Static Code Analysis Tools: Effects on Development of Open Source Software

Master's Thesis

Bastiaan van Graafeiland

Static Code Analysis Tools: Effects on Development of Open Source Software

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bastiaan van Graafeiland
born in Delft, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Static Code Analysis Tools: Effects on Development of Open Source Software

Author:	Bastiaan van Graafeiland
Student id:	1399101
Email:	`b.vangraafeiland@student.tudelft.nl`

Abstract

Nowadays, many different tools to perform static analysis on software (ASATs) are available. These can be used as standalone tools, but also integrated into code reviews, build processes, or continuous integration. ASATs can be configured by their user and report a list of warnings for each rule that has been violated by the analyzed code. While some research has been performed regarding ASATs, little is currently known about the correlations between use of ASATs and other properties of projects or their communities, or about the extent to which developers solve violations reported by ASATs. In this thesis, we attempt to answer these questions by obtaining information about a large number of relevant open source projects hosted on GitHub. We found that the usage rate for ASATs is relatively low, while ASAT usage can be associated with several positive changes in other properties; in general, popular and successful projects are more likely to use ASATs. Furthermore, projects that use ASATs typically have a more active community, and receive more contributions. The amount of warnings generated varies between projects, but projects with large code bases tend to have fewer warnings. When looking at the types of warnings reported, not all categories are equally represented; violations of Style Conventions are most common. We also found that warnings of different categories are solved at different rates; warnings with more impact on maintainability were solved faster, while warnings with little impact on correctness or maintainability were left unattended for longer.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. S.T. Erdweg, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

Software development is rarely performed by a single person. As different people have different preferences, a project's code base has the potential to become inconsistent. As a result, it becomes harder for new developers to understand existing code. To prevent this, teams of developers may define a set of code conventions which all team members must conform to. This can be accomplished by the use of Automated Static Analysis Tools (ASATs). Static analysis tools inspect a program's source or byte code, unlike dynamic analysis tools, which inspect a program's behavior during runtime. ASATs have existed since 1977, when Johnson released *Lint*, a command-line tool to detect inconsistencies and inefficiencies in C code[27]. Most ASATs are configurable by the user, who can establish a set of rules which the code must adhere to. If any code is found that violates a rule, the ASAT will output a warning to notify the user. Although ASATs can be used as command-line tools, it is more convenient to integrate them into the text editor which is used for programming. This way, warnings can be displayed in real time at the location of the violating code, much like compiler warnings in IDEs.

ASATs may be used as part of *code reviews*, where one person's code is inspected by another person, such as a colleague. Doing so decreases the manual effort and time taken[43]. The concept of code reviews has first been formalized by Fagan in 1976[17] and was found to be beneficial for software development organizations[1, 6]. Because of the (potentially) larger amount of collaborators, code reviews can be a valuable asset to open source projects[41], and by extension, ASAT as well. A survey performed by Beller et. al found that ASAT usage among open source projects is "common, but not ubiquitous"[8].

On GitHub, an open source hosting service, contributions are generally done through pull requests, a form of code reviews that makes it easier for a project maintainer to merge the contribution into the project. Pull requests are popular, being used by around half of all projects with multiple developers[20]. To assist maintainers with this, GitHub allows automated checks by third party services for every pull request[1]. A popular service used for this is Travis CI[2][46], a Continuous Integration environment for open source projects that is configurable by the user. When enabled, each pull request and commit will trigger Travis to

[1]https://github.com/blog/1227-commit-status-api

[2]http://www.travis-ci.org

build the project, and report the result of the build back to GitHub. This result is clearly displayed for each pull request, making it easier for maintainers to filter out pull requests that cause the build to fail, but also providing instantaneous feedback to contributors, allowing them to solve any problems without having to wait for maintainer's answer. Because Travis works with scripts to build a project, ASATs can easily be added into the mix. Aside from running an ASAT directly in a script, build tools (such as Maven[3]) often have plugins that make it easier to integrate ASATs into the build process (e.g. Maven Checkstyle Plugin[4]). These plugins act as a wrapper around the command-line tool and make the ASAT behave in a manner that is consistent with the other build tools (e.g. Maven can generate an HTML report with Checkstyle warnings like it does with test coverage).

In this study, we will analyze 7 different ASATs: *Checkstyle*, *PMD*, *ESLint*, *JSHint*, *JSCS*, *RuboCop*, and *Pylint*. These have been selected because they also appeared in the study by Beller et al.[8]. With these ASATs, we end up with Java, JavaScript, Ruby, and Python as programming languages for the projects under study. ASATs studied by Beller which we excluded are *JSL* and *FindBugs*, because the former has been discontinued and the latter has a very small userbase compared to the others.

1.1 Research Questions

Beller et. al have conducted a study with regards to ASAT usage in open source projects[8]. In this study, ASAT prevalence was analyzed by means of a qualitative survey, and ASAT configuration characteristics were analyzed on a large number of projects. In addition, a classification was made for a generalization of rules from different ASATs, called the General Defect Classification. However, the prevalence survey was performed on a relatively small amount of only the most popular projects without looking at factors that may influence use rate. This leads to our first research question:
RQ1: Which factors influence ASAT prevalence?

- **RQ1.1:** Is there any difference in ASAT prevalence between projects of different languages?
- **RQ1.2:** Which ASATs are used most?
- **RQ1.3:** Does the use of a build tool/task runner influence the use of ASATs?
- **RQ1.4:** Are projects more likely to use ASATs if they use Continuous Integration?
- **RQ1.5:** Is there any age difference between projects that use ASATs and those that do not?
- **RQ1.6:** Are popular projects more likely to use ASATs?

Other than direct properties of the project itself, it would also make sense if characteristics of the project's community are related to ASAT usage. As the community and amount of contributions grows, ASATs save increasing amounts of time by providing instantaneous initial feedback on contributions. Furthermore, because this feedback is aimed at the con-

[3]http://maven.apache.org/

[4]https://maven.apache.org/plugins/maven-checkstyle-plugin

tributor as well, the quality of contributions may also benefit. The second research question is therefore:
RQ2: How does community activity affect ASAT usage?

- **RQ2.1:** Is there a relation between ASAT usage and the amount of pull requests?
- **RQ2.2:** Are projects with many different contributors more likely to use ASATs?
- **RQ2.3:** Are contributions more likely to be accepted in projects using ASATs?
- **RQ2.4:** Are pull requests closed faster in projects that use ASATs?

Finally, we want to figure out what sort of warnings are reported by ASATs for different projects, and how project developers deal with this. For this, we consider warning categories as defined by the GDC.
RQ3: What is the prevalence of rule violations reported by ASATs?

- **RQ3.1:** How does the amount of warnings change over time?
- **RQ3.2:** Is there a relation between the code base size of a project and number of warnings?
- **RQ3.3:** Which kinds of warnings appear most often?
- **RQ3.4:** Are warnings of different categories solved at different rates?

We attempt to answer these questions by analyzing a sufficiently large amount of open source projects on GitHub, using its API. To obtain warnings, we will clone repositories locally and run the appropriate ASATs.

The remainder of this thesis is structured as follows. In chapter 2, we explore existing research on ASATs, CI, and Code Reviews. Chapter 3 details what we have done to conduct our study, and chapter 4 provides technical details on how we implemented a tool to assist in obtaining data. In chapter 5 we present the results of our study and threats to validity. In chapter 6 we give answers to the research questions, draw conclusions, and give pointers for potential future work.

Chapter 2

Related Work

In this chapter, we explore some of the existing research on topics related to this study. We start with Code Reviews, considering that is a topic closely related to ASATs, and seeks to solve the same problems. We then move on to Code Analysis Tools, and present research on both the development of ASATs and their use and benefits. Finally, we focus on Continuous Integration in open source projects and see which aspects of it affect development.

2.1 Code Reviews

Some of the first research regarding code reviews was done by Fagan in 1976[17]. He defined a formal way called *code inspections*, meant to be performed during group meetings. This early form was very specific about the roles different people have and the steps they should take, while inspecting the code on a line by line basis, basically checking off items from a list. In this process, only methods to find errors are specified, but no way of solving problems is mentioned. This process was found to benefit software development teams by Basili and Selby in 1987[6] and Ackerman in 1989[1]. However, Porter et al. later found that the costs were often understated while benefits were overstated, especially for large software[40].

Aurum et al. published a summary of improvements and additions to the inspection process over the years[2], and found several other techniques. In the *active design review*[38], several brief reviews are conducted instead of one large review, having a different reviewer with specific expertise for each review. The *two-person inspection*[11] reduced the inspection team size from four to two, just the author and a reviewer, and was found to improve productivity. With *N-fold inspection*[32], a number of small teams performs reviews independently, based on the idea that different teams will find different faults. This approach was found be costly, but also provides substantial benefits as there was no significant overlap between faults found by different team. *Phased inspection*[30] describes different phases of the reviewing process, each with specific goals. At the end of each phase, corrections are applied. Although Fagan emphasized the importance of meeting for code inspections, Votta raised several points against meetings[48]; only two reviewers can interact at any time, only 30–80% of the rest was listening to the conversation, and reviewer hours were being wasted, while most defects are identified before the meetings start.

