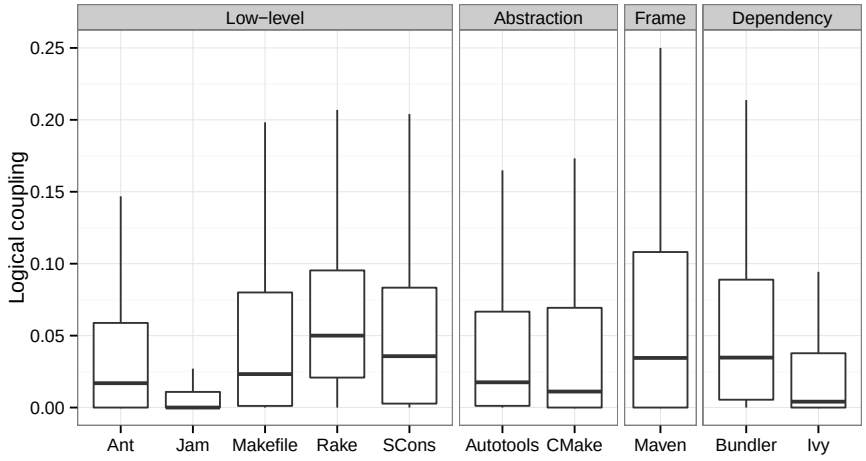
Figure 9b shows that the Maven changes tend to be more evenly dispersed among developers than changes of other technologies are. Tukey HSD tests confirm that a larger proportion of developers for Maven projects make build changes than developers using the other technologies. The median Maven build author 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 10a shows that Maven and Autotools tend to be more tightly coupled to source changes than Ant and `make`. Furthermore, Figure 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.

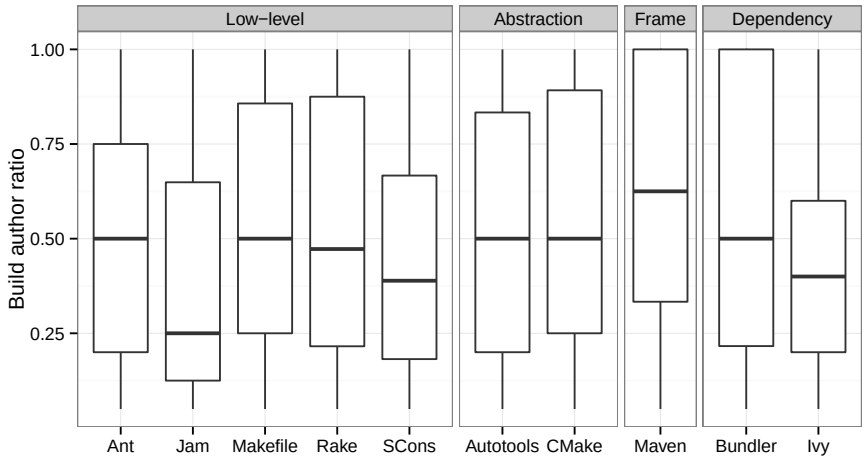
The finding that is most consistent across the software forges and ecosystems is that Maven changes tends to be tightly coupled to source code changes. To that end, a larger proportion of the development team tends to become involved in maintaining the Maven specifications.

Observation 6 – Build change more often co-occurs with source change than without: Figure 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 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



(a) Source-build coupling.

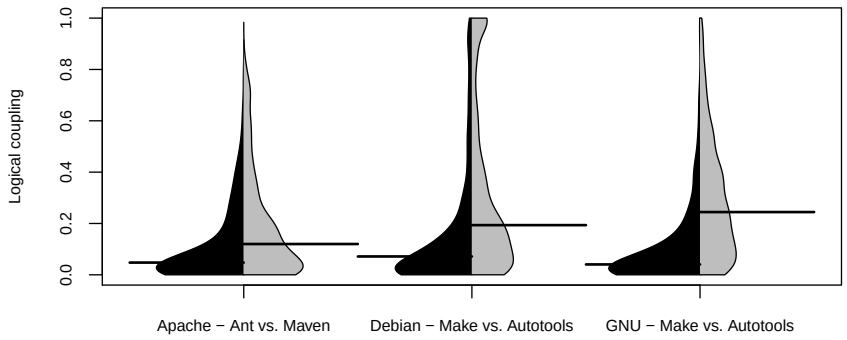


(b) Build author ratio.

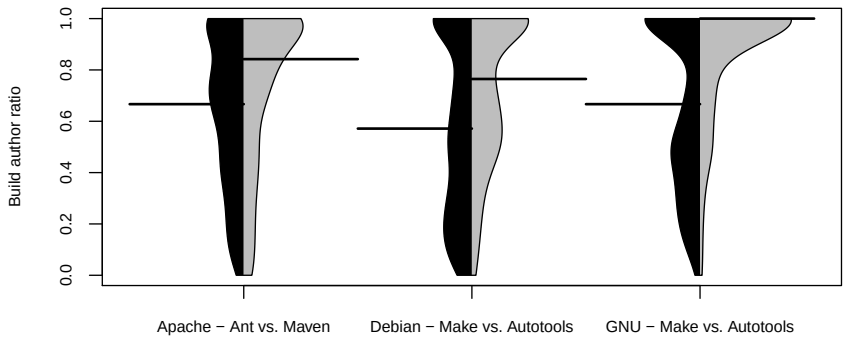
Fig. 9 Median source-build coupling and build author ratios in the studied forges.

indistinguishable from each other. Furthermore, the proportions of build changes that are accompanied with source changes shown in Figure 11b indicate that, with the exception of Maven, build changes tend to occur more frequently with source changes than without.