Although improvements and variations to Fagan's code inspections have been developed over the years, most of these are defined in a formal way, with specific steps to be taken and checklists to be followed. Nowadays, code reviews are more informal and often assisted by tools. Bacchelli and Bird define these as *Modern Code Reviews*, or MCR[4], and found them to be less about finding defects and more about "increasing team awareness, providing knowledge transfer, and revealing alternative solutions to problems". Through a study of different software projects, McIntosh et al. found that both code review coverage and participation had a positive effect on software quality, reducing the amount of post-release defects[33]. In a study of MCR in open source projects, Beller et al. also looked at the kind of changes after a review[7]. Among other things, they reported that 7-35% of review suggestions did not lead to any changes in the code, and that bug-fixing tasks have fewer changes, while contributions with a higher code churn have more changes. 78-90% of the changes are triggered by review comments.

2.2 Code Analysis Tools

Since the release of Lint[27], analysis tools have become a helpful addition to code reviews. Rather than guaranteeing the absence of any errors, these tools can ascertain that specific flaws, which can be defined by the user, are absent. Using ASATs is a cost-effective way of inspecting code, helps prevent faults that could cause security vulnerabilities, and has proven to be effective in identifying problem modules[50]. Furthermore, the results produced by ASATs can be used to effectively determine the quality of a software component[34]. Other research found that many bug patterns are relatively easy to recognise[24] and that security issues can be avoided with static analysis, and runtime overhead can be greatly reduced[25]. The tools build a model of the system and use that to analyze its data flow, but that is computationally expensive because of the large state space. Therefore, abstractions are introduced[13]. An example of such abstractions is *predicate abstraction*, used in the SLAM toolkit[5]. Although checking with abstractions is faster, a smaller set of problems can be detected than when modeling[15].

Despite their promising features, ASATs are not widely adopted[8]. Some reasons for this are the lack of suggestions for quick fixes, warnings messages that are not informative enough, false positives, and an overload of information[26]. Many of the issues reported are trivial[3], and while many errors may be reported, few of those actually reflect a real defect in the code instead of bad style[22]. In addition, false positives are a common sight[31]. As a result, when ASATs are introduced in large existing systems, thousands of warnings typically pop up. To attempt to solve all of them would be a lot of work, for which there generally is no budget. Steidl[44] introduced a model to find costs and benefits, which helps developers to prioritize violations to remove, while at the same time being transparent to their managers.

The value of ASATs increases when they are tightly integrated into a developer's daily workflow. Tricorder[42] was developed to accomplish this for developers at Google. The tool aims to be scalable, easily adopted and actively used to fix code issues, and allow developers to write and deploy their own static analyses. In addition, there is a continuous feedback loop between developers and analyzer writers to improve analysis.

Although most ASATs can be configured, users generally stay close to the default configuration, and rarely change the configuration of the ASATs they use[8].

Through the use of ASATs with code reviews, around 6–22% of warnings are removed, but this percentage varies between warning categories[37]. However, the overall warning density generally stays the same and many errors stay unsolved[29]. Code smells may also be detected by ASATs, but developers often are not very keen on refactoring these either[39].

2.3 Continuous Integration

Continuous Integration, or CI, is a practice where each developer's work is integrated into the main system frequently. With this integration, the software is automatically built and tests are run. The purpose of CI is to spot integration problems early[18]. It also allows for more frequent deployments. CI is easily integrated into GitHub when done by Travis, a CI service for open source projects. Beller et al.[10] found that a little over 30% of projects on GitHub have used Travis for at least one build. Tests are often run during the build, and these are the main reason for builds to fail. The test failure rate is rather low, which could be because developers test their code locally before committing. This contradicts another study by Beller et al. which found that testing is not a popular activity, at least within the IDE[9].A study performed by Vasilescu et al. pointed out that although many projects on GitHub are configured to use Travis, less than half of them do[46]. They also found that pull requests are more likely to result in a successful build than a direct commit.

There are many benefits to making use of CI for open source projects. Research by Vasilescu et al.[47] found that development teams are significantly more effective at merging pull requests from core members, while pull requests from external contributors are more likely to be merged. In addition, more bugs are discovered. Holck and Jørgensen found that CI helps open source projects FreeBSD and Mozilla to produce quality software by allowing contributors to add to the development version at any time, and at the same time making sure that each contribution does not break the build[23].

2.4 Software Repository Mining

In order to perform our project analysis, we obtain data from multiple repositories that are publicly hosted, a process called repository mining. This is a popular method of obtaining data, due to the sheer amount that is available publicly, and has been applied successfully before. As an example, Zaidman et al. used repository mining to look at whether production and test code co-evolve across versions[49]. Vandecruys et al. described the AntMiner+ classification technique, which effectively predicts software quality by deriving a model from mined repositories[45]. Kagdi et al. used version history to find traceability patterns consisting of source files and other software artifacts. By doing so, they were effective in predicting changes in software repositories, and recognizing patterns of change.

Through mining, rather than using static archives, multiple versions of software artifacts can be studied. Kagdi et al. performed a large scale survey on this practice, exploring different ways MSR was used in studies[28]. Nikora and Munson studied how fault counts in software systems evolve between builds[36], similar to how we intend to look at changes to ASAT warnings. Capiluppi et al. looked at the complexity of systems in terms of number of files and folder tree structure, and how this changes over time[12]. They found that the number of components follows a linear trend with a superimposed ripple, and that average folder and file sizes stabilize over releases. Nagappan et al. built predictor models for post-release faults in software components by using historical defect data from different versions of five Microsoft software systems[35]. These models were found to be able to effectively predict the likelihood of post-release defects for new entities.

Although repository mining seems like an easy way to obtain lots of diverse data for research, it is not without downsides. Kalliamvakou et al. looked at the most common pitfalls when mining repositories, and pointed out that not all data should be assumed to be valid[16]. During our own study, we kept their recommendations in mind.

Chapter 3

Experimental Setup

In this chapter, we describe the work that has been performed as part of the study, using the tool that we developed (described in chapter 4). Each section corresponds to a research goal:

1. Finding the prevalence of ASATs, both in general and in relation to use of build tools and CI
2. Finding the effects of ASAT usage on project community and contributions
3. Gaining insight in the occurrences of different warning categories, both numbers and time to solve

First, we collected metadata of a fair amount of GitHub repositories and analyzed the file trees of each repository to determine which, if any, ASATs and Build Tools are used. After that, we obtained information about the latest pull requests for each repository. Finally, we ran each ASAT on different commits a selected number of repositories to obtain a set of warnings for each commit of each repository. The following sections explain each step in more detail.

3.1 Retrieving Repositories

To get started, we need a sizeable list of repositories from to work with. As of 2016, GitHub hosts over 35 million repositories[1], so we can safely limit ourselves to this repository hosting. A majority of these are personal and inactive[16], and it would be infeasible to work with all repositories. Because we will also be gathering statistics on pull requests, we want active repositories, that have reached a decent level of popularity. To get active repositories, we require the last push to have been performed on or after January 1, 2016. For the popularity, we require the repositories to have been "starred" – a way for other GitHub users to save a repository to their list of favorites – at least 200 times. In addition, due to the ASATs that this study focuses on, we will only retrieve repositories with Java, Ruby,

[1]`https://github.com/features`

Python, or JavaScript as their main language. Finally, the GitHub Search API only returns public repositories that aren't forks of other repositories by default.

Upon applying these constraints, we ended up with 9443 repositories. For these repositories, we retrieved their repository id, full name, default branch, star count, whether or not they use GitHub issues, the number of open issues, date of creation, date of the last push, and main language.

3.2 Basic Repository Properties

Now that we have a list of repositories to work with, we are ready to retrieve some basic information on a per-repository basis. We look for three properties of the repository's project here.

3.2.1 Build Tools

For a select number of different build tools (see table 3.1), we check if the project uses them. These build tools have been selected because they are most commonly used for their respective language, and Travis CI[2] has default scripts built in for them. The exception is Python, which does not really have a standardized build tool. Tox and Make are mostly used in projects to run automated tests and other build tasks, though not in the majority of the studied projects. With this information, we can then see if there is a correlation between the use of ASATs and the use of build tools. We assume that this is the case, because it is easier to enforce the use of an ASAT if it is contained within a build task; the task can cause the build to fail if there are ASAT warnings.

Language	Build Tools
Java	Gradle, Maven, and Ant
Ruby	Rake
Python	Tox and Make
JavaScript	Grunt, Gulp, and Make

Table 3.1: Build Tools checked per language

3.2.2 Analysis Tools

With them being the main topic of the study, we are mainly interested if ASATs are used, and if so, which ones. This information can later be used to find out which ASATs are most popular, and the prevalence of ASATs per programming language. We define three different signs of an ASAT being used:

1. A specific configuration for the ASAT is stored in the repository, either as separate file or as part of build tool configuration

[2]`http://www.travis-ci.org`

2. The ASAT is included in at least one of the tasks of the build tool that the project uses
3. The ASAT is added as a (development) dependency in the project's dependency manager's configuration

Although #2 and #3 should be tied together most of the times (it does not make sense to have an ASAT in a task without having a dependency to it), some ASATs work with a default configuration in the absence of a specified configuration file. Therefore we check for all 3 of these, and decide that an ASAT is used if any of them is true. Although we still cannot say that the ASAT is actively being used with certainty, we know that at some point, a maintainer consciously added it to the project.

3.2.3 Continuous Integration

About 30% of active, meaningful open source projects on GitHub use Travis for CI[10]. Since ASATs can easily be included in CI builds (either as part of the build tool configuration, or added to the Travis script directly), we want to check if a project is currently using Travis. This way we can see if Travis is often used along with ASATs or build tools, and how its use affects the amount of warnings in commits.

3.3 Analyzing Pull Requests

In addition to information about the project itself, we are also interested in the activity of the project's community. For this we analyze the pull requests that have been submitted and closed on GitHub since the repository was created. We also retrieve the number of closed pull requests for each repository since its creation. Using this number, we can later determine whether there is a correlation between using ASATs and the amount of pull requests that a repository receives.

Then, for some more detailed data, we retrieve information about the last 100 closed pull requests for each repository. The reason for this number is that the GitHub API can return up to 100 results per request, which we believe should be a sufficient amount. The obtained data includes the id of the user that submitted the pull request, the moment the pull request was created and the moment it was closed (i.e. merged or rejected). We only obtain data for closed pull requests because we want to analyze their lifetime, which is undefined for open pull requests.

3.4 Analyzing Warnings

For the main part of the study, we want to obtain the ASAT reports for repositories. That way, we can find out which types of warnings occur most frequently, and the distribution of the total number of warnings in a repository. However, we also want to go a step further, by obtaining this report for many different commits of the repository. This way, we can observe the changes in the amount of warnings, but also the rate at which individual warnings are solved.

Unlike the previous steps, we cannot obtain ASAT reports through the GitHub API. An initial idea was to parse build logs from Travis. However, this idea was quickly dismissed due to its complexity; reports in build logs can be formatted in many different ways, depending on whether the ASAT is run directly or through a build tool plugin. Furthermore, not all projects run ASATs as part of the Travis build, and this method would require scanning large amounts of data, which would be infeasible. Instead, we decided to locally clone each project we want to study, then run the ASAT ourselves. This way, we have complete control over the output format, making it more suitable for parsing.

Unfortunately, we could not fully automate the process of code analysis on all repositories. There are several reasons for this: the directories where source code is stored are not consistent between repositories, and configuration files may contain settings that have been removed in a newer version of an ASAT, or settings that only recently have been added. In addition, for Checkstyle and PMD, the configuration file can be stored anywhere in the repository, so it should be provided manually. In some cases, this information could be retrieved from the build tool configuration, however, we did not want to limit ourselves to just the repositories where this could be automated.

To decide if a project is suitable for analysis, we have some requirements. First, a specific configuration file should be present in the repository, so that we can be sure the project developers have consciously integrated the ASAT. Besides that, some ASATs require a custom configuration to be provided and yield an error message when used without any, while others can be used without configuration. However, the default behavior is not consistent amongst different ASATs; some have default rules they will use, others will not use any rules. To be safe, a configuration file needs to be present at all times. Another requirement is that the number of closed pull requests should at least be 100, so that only projects which are actively contributed to (with response from the maintainers) are left.

3.4.1 Collecting Results

We configure the ASAT in question to yield output parseable by our tool, in XML or JSON format. Different ASATs report warnings in different ways, so we normalize the results. This way we end up with the file where the warning occurred, the line and column numbers, the warning message, and the identifier of the rule that was violated, unique per ASAT.

Basically, the results could be saved like this, but with 7 different ASATs, there are a lot of different rules. This would mean that the total amount of violations per rule would be rather low. To deal with this, we use the General Defect Classification, defined by Beller et al[8]. This classification defines a relatively low number of categories for ASAT rules, with mappings included for the ASATs of this study. By mapping the rules to these categories, we can compare warnings across ASATs.

For each warning, we also keep a reference to the commit (and its timestamp) of the repository that caused said warning. That way, we can later figure out how long it took to fix a warning.

Chapter 4

Tool Implementation

The retrieval of the data, and transforming this data into usable results, has mostly been performed by a self-developed tool. This tool, along with exported results, can be found on GitHub[1]. In this chapter, we first describe the implementation of the tool itself, and then focus on the database design.

4.1 Repository Data Retrieval

For this study, we are going to be mining software repositories on GitHub. Since this will be quantitative research, we want to obtain data on as many repositories as possible. This is most easily accomplished using the public API[2]. Most of the steps described in chapter 3 are easily automated, so having a program retrieve all necessary data is the best solution. This program needs to be able to connect to GitHub's API, store data, and export usable results. A command-line tool would suffice, since only basic input will be required, such as the specific task to be performed. This task can be one of retrieving or analyzing repositories, retrieving pull request data, or running ASATs while also using Git to switch between commits.

We have implemented this tool in over 2000 lines of PHP code. The reason to develop in PHP was made mainly because of personal experience with the language and its ecosystem. Interaction is done through a Command-Line Interface, for which the Symfony Console Component[3] was used. Furthermore, the Guzzle HTTP Client[4] was used to interact with GitHub's API, which is accessed over HTTPS. For database interaction, we used the Illuminate Database Component[5], which is part of the Laravel[6] PHP framework.

Figure 4.1 shows the general process for most interactions. Once a command is issued, it passes a search query to the `GithubClient` instance, a singleton which contains all the logic for interacting with the GitHub API using Guzzle. This object translates the query

[1]https://github.com/bvangraafeiland/RepositoryAnalyzer

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

[3]http://symfony.com/doc/current/components/console/introduction.html

[4]http://docs.guzzlephp.org/en/latest/

[5]https://github.com/illuminate/database

[6]https://laravel.com

into an HTTP request, makes that request, parses the response and returns it to the calling command. The command can then store the results into the local database.

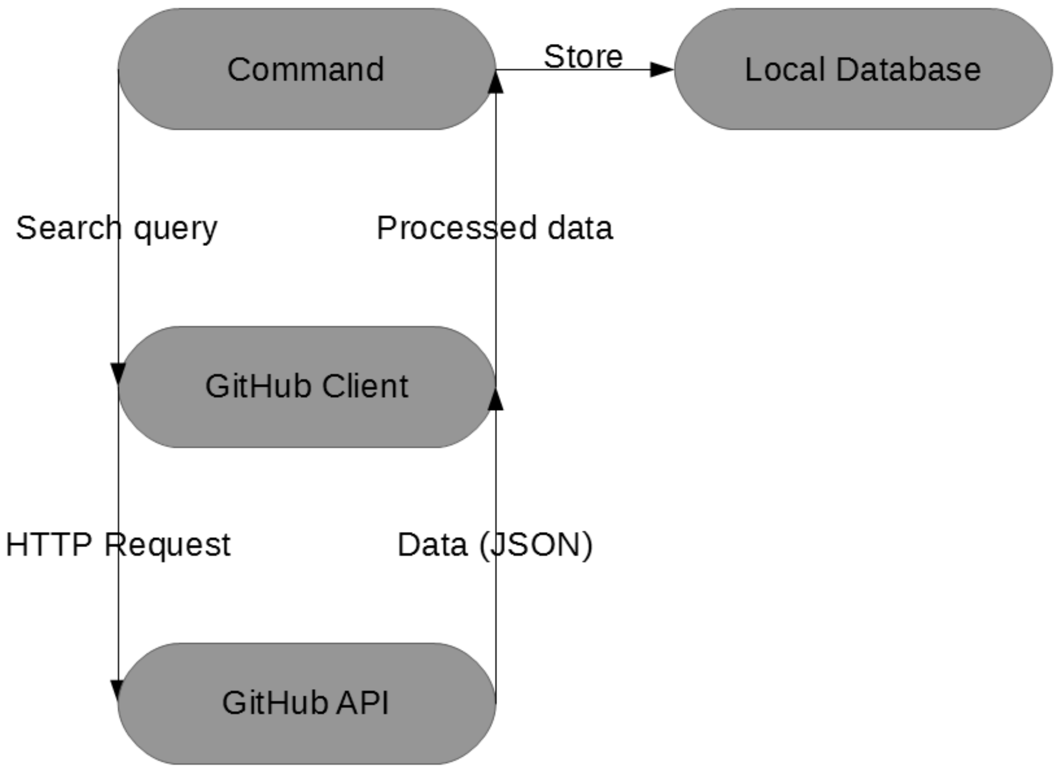


Figure 4.1: Flowchart for obtaining data from GitHub

4.1.1 Obtaining Repositories

The first step is to retrieve a large number of repositories to work with. Our goal is to obtain as many repositories as possible, but not include inactive or personal repositories. To cover that, we limit ourselves to repositories with at least 200 stargazers that have been pushed to since January 1, 2016. Besides that, we only retrieve repositories with Java, JavaScript, Ruby or Python as their main language, since we only study ASATs for these languages. These parameters are then specified in a search query. Through the GitHub API, we obtain the name, default branch, stargazers count, creation date, last push date and language of repositories that meet the requirements.

To retrieve repositories, we make use of the `/search/repositories` API endpoint. However, a single request returns only up to 1000 results, as this endpoint is meant to be used for searching for specific repositories. Because we instead want to obtain all repositories that meet the requirements, we limit each search to a single year. To get all repositories, we perform a search for each year from 2008 (creation year of the oldest repository on GitHub) to 2016 per language. This way, each result set ends up being smaller than 1000. This is depicted in figure 4.2.