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



(a) Source-build coupling.

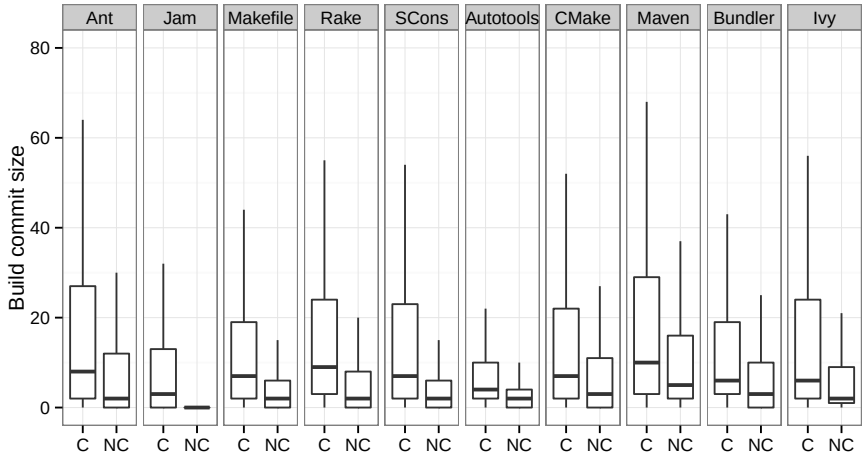


(b) Build author ratio.

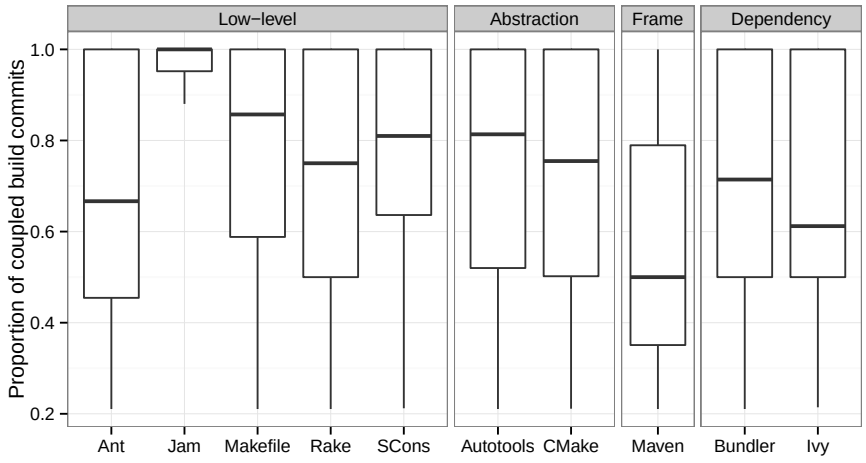
Fig. 10 Median source-build coupling and build author ratios in the studied ecosystems.

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 analy-



(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.

Fig. 11 Comparison of coupled and not coupled build changes.

sis on the most active build technologies of each large-scale project. Table 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 (Neundorf, 2010; Suvorov et al, 2012), hence we study trends with respect to both technologies.