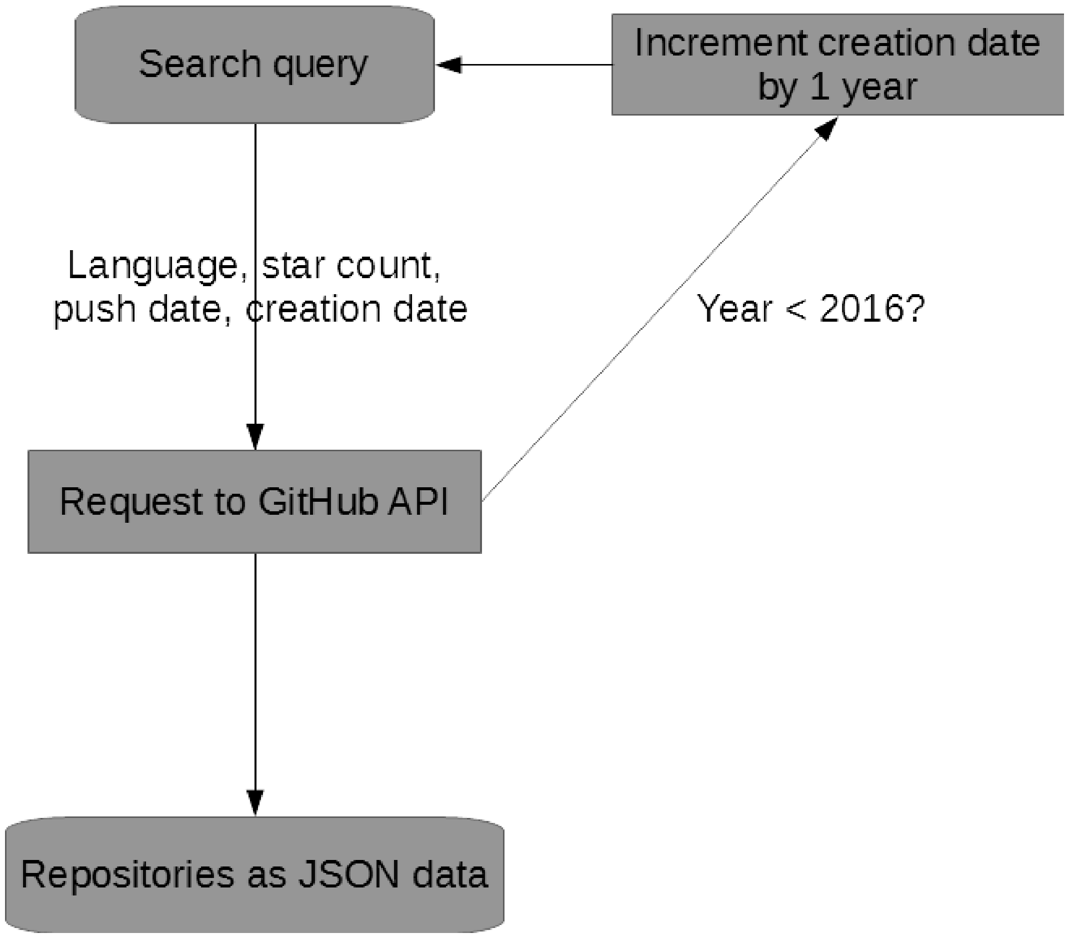

Search query
Increment creation date by 1 year
Language, star count, push date, creation date
Year < 2016?
Request to GitHub API
Repositories as JSON data

Figure 4.2: Obtaining all desired repositories from GitHub

4.1.2 Analyzing Project Properties

Now that we have the base dataset with repository names, we are ready to obtain more detailed information about each repository. We analyze which ASATs are used (if any), which build tool is used (if any) and whether the project makes use of Travis. For each repository, we first obtain the names of all files in the root directory. This information is used as part of the build tool and ASAT checks. From the repository contents, we also retrieve the dependencies file and build tool configuration, if those exist. This process is depicted in figure 4.3.

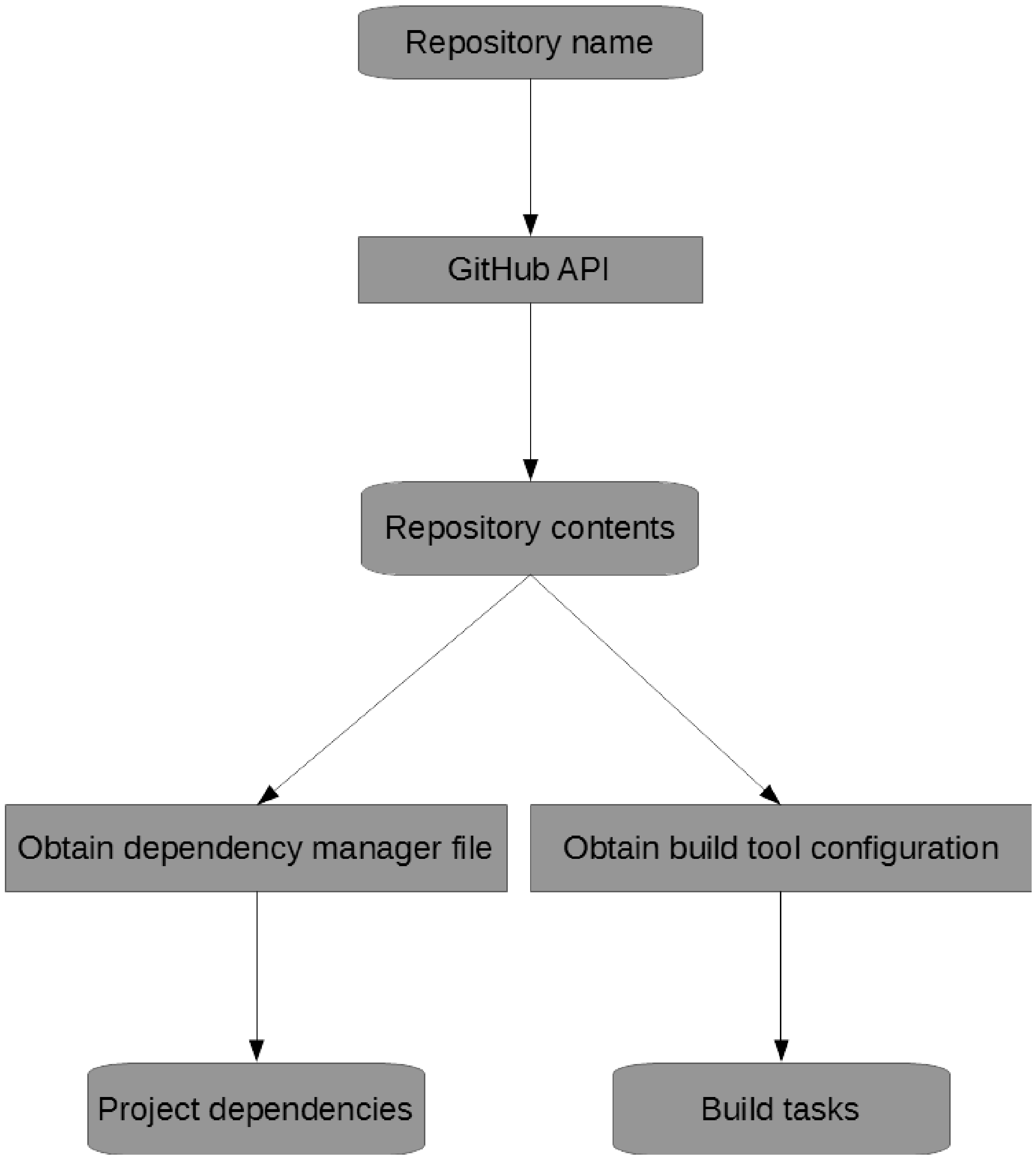


Figure 4.3: Retrieving information about the project's dependencies and build tasks

When the repository contents, project dependencies and build tool configuration have been retrieved, we can look for evidence of ASAT, build tool, and Travis usage. The general idea of this process is displayed in figure 4.4 and described in more detail in the following paragraphs.

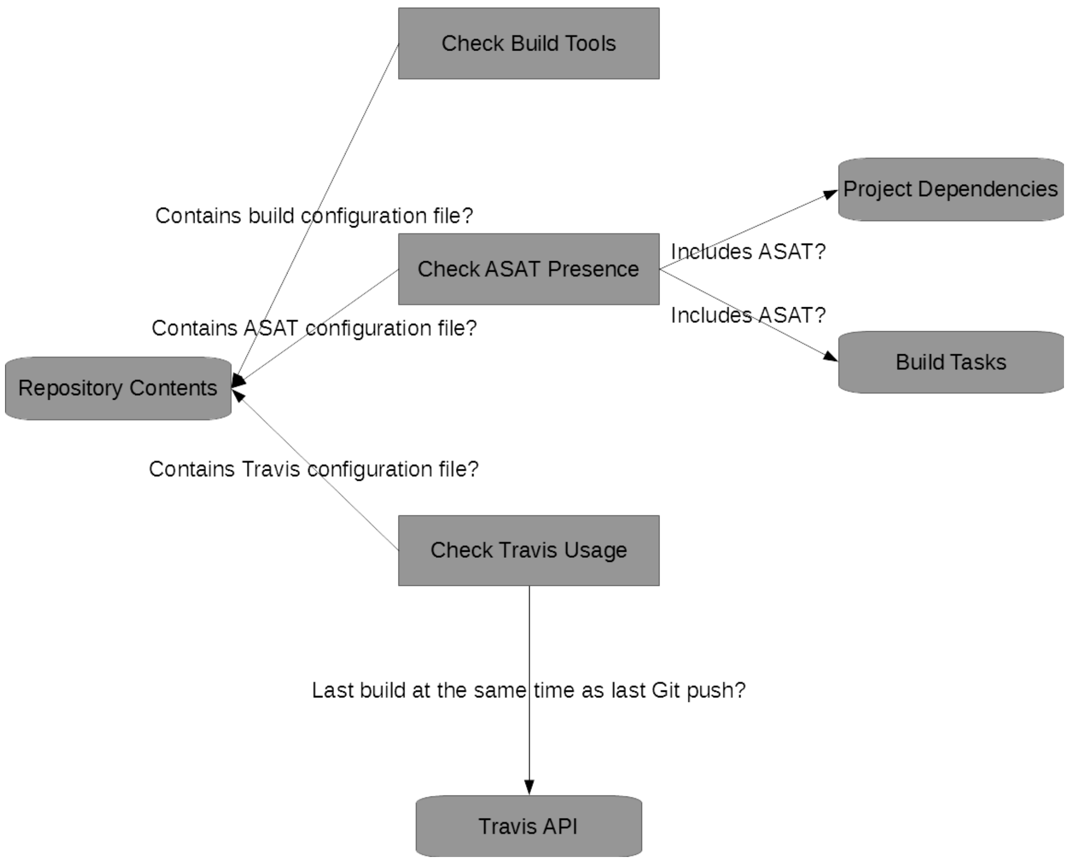

Check Build Tools
Project Dependencies
Contains build configuration file?
Check ASAT Presence
Includes ASAT?
Includes ASAT?
Contains ASAT configuration file?
Build Tasks
Repository Contents
Contains Travis configuration file?
Check Travis Usage
Last build at the same time as last Git push?
Travis API

Figure 4.4: Deciding ASAT, build tool, and Travis usage

Build Tool Usage

To build tool usage, depending on the language of the repository, we check for the existence of a list of files, defined by the language-specific subclass, in the root directory contents, as shown in the top rectangle of figure 4.4. This means `build.gradle` for Gradle, `pom.xml` for Maven, `build.xml` for Ant, `Rakefile` for Rake, `tox.ini` for Tox, `Makefile` for Make, `Gruntfile.js` for Grunt, and `gulpfile.js` for Gulp. We just check for the files' existence, so in case other build tools should be checked, our repository analysis tool can easily be expanded to account for that.

ASAT Usage

After that, language-specific checks are performed, which are also implemented by the respective subclass. Three checks are performed:

- **Looking for an ASAT in the project's dependencies**. Projects often use some sort of dependency management, like Maven for Java or Bundler for Ruby. To determine an ASAT being listed as a dependency, we search the dependencies file for a mention (e.g. `Gemfile` for Ruby).

- **Looking for an ASAT in the project's build tasks**. Within the build tasks file, we search for a plugin of the ASAT under study (e.g. `gulp-eslint` in `gulpfile.js` for ESLint in JavaScript projects).

- **Looking for an ASAT configuration file in the repository**. All ASATs, except PMD and Checkstyle, look for a specific configuration file in the root directory by default. We check for this file's existence in the previously obtained root directory contents (e.g. `.pylintrc` for Pylint).

These steps are illustrated in the middle rectangle of figure 4.4 and the following paragraphs explain each part in more detail.

ASAT Configuration

The way configuration files are used differs per ASAT. JSHint, JSCS, and RuboCop all have a single file that they use by default (`.jshintrc, .jscsrc, .rubocop.yml` respectively). ESLint accepts different file types for configuration: `.eslintrc.js, .eslintrc.yaml, .eslintrc.yml, .eslintrc.json` for JavaScript, YAML or JSON (alternatively, a plain `.eslintrc` file can be used containing either YAML or JSON, which is deprecated but still valid). Pylint accepts either `.pylintrc` or `pylintrc`. All mentioned files are assumed to be stored in the repository's root directory, although a different location can be specified. Checkstyle and PMD use XML files for their configuration, which can be stored anywhere, and then referenced in the task of the build tool or on the command line. In addition to a separate configuration file, JSHint, JSCS, and ESLint configurations can also be stored in the `package.json` file[7] under the `jshintConfig, jscsConfig`, or `eslintConfig` fields.

To determine that a custom configuration is used, we check for the existence of the mentioned file names in root directory of the repository. Although it is possible that configuration is stored elsewhere, we do not check for that because it would require searching through each single repository, which is time consuming due to GitHub's API rate limit (30 searches per minute). In addition, it seems that very few repositories actually store their configuration in a directory other than the root. In case of the JavaScript ASATs, we also read the `package.json` file to see if configuration is specified there. As for the Java ASATs, we parse the build tool's configuration file (Gradle, Ant or Maven) and check if a path to a Checkstyle/PMD configuration file is mentioned there. Unfortunately, this means that Java projects need to use either Gradle, Maven or Ant, but most projects seem to be using one of these, and it can be argued that without a build tool, a project is not very likely to use any ASATs.

[7] In the `package.json` file, npm packages can store basic documentation, dependencies on other packages, and build scripts.

ASAT in Build Task

By themselves, ASATs can easily be overlooked. Just having a configuration in the repository hardly guarantees that a tool will ever be used by collaborators. IDEs may be configured to pick up ASAT configurations and point out warnings, but there is no control over which IDE collaborators use, if any. This would also mean mean IDE configuration for multiple IDEs needs to be stored in the repository, which seems undesirable. To make sure that code analysis will be included in the regular workflow, it would be wise to incorporate an ASAT in the build process instead. For most build tools (Python excepted), plugins are readily available to integrate an ASAT in the build process, usually requiring only a couple extra lines of configuration.

Deciding whether an ASAT is in a build task is different for each language. Again, we consider the build tools of table 3.1. For this reason, we first checked for build tools; we then already know which build tool configuration files are present for parsing. Evidence of ASATs in the build is specified in table 4.1. Based on these observations, we decide which ASATs are included in build tasks.

Build Tool	Evidence
Gradle	In `build.gradle`, the respective plugin for PMD or Checkstyle is included.
Maven	An `execution` element has been added in the Checkstyle/PMD plugin, with the `check` goal to be executed.
Ant	A Checkstyle/PMD element is present in on of the targets defined in `build.xml`.
Rake	A Rake task has been created to run RuboCop, meaning `Rakefile` contains the text `RuboCop::RakeTask.new` (RuboCop comes with a Rake task to use).
Tox	The text `pylint` exists in `tox.ini`.
Make	The name of the ASAT is mentioned in `Makefile`.
Grunt	The name of the plugin for the ASAT is mentioned in `Gruntfile.js`.
Gulp	The name of the plugin for the ASAT is mentioned in `gulpfile.js`.

Table 4.1: Signs of an ASAT being included in the build for each build tool

ASAT as Dependency

Finally, we check whether a project lists an ASAT, or a build tool plugin of an ASAT, as a dependency in its language's dependency management. We consider dependency management as follows: Gradle or Maven for Java, Bundler for Ruby, Pip or `setup.py` for

Python, and NPM for JavaScript. In most cases, a dependency should be defined if there is a build tool task, otherwise a project would have to depend on other developers having tools already installed on their machine. Nevertheless, we check for dependencies because the ASATs could be run directly without a build tool as middle man. For example, `beautify-web/js-beautify` contains a `build.sh` script which runs jshint with

```
$PROJECT_DIR/node_modules/.bin/jshint 'js' 'test' || exit 1
```

For Java, we consider a dependency to be the same as having a build task, so the check is essentially the same as in the previous step. This is because adding the Checkstyle/PMD plugin to Gradle/Maven automatically causes the build tool to download the ASAT when running the build. For Ruby, we look for a `rubocop` mention in `Gemfile` or the `.gemspec` file (in case the project itself is a Gem), which Bundler uses to download dependencies. For Python, we search the `requirements.txt` and `setup.py` files to see if they include `pylint`. Finally, for JavaScript, we parse the `package.json` file, combine the contents of the `dependencies`, `devDependencies`, and `optionalDependencies` fields, and search for the name of the ASAT (`jscs, jshint, eslint`) in there.

Travis Usage

Evidence for Travis usage comes in two parts, as shown in the bottom rectangle of figure 4.4. First, we make sure the configuration file `.travis.yml` exists in the repository root. Second, we make an HTTP request to the Travis API, obtaining information about the last build of a project. If the last build exists and it was performed at the same time of the last push, we consider the project to be actively using Travis.

4.1.3 Retrieving Pull Requests

All required data regarding the community around projects can be obtained by looking at the properties of pull requests. As mentioned in chapter 3, we restrict ourselves to the last 100 closed pull requests, because the GitHub API returns that many per request, and we are analyzing the time that pull requests stay open, which is undefined for those that are still open.