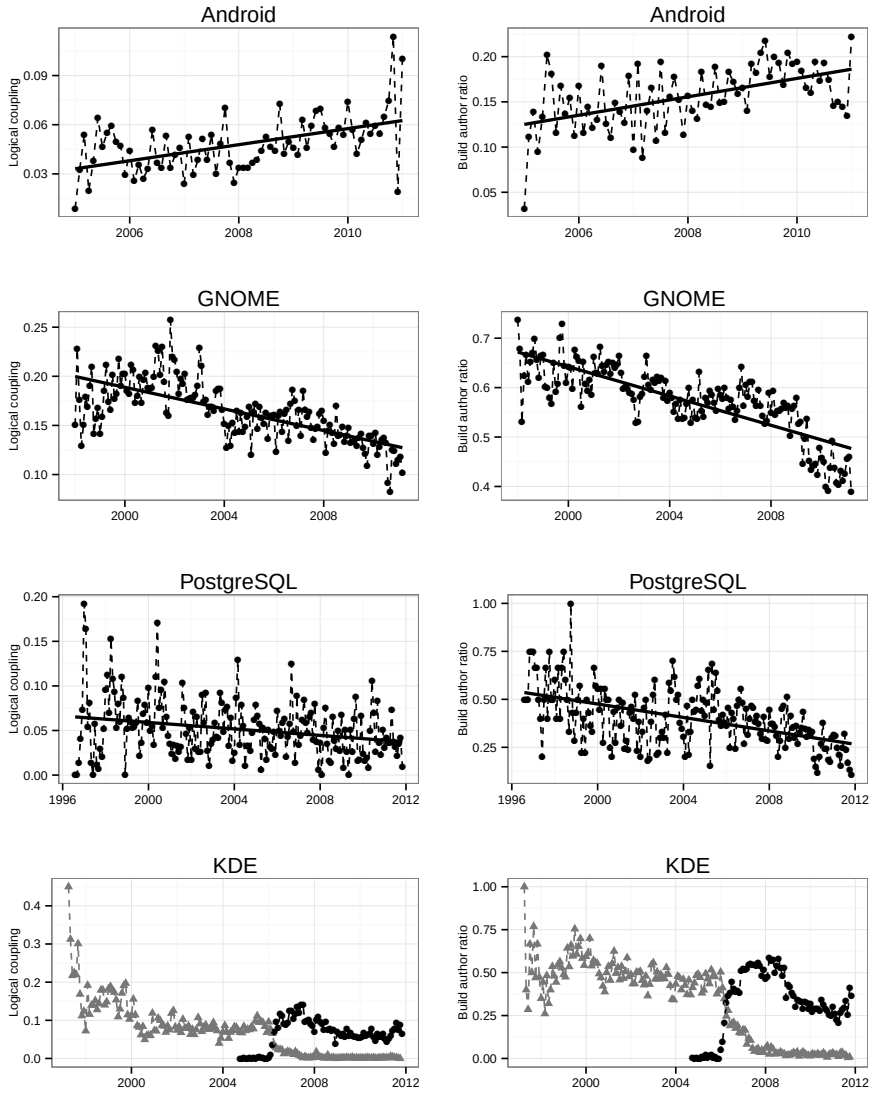


Fig. 12 Monthly source-build coupling rate (left) and monthly build author ratio (right) in Android (`make`), GNOME (Autotools), PostgreSQL (Autotools), and KDE (Autotools in grey, CMake in black).

Figure 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 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%.

Summary: *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.*
Implications: *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.*

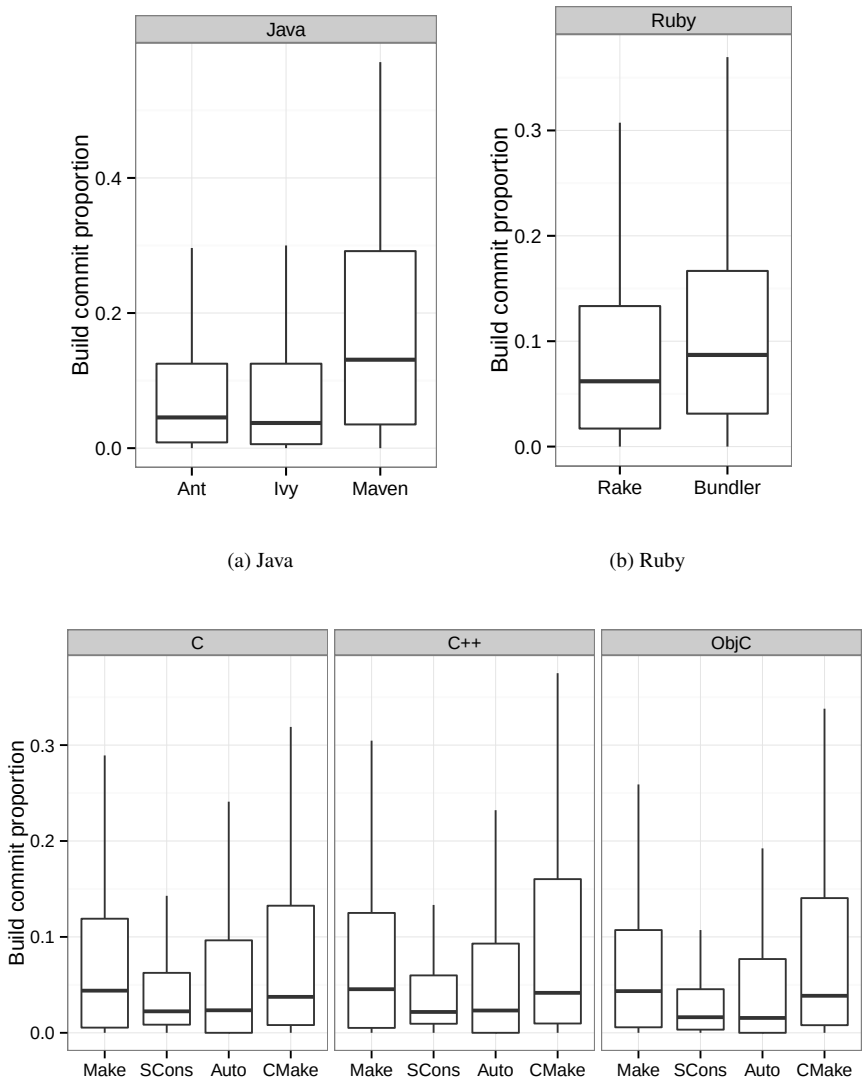
6.2 Programming Language Centric Technology Analysis

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 5, 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 5).

We first categorize the technologies typically used by a programming language by examining Figure 5. In doing so, we produce the below mapping:

Java $\longrightarrow$ Ant, Ivy, Maven
C, C++, Objective-C $\longrightarrow$ `make`, Autotools, SCons, CMake
Ruby $\longrightarrow$ 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 5, 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 1) metrics of each labelled repository to compare the use of build technologies for each programming language separately.



(a) Java

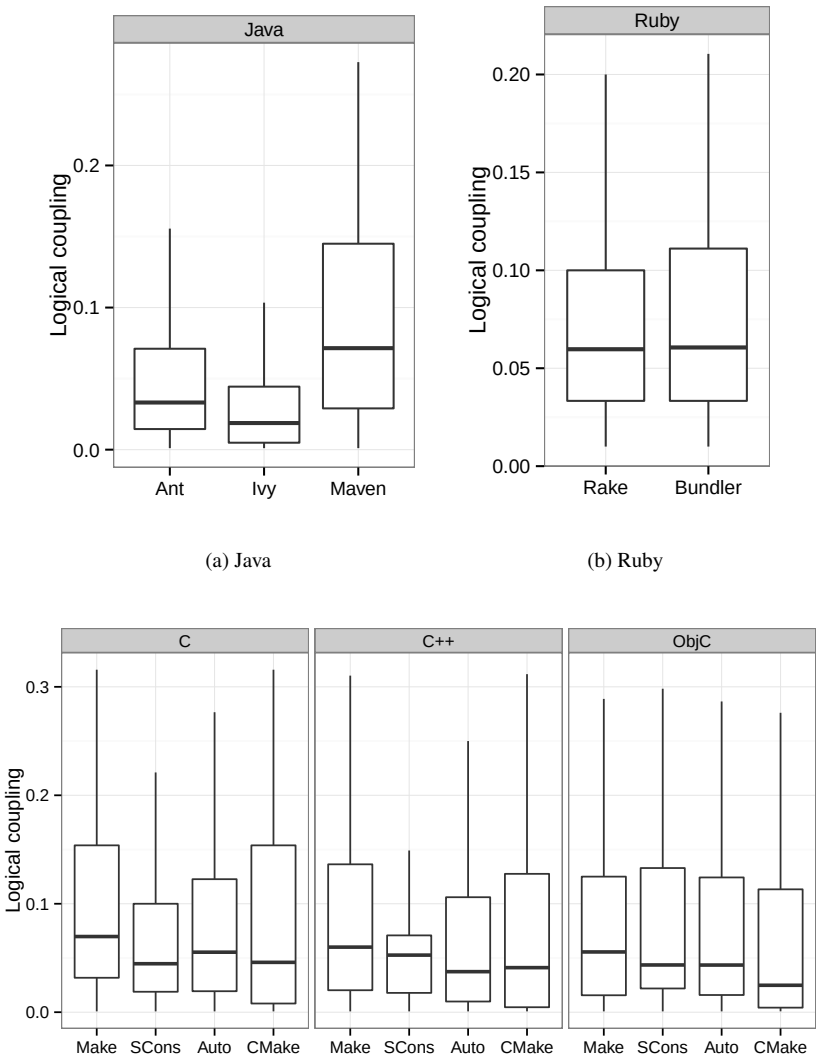
(b) Ruby

(c) C family

Fig. 13 Build commit proportion in the studied forges classified by source languages used.

6.2.1 Language-Specific Build Commit Proportion

Observation 8 – External dependency management specifications require plenty of maintenance: Figure 13 shows the monthly build commit proportion for each group of programming languages. Figure 13a confirms that Maven specifications do in-



(a) Java

(b) Ruby

(c) C family

Fig. 14 Source-build coupling in the studied forges classified by source languages used.

deed 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 changes 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).

As shown in Figures 13a and 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.

6.2.2 Language-Specific Source-Build Coupling

Figure 14 shows the source-build coupling between build technologies and specific programming languages. Figure 14a shows that similar to the overall coupling in Figure 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 13c shows that CMake specifications have a higher commit proportion than the other technologies for C family languages, Figure 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.

Summary: *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.*
Implications: *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.*

6.3 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.

7 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?*

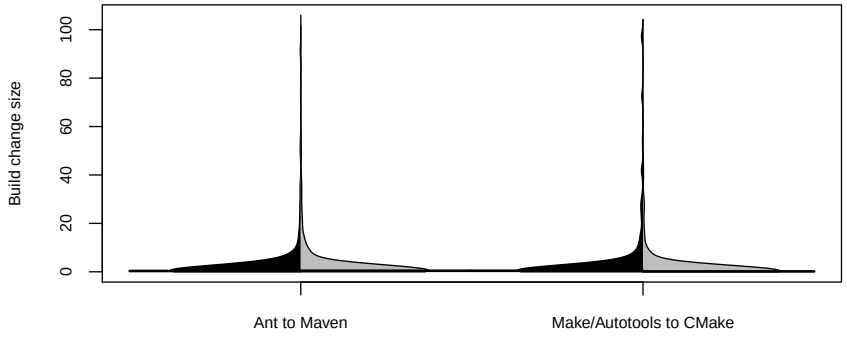
A recent trend suggests that projects are migrating towards CMake (Grimmer, 2010; Linden Labs, 2010; Neundorf, 2010) and Maven (Ebersole, 2007). Hence, we focus our migration study on these technologies. Specifically, we compare median monthly churn rate, source-build coupling, and build author ratios 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 (2012)], 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.