Again, the GitHub API provides us with all the data we need. For each pull request, we retrieve the title, number, ID of the user who submitted the pull request, and timestamps of when the pull request was opened, merged (`null` if the request was rejected), and closed (identical to merged if the request was accepted). Each pull request is also associated with its repository internally. We also obtain the total amount of pull requests that a repository received by fetching the list of pull requests from GitHub and setting the number of results per page to 1. By checking the total number of pages (which the response includes in its headers), we know the total number of pull requests. With this basic data, we can compute some more properties of repositories:

- **Time to close pull requests.** For each of the repository's pull requests, we calculate the difference in seconds between the `created_at` and `closed_at` properties. The repository's time to close is the average of the obtained values.

- **Pull request density.** This property comes in two forms: lifetime density and recent density. The former is calculated as the total number of pull requests divided by the repository age in hours. The repository age is calculated by taking the difference between the `created_at` and `pushed_at` attributes of the repository. The recent density is calculated by taking the difference in hours between creation dates of the first and last of the (up to) 100 pull requests retrieved for the repository. This difference is then divided by the amount of pull requests we obtained for the repository.

- **Unique user count**, defined as the number of different user ID's that appear in the repository's pull requests.

- **Merged count**, the number of pull requests where `merged_at` is set, i.e. the pull request was accepted.

4.1.4 Running ASATs

Preparation

We start by figuring out which directories of the repository contain the source code by manual inspection and determine which version of the ASAT should be used, sometimes with the help of build tool configuration. We then install the required ASAT version if needed. Sometimes, other information is required, such as additional command line options. In the cases of Checkstyle and PMD, we also look up the configuration file location. All of this is then saved to a configuration file for our tool.

Code Analysis

With the configuration set, we are ready to run the ASAT. Using Git, we first obtain a list of commit hashes representing the versions of the repository to run the ASAT on. For each project, we retrieve the last 500 versions, while using the `--first-parent` option in order to exclude duplicate commits caused by pull requests. Then, using the command line interface, we run the ASAT over the directories specified in the repository's configuration.

When running the ASAT, we make it output either JSON or XML data in order to make parsing easier. Most ASATs provide a formatter for this out of the box, but for ESLint, JSHint and JSCS we implemented our own formatter due to the ease of doing so and having the output exactly in the way that suits our tool best. For each warning encountered, we retrieve the name of the file where the warning occurred, the line and column numbers, the warning message, and the identifier of the rule that was violated. We also categorize each warning according to the GDC, using preset mappings, and using `sed`, we obtain the content of the line of code that the warning occurred on.

The ASAT is run on each commit, until either 500 versions have been run, or the ASAT configuration file is no longer present (meaning it has been added to the repository within the last 500 commits). The full process is displayed in figure 4.5.

4.1.5 Exporting Data

At this point, we have most of the data we need. The analyses done on the data are performed by means of an R script, which works best with input files of CSV format. Therefore, the data stored by the tool is exported to CSV files. The results we export are:

- **Statistics per language** – Repositories are aggregated by language to count the amount that use asats, build tools, and Travis.
- **Basic repository properties** – The name, star count, language, pull request count, age, pull request density, and use of ASATs and Travis for each repository. This can directly be retrieved from the data.
- **Pull request properties** – The information about pull requests for each repository: total count, merged count, time to close, density, and unique user count. Also includes whether the repository uses ASATs, so that the other properties can be compared between projects using ASATs and those that do not.
- **Warning counts** – For each repository, a separate file is exported that lists the commits of that repository, along with the total number of ASAT warnings in that commit.
- **Statistics for analyzed repositories** – For each repository that we ran ASATs on, we export the average warning count per commit, the total lines of code (using the command-line tool `cloc`[8]), and the normalized warning count, expressed as warnings per 100 lines of code.

In addition, we compute solve times for each GDC category. The way this is done is explained below.

Solve Times

After having run the ASATs on a number of different commits, we end up with a set of warnings across all of these commits. By comparing warnings between commits, we obtain the dataset of solve times. The way these are computed is depicted in figure 4.6 and explained in more detail below.

First, we remember the set of warnings belonging to the first commit. For these warnings, we cannot say anything about their solve times, because we cannot tell how long the warnings had been present prior to that first commit. Then, for each commit, we increase a counter for each warning that is present in the commit and not present in the initial set. Furthermore, we save the counter for each warning that has a counter value, but is not in the set of warnings of the current commit, and remove that warning's counter. In other words, we consider that warning solved, so the number of commits that it has existed can be saved as its solve time. Eventually, after this has completed for all commits, we end up with a set of solve times. These solve times are then grouped per GDC category. After completing these steps for all repositories, we merge the solve times per category of each repository to obtain the final dataset to export.

[8]http://cloc.sourceforge.net

4.2 Database Design

We used a MySQL database to store retrieved data from both GitHub and ASAT output. The main reason is again personal experience, but also the ease of integrating with PHP. The database contains the following tables for the base entities:

- **`repositories`** – Basic repository data as mentioned in the previous section.
- **`analysis_tools`** – Names of the ASATs under study.
- **`build_tools`** – Names of the build tools under study.
- **`pull_requests`** – Pull request data, including a `repository_id` column.
- **`results`** – The hash and commit date for the results produced by an ASAT of a single commit. Includes the `repository_id` column.
- **`warning_classifications`** – Names of the categories defined by the GDC.
- **`warnings`** – Single warnings produced by ASATs. Includes the `classification_id`, `results_id`, and `analysis_tool_id` columns.

In addition to the base entities, the schema contains three pivot tables:

- **`analysis_tool_repository`** – Links repositories to ASATs. Also includes the method of ASAT usage in the form of `config_file_present`, `in_dev_dependencies`, and `in_build_tool`.
- **`build_tool_repository`** – Links repositories to build tools. The reason for this being a pivot table rather than a column in the `repositories` table is that a repository may include multiple build tools.
- **`analysis_tool_result`** – Links ASATs to results. This being a pivot table instead of having a column in the `results` table makes it possible to store results for more than a single ASAT per repository.

The relations between the entities are displayed in figure 4.7.