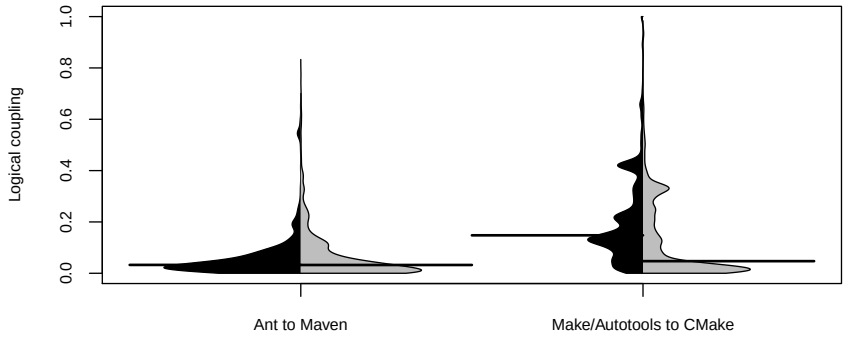
Observation 9 – Build technology migration often pays off: Figure 15 shows that, despite Maven projects typically having higher source-build coupling rates (Observation 5), migration from Ant to Maven tends to have little impact on churn rate or source-build coupling. In the projects that have migrated, the median monthly churn rate and source-build coupling rate of Maven is almost identical to those of Ant (Figures 15a and 15b). Also contrary to Observation 5, we find that the build author ratio tends to drop as projects migrate from Ant to Maven (Figure 15c). Wilcoxon signed rank tests of build author ratio confirm that the results are statistically 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 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.

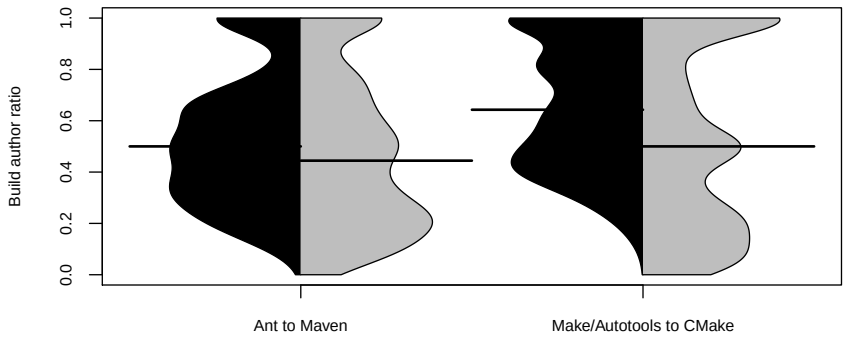
Complementing our software forge findings, Figure 16a shows that the median monthly churn rate in the studied ecosystems is rarely impacted by migration projects. Figure 16b shows that again source-build coupling tends to drop after a migration to CMake, however is rarely impacted by migration to Maven. 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.



(a) Build churn rates.

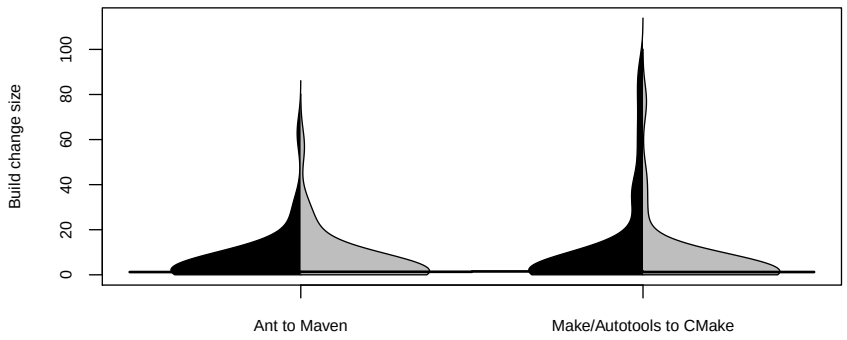


(b) Logical coupling.

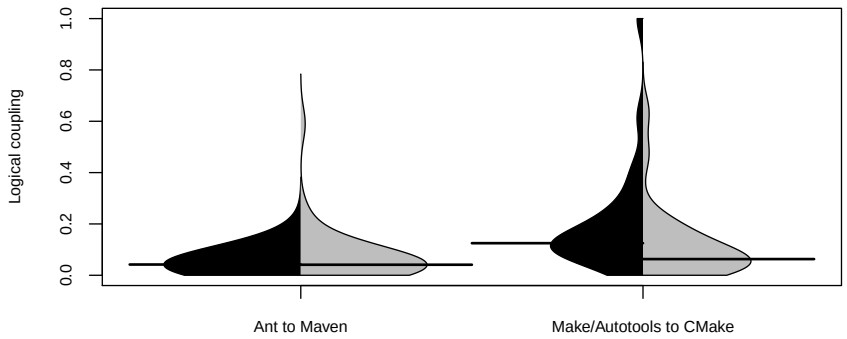


(c) Build author ratio.

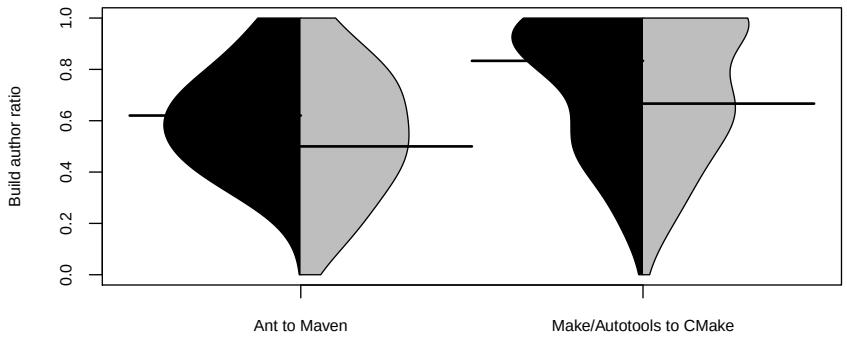
Fig. 15 Build technology migration in the studied forges



(a) Build churn rates.



(b) Logical coupling.



(c) Build author ratio.

Fig. 16 Build technology migration in the studied ecosystems

7.1 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 12 are decreasing for both Autotools and CMake. 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 12 shows decreasing trends in KDE build author ratio for both Autotools and CMake build systems. After an early period of highly 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 initial period of growth in 2007, the KDE CMake decreases further, dropping as low as 24%.

Summary: *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.*
Implications: *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.*

8 Threats to Validity

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

8.1 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 (McIntosh et al, 2011). There is a well-documented lack of well-linked data (Bird et al, 2009a; Nguyen et al, 2010) that prevents us from grouping related commits together. Regardless, our analysis draws on comparisons among repositories, not on the absolute values of the metrics.

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.

8.2 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.

8.3 Reliability Validity

We use a modified version of the Github Linguist tool[11] to conservatively classify files as source or build files. We have made our extended version available online.[12] 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.

8.4 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.

9 Related Work

We present the related work with respect to build system maintenance, tool assistance for build maintenance, and build migration projects.

9.1 Build System Maintenance

There have been several recent studies on build system maintenance. Our prior work shows that source code and build system tend to co-evolve (Adams et al, 2008; McIntosh et al, 2012). Nadi *et al.* show that inconsistencies between the source code and build system of the Linux kernel can cause defects (Nadi and Holt, 2011) and propose a method for automatically detecting these inconsistencies (Nadi and Holt, 2012). Indeed, Neitsch *et al.* find that abstractions tend to "leak" between source and build domains (Neitsch et al, 2012). In order to help Linux developers avoid these inconsistencies, Dietrich *et al.* propose a technique for extracting the mapping of features to source code from the Linux build system (Dietrich et al, 2012). Hochstein *et al.* and our prior work show that build maintenance imposes a non-trivial "tax" on the software development (Hochstein and Jiao, 2011; McIntosh et al, 2011). In this paper, we find that build migration and team specialization can offer some relief for many developers by offloading the responsibility of build maintenance on a dedicated team of build maintainers.

[11] `https://github.com/github/linguist/`

[12] `http://sailhome.cs.queensu.ca/replication/shane/EMSE2013/`

9.2 Tool Assistance for Build Maintenance

Researchers have created several tools to assist with build maintenance. Adams *et al.* (Adams et al, 2007) and Tu and Godfrey (Tu and Godfrey, 2002) developed tools to visualize and query build dependencies. Tamrawi *et al.* propose a technique for visualizing and verifying build dependencies using symbolic dependency graphs (Tamrawi et al, 2012). Al-Kofahi *et al.* propose a tool for extracting the semantics related to `make` specification changes (Al-Kofahi et al, 2012). In this paper, we do not propose a tool, but rather provide empirical evidence of the relationship between build technology and build maintenance.

9.3 Build Technology Migration

Recent research has also studied build system migration efforts, i.e., reimplementation of a build system using a different build technology or build design methodology. Suvorov *et al.* studied failed and successful build migrations, reporting that failed migrations often do not gather sufficient requirements prior to prototyping (Suvorov et al, 2012). Zadok found that the migration from `make` to Autotools in the Berkeley Automounter project reduced build code and accelerated development (Zadok, 2002). Complementing Zadok's work, we provide further evidence that build migration projects can help to reduce the build maintenance burden.

10 Conclusions

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 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.[13] 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 paper, 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. 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 dimensions 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

[13] `http://en.wikipedia.org/wiki/List_of_build_automation_software`

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.
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[14] 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.