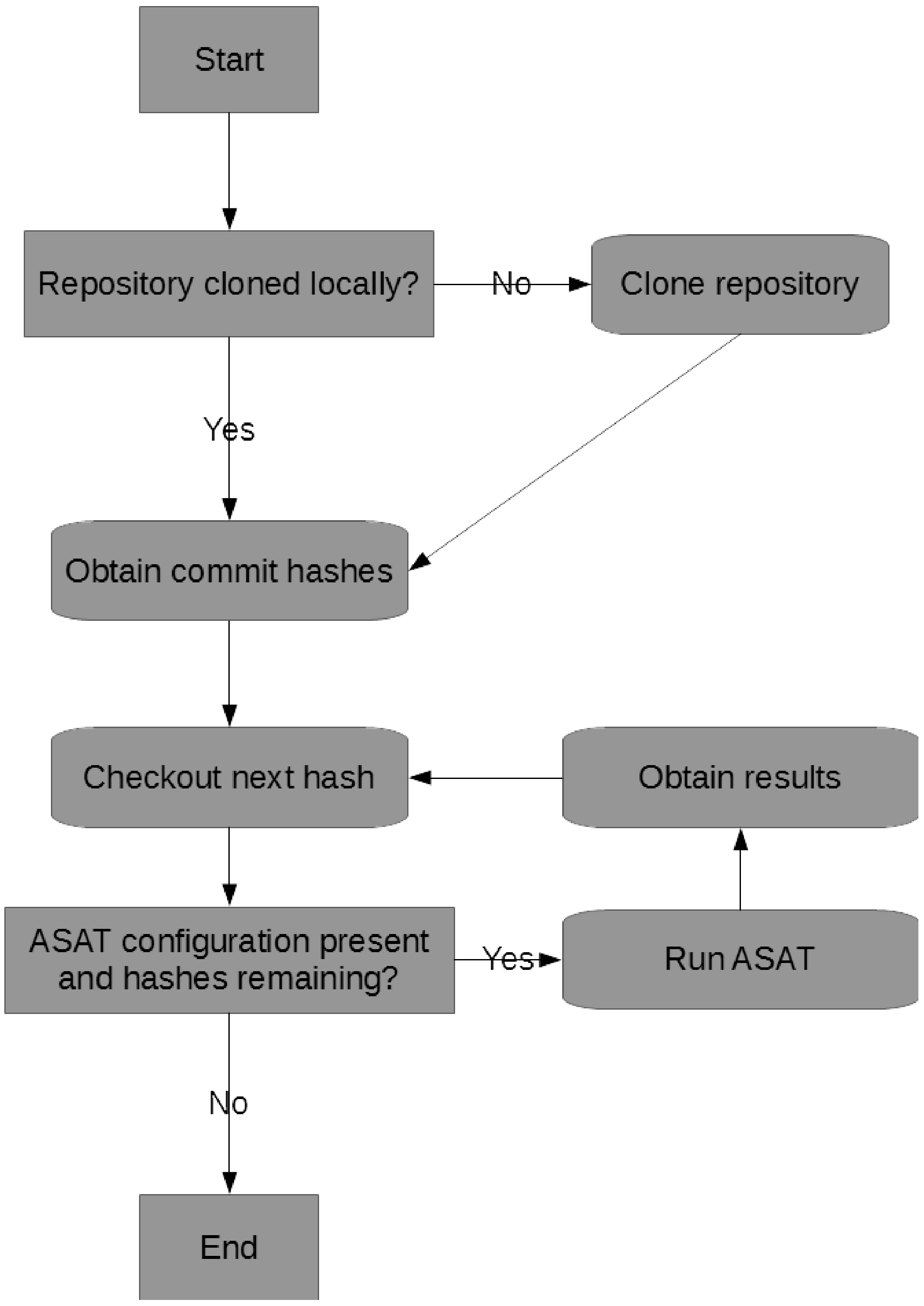


Figure 4.5: The process of running an ASAT on each version of a project

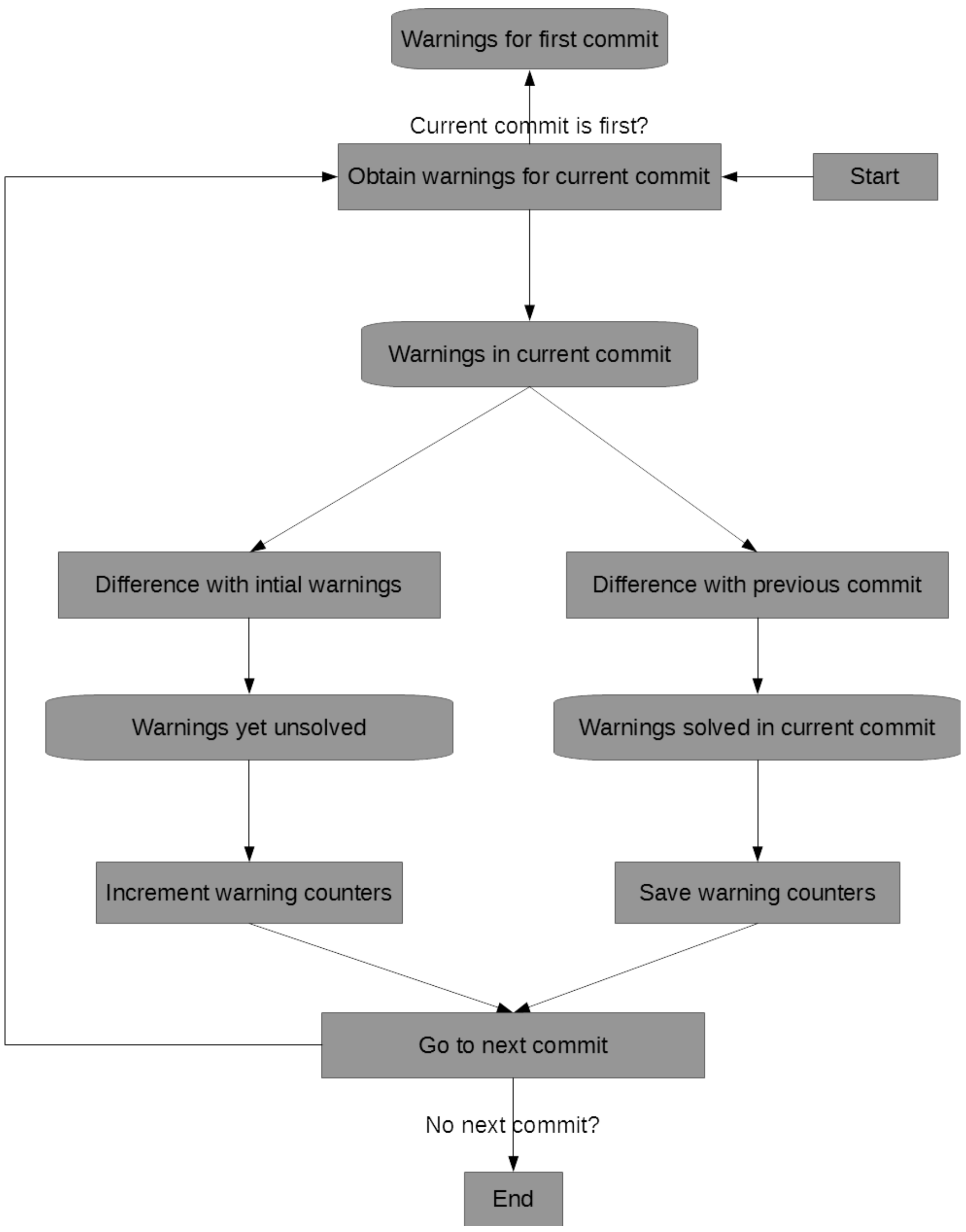

Warnings for first commit
Current commit is first?
Obtain warnings for current commit
Start
Warnings in current commit
Difference with intial warnings
Difference with previous commit
Warnings yet unsolved
Warnings solved in current commit
Increment warning counters
Save warning counters
Go to next commit
No next commit?
End

Figure 4.6: Algorithm for computing solve times

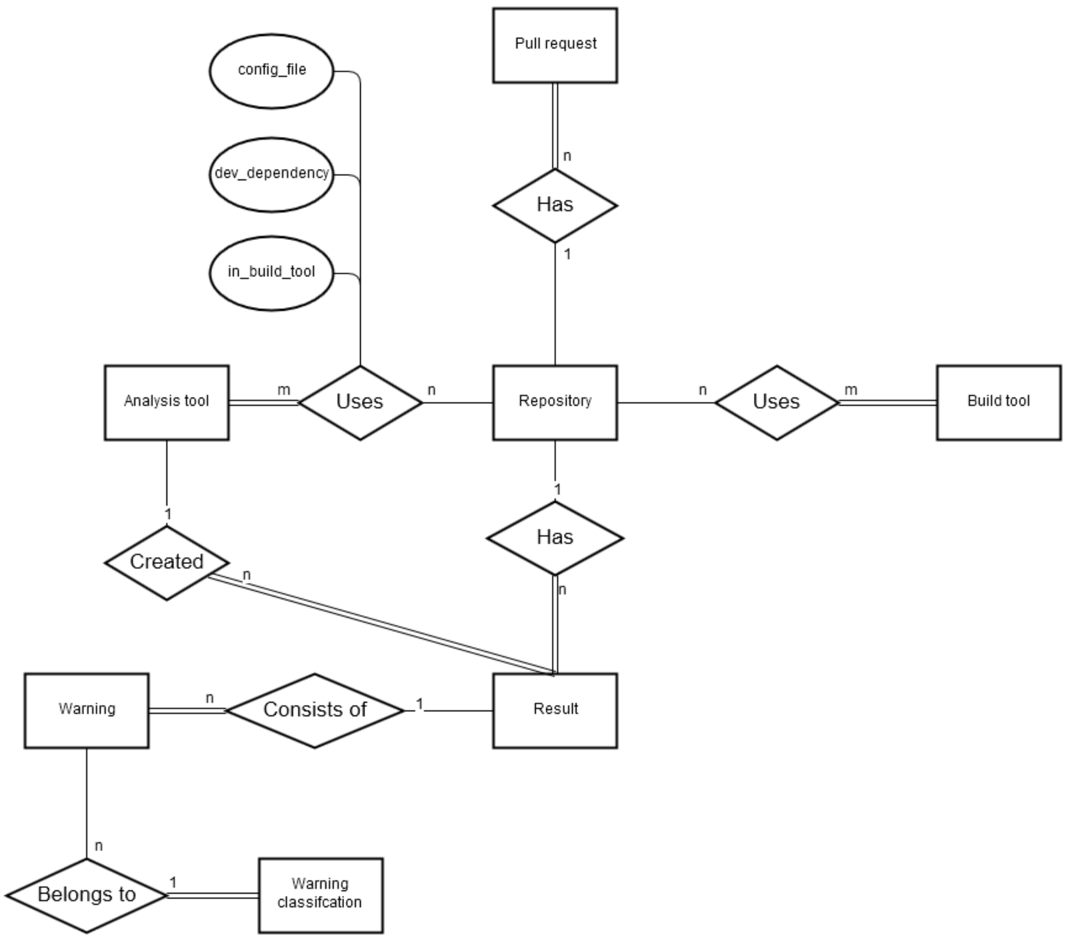


Figure 4.7: Entity-relationship diagram of the database

Chapter 5

Results

In this chapter, we present the results obtained in the study. First, we provide basic information about the data that we have collected, present insight into ASAT prevalence, and look for correlations between ASAT usage and several other repository properties. In the second section we compare community activity between projects that use ASATs and projects that do not. The final section contains an in-depth analysis of warning counts and the time it takes to solve warnings of different categories as specified by the General Defect Classification.

Unless mentioned otherwise, the statistical test we use in this chapter is the two-sample Wilcoxon test, also known as the Mann-Whitney U test, reporting a two-tailed *p*-value with the default threshold of 0.05. The reason for this is that in the first place we want to find out whether the samples have different means. Furthermore, we cannot assume normality, as determined by means of the Shapiro-Wilk test, which yielded *p*-values below 0.01 for all of the datasets. In particular, the star counts (figure 5.3) and pull request counts (figure 5.4) seem to more closely resemble a Pareto distribution.

5.1 Basic Statistics

In total, we ended up with 9443 repositories that met the language, activity, and star requirements as described in chapter 3. Of these, 6772 do not use ASATs, while 2671 do. To get an idea of the prevalence of ASATs per language, table 5.1 shows ASAT statistics about repositories grouped per language. The last column shows the percentages of projects that use ASATs as build task relative to all projects that use build tools. JavaScript projects turn out to be likely to use an ASAT as build task when using a build tool, but this does not hold for other languages.