References

Adams B, De Schutter K, Tromp H, Meuter W (2007) Design recovery and maintenance of build systems. In: Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM), pp 114–123

Adams B, Schutter KD, Tromp H, Meuter WD (2008) The Evolution of the Linux Build System. Electronic Communications of the ECEASST 8

Al-Kofahi JM, Nguyen HV, Nguyen AT, Nguyen TT, Nguyen TN (2012) Detecting Semantic Changes in Makefile Build Code. In: Proc. of the 28th Int'l Conf. on Software Maintenance (ICSM), pp 150–159

Bauer DF (1972) Constructing Confidence Sets Using Rank Statistics. Journal of the American Statistical Association 67(339):687–690

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009a) 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), pp 121–130

[14] `http://travis-ci.org/`

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009b) The Promises and Perils of Mining Git. In: Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)

Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D (2012) A Robust Approach for Variability Extraction from the Linux Build System. In: Proc. of the 16th Int'l Software Product Line Conference (SPLC), pp 21–30

Ebersole S (2007) Maven migration. http://lists.jboss.org/pipermail/hibernate-dev/2007-May/002075.html, last viewed: 18-Mar-2010

Feldman S (1979) Make - a program for maintaining computer programs. Software - Practice and Experience 9(4):255–265

Gall H, Hajek K, Jazayeri M (1998) Detection of logical coupling based on product release history. In: Proc. of the 14th Int'l Conf. on Software Maintenance (ICSM), pp 190–198

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting Fault Incidence using Software Change History. Transactions on Software Engineering (TSE) 26(7):653–661

Grimmer L (2010) Building MySQL Server with CMake on Linux/Unix. `http://www.lenzg.net/archives/291-Building-MySQL-Server-with-CMake-on-LinuxUnix.html`, Last viewed: 20-Aug-2010

Herraiz I, Robles G, Gonzalez-Barahona J, Capiluppi A, Ramil J (2006) 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), pp 213–221

Hochstein L, Jiao Y (2011) The cost of the build tax in scientific software. In: Proc. of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 384–387

Humble J, Farley D (2010) Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley

Kampstra P (2008) Beanplot: A boxplot alternative for visual comparison of distributions. Journal of Statistical Software, Code Snippets 28(1):1–9, URL `http://www.jstatsoft.org/v28/c01/`

Lawrence R (2004) The space efficience of XML. Information and Software Technology (IST) 46(11):753–759

Linden Labs (2010) CMake. `http://wiki.secondlife.com/wiki/CMake`, Last viewed: 20-Aug-2010

McIntosh S, Adams B, Nguyen THD, Kamei Y, Hassan AE (2011) An Empirical Study of Build Maintenance Effort. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE), pp 141–150

McIntosh S, Adams B, Hassan AE (2012) The evolution of Java build systems. Empirical Software Engineering 17(4-5):578–608

Miller P (1998) Recursive make considered harmful. In: Australian Unix User Group Newsletter, vol 19, pp 14–25

Miller RG (1981) Simultaneous Statistical Inference. Springer

Mockus A (2007) Software support tools and experimental work. In: Proc. of the Int'l Conf. on Empirical Software Engineering Issues: Critical Assessment and Future Directions, pp 91–99

Mockus A (2009) 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), pp 11–20

Nadi S, Holt R (2011) Make it or Break it: Mining Anomalies in Linux Kbuild. In: Proc. of the 18th Working Conf. on Reverse Engineering (WCRE), pp 315–324

Nadi S, Holt R (2012) Mining Kbuild to Detect Variability Anomalies in Linux. In: Proc. of the 16th European Conf. on Software Maintenance and Reengineering (CSMR), pp 107–116

Neitsch A, Wong K, Godfrey MW (2012) Build System Issues in Multilanguage Software. In: Proc. of the 28th Int'l Conf. on Software Maintenance, pp 140–149

Neundorf A (2010) Why the KDE project switched to CMake – and how (continued). `http://lwn.net/Articles/188693/`, last viewed: 06-Mar-2010

Neville-Neal GV (2009) Kode vicious: System changes and side effects. Communications of the ACM 52(4):25–26

Nguyen THD, Adams B, Hassan AE (2010) A Case Study of Bias in Bug-Fix Datasets. In: Proc. of the 17th Working Conf. on Reverse Engineering (WCRE), pp 259–268

Savage B (2010) Build Systems: Relevancy of Automated Builds in a Web World. `http://www.brandonsavage.net/build-systems-relevancy-of-automated-builds-in-a-web-world/`

Smith P (2011) Software Build Systems: Principles and Experience, 1st edn. Addison-Wesley

Suvorov R, Nagappan M, Hassan AE, Zou Y, Adams B (2012) 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), pp 160–169

Tamrawi A, Nguyen HA, Nguyen HV, Nguyen T (2012) Build Code Analysis with Symbolic Evaluation. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE), pp 650–660

Tu Q, Godfrey M (2002) The build-time software architecture view. In: Proc. of Int'l Conf. on Software Maintenance (ICSM), pp 398–407

Zadok E (2002) Overhauling Amd for the '00s: A Case Study of GNU Autotools. In: Proc. of the FREENIX Track on the USENIX Technical Conf., USENIX Association, pp 287–297

```
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

Fig. 17 Example low-level technology specifications.

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 17 provides working examples of the five studied low-level build technologies.

Make: One of the earliest build technologies on record is Feldman's `make` tool (Feldman, 1979), 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 17a 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 8. Line 1 of Figure 17a specifies that the `all` target is *phony*, representing an abstract phase in the build process rather than a concrete file in the filesystem.

Jam: Jam provides a more procedural-style structure for target-dependency-recipe tuples. Figure 17b 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.

Ant: Ant borrows the 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 17c 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.

```
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

Fig. 18 Example abstraction-based technology specifications.

SCons: SCons provides several advanced build system features (e.g., implicit dependency tracking for popular programming languages) and allows maintainers to write highly portable build specifications using Python. Line 7 of Figure 17d 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.
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 17e 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 18 provides working examples of the two studied abstraction-based technologies.
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 18a and 18b respectively. Lines 1 and 2 of Figure 18a 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 18b 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.
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 18c 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.
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 19a 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.

```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

Fig. 19 Example Framework-driven and dependency management technology specifications.

Lines 10-18 of Figure 19a 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 19 provides working examples of dependency management in Maven (Figure 19a) and the two studied dependency management technologies (Figures 19b and 19c).

Maven: In addition to providing a framework-driven build environment, Maven doubles as a dependency management technology. Lines 22-26 of Figure 19a provide an example dependency declaration on the JUnit tool, version 3.8.1.

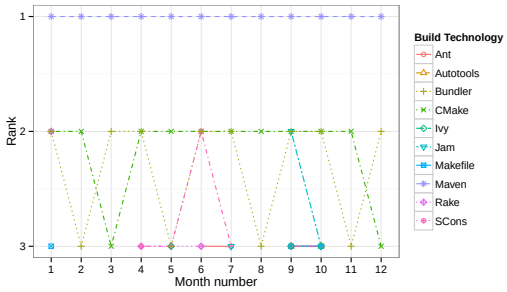
Ivy: Ivy provides dependency management features that are most notably leveraged by Ant. Figure 19b shows an Ivy specification for the same JUnit dependency as depicted in Figure 19a.

Bundler: Bundler provides packaging and dependency management for Ruby applications. Line 1 of Figure 19c 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).

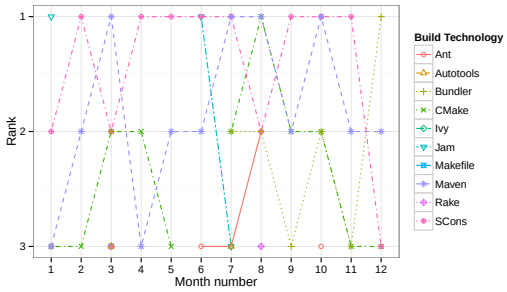
B Additional Build Maintenance Figures

We perform longitudinal analyses of the Tukey HSD ranks for each metric in the forges to complement our median-based analyses in Section 6. Figures 20 and 21 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.[15]

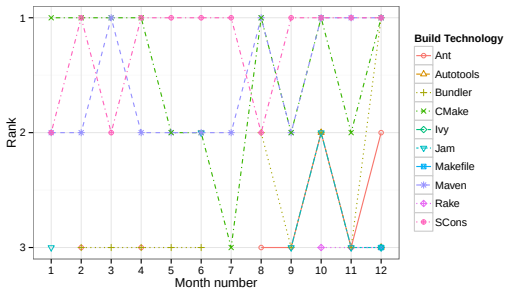
[15] `http://sailhome.cs.queensu.ca/replication/shane/EMSE2013/`



(a) Build commit proportion.

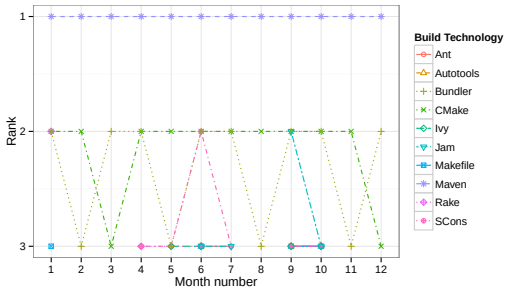


(b) Build change size.

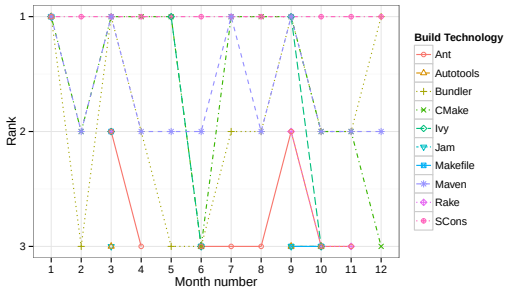


(c) Build churn volume.

Fig. 20 Monthly build commit proportion, sizes, and churn volume in the studied forges.



(a) Logical coupling.



(b) Build author ratio.

Fig. 21 Monthly source-build coupling and build author ratios in the studied forges.