5.1.1 ASAT Prevalence

We are also interested in the popularity of each ASAT. Figure 5.1 displays this as the number of projects that has included an ASAT in some way. Here, the total count may be larger than the total number of repositories, because a project can be included in the counts of more than one ASAT.

Language	# repositories	Uses build tool	Uses ASAT	ASAT as build task
JavaScript	4504	1343 (30%)	2154 (48%)	830 (62%)
Java	1823	1589 (87%)	160 (9%)	122 (8%)
Ruby	1316	1190 (90%)	247 (19%)	126 (11%)
Python	1800	768 (43%)	117 (7%)	54 (7%)

Table 5.1: Basic repository statistics, with percentages of the total number

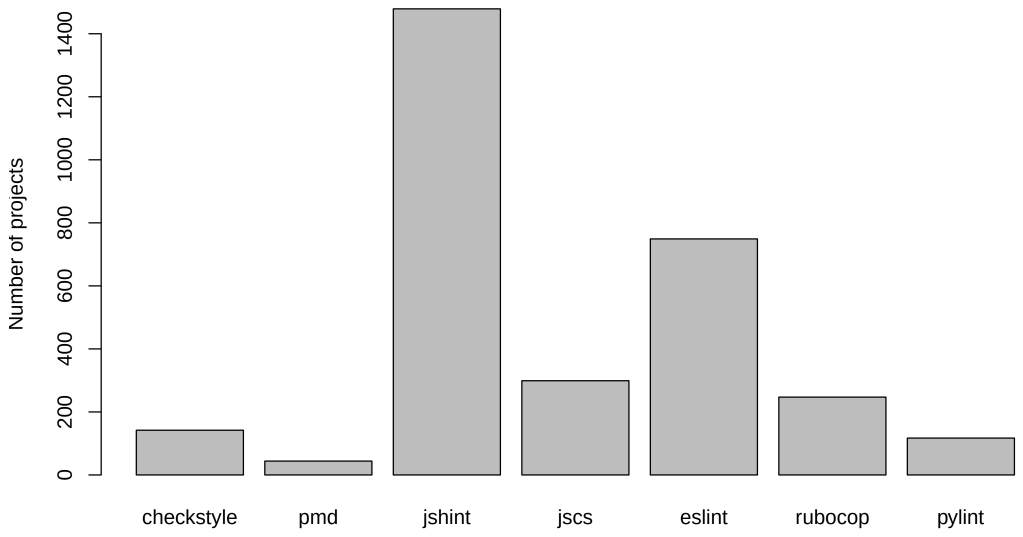


Figure 5.1: Number of projects for each separate ASAT

From this we can see that the ASATs for JavaScript (JSHint, ESLint and JSCS) are most popular. This could be attributed to JavaScript being the most frequent language along with having the highest ASAT use.

Because we studied more than a single ASAT for Java and JavaScript, we can also say something about the number of ASATs used in repositories of those languages. 26 of the 160 Java projects use both Checkstyle and PMD, whereas 353 out of 2154 JavaScript projects use more than one ASAT, of which 20 projects use all three of them.

5.1.2 ASATs with Travis

We now look at ASAT usage within Travis-enabled repositories, to find out whether Travis usage has any effect. These are presented in table 5.2. Compared to the statistics of all repositories in table 5.1, the percentage of projects using ASATs is higher for all languages. The biggest difference can be observed for JavaScript (48% to 64%), which already had the highest use rate. The reason for the higher use rate could be because ASATs offer more value when integrated into Travis (automatic feedback for use in code reviews), but another explanation is that both Travis and ASATs are workflow-enhancing tools and once project maintainers decided to use such tools, they are likely to integrate both.

Language	# repositories using Travis	# using ASATs (% of total)
Java	698	102 (15%)
JavaScript	2084	1337 (64%)
Python	966	81 (8%)
Ruby	924	204 (22%)

Table 5.2: ASAT usage for repositories using Travis

5.1.3 ASAT Usage and Repository Age

To find out whether age of a repository influences ASAT usage, figure 5.2 shows box plots of age in hours for both groups. From this figure, it would seem that newer projects are more likely to use ASATs. Indeed, the average age is 25757 hours without ASATs and 24278 with ASATs. With a p-value of 0.0022 ($U = 9409700$), the groups have a statistically significant difference.

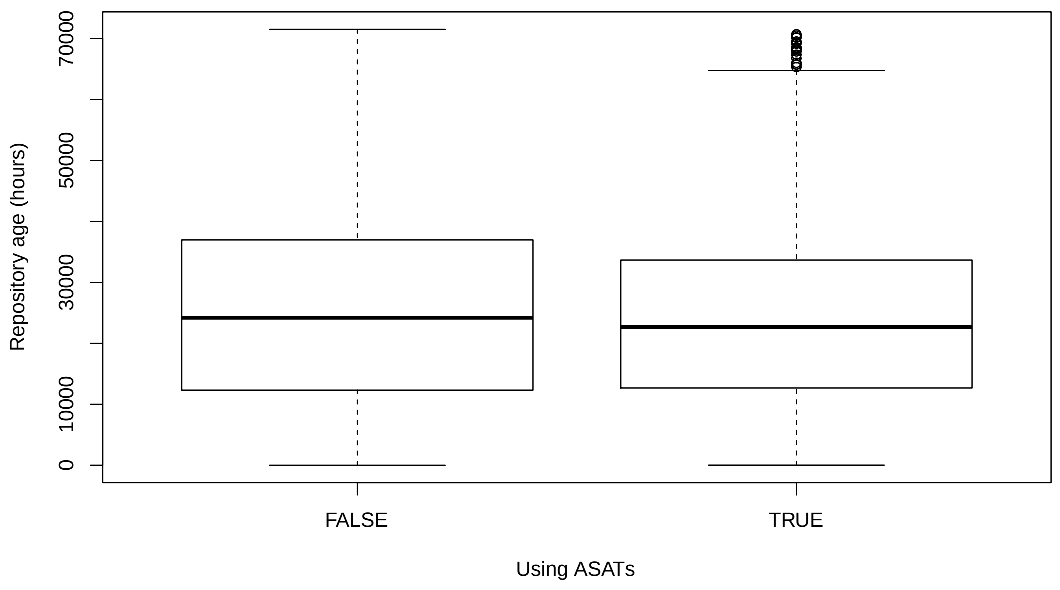


Figure 5.2: Distributions of repository age

To explain this, we look at the previous statistics on languages and ASAT use. Out of 2678 projects that use ASATs, 2154, or 80%, have JavaScript as their main language. For the remaining 6765 projects, this is 2350, or slightly below 35%. The average age for JavaScript projects is 22863 hours, which is well below the average age for all ASAT-using projects. Therefore, it is possible that the difference is caused by the higher percentage of JavaScript projects in ASAT-using projects, rather than whether ASATs are used.

To find out, we perform the same analysis on projects of each language separately. The results can be found in table 5.3. Here we see that repositories with ASATs are newer on average for JavaScript and Ruby, but older for Java and Python. The difference is statistically significant for all languages except Ruby. As mentioned, a majority of ASAT-using projects are in JavaScript, which negatively influences the mean age. The group of non-ASAT projects is dominated by the other three languages, which are older with the exception of Java, but this is compensated by the much higher average age for Ruby, leading to higher age overall. Because of the differences between languages, we cannot draw any conclusions about the relation between ASAT usage and repository age.

Language	Mean without ASATs	Mean with ASATs	*p*-value
JavaScript	23399	22274	0.013
Java	19862	28522	$5.45 \cdot 10^{-12}$
Ruby	38506	35943	0.052
Python	26784	30623	0.0067

Table 5.3: Mean ages and *p*-values of repositories with and without ASATs, per language

There is another factor that could influence the repository age averages. All four languages have been around since long before GitHub was launched, so it is possible that some projects migrated from other hosting (e.g. SVN, CodePlex, Google Code, or SourceForge) to GitHub at some point during their lifetime. In that case, our data makes these projects seem newer than they actually are. Unfortunately, we have no way to detect this automatically, and assume that the amount of migrated projects is roughly equal for all languages, and migrated projects are equally likely to use ASATs compared to original GitHub projects. Under this assumption, migrations do not significantly affect the results as presented.

5.1.4 ASATs and Star Count

The last basic property we present here is the number of stars a repository has, and compare this number to whether or not the project uses ASATs. This is interesting to know because intuitively, projects with a higher star count (hence more popular) are likely to be depended upon by more people. To ensure higher quality, ASATs could be introduced. Projects that do use them have 1688 stars on average, with a median of 636, whereas projects that do not have 1035 stars on average with a median of 484.5. The Mann-Whitney U test yields a *p*-value of $2.2 \cdot 10^{-16}$ ($U = 7797900$), so ASAT-using repositories seem to have more stars. Figure 5.3 shows the histogram for both groups of repositories. We opted for a histogram instead of a boxplot, because the data does not seem to fit a normal distribution and has a lot of outliers, which would result in a flat box. In both cases, we see a distribution leaning towards zero (the minimum here is actually 200, because that was the threshold for repositories to study) with a heavy tail. The star count with ASATs is more evenly distributed, since the y-axis only goes to 12, compared to 30.

5.2 Community Contributions

In this section we present several properties of the pull request data. For all properties except pull request count and lifetime density (i.e. amount of pull requests per hour), we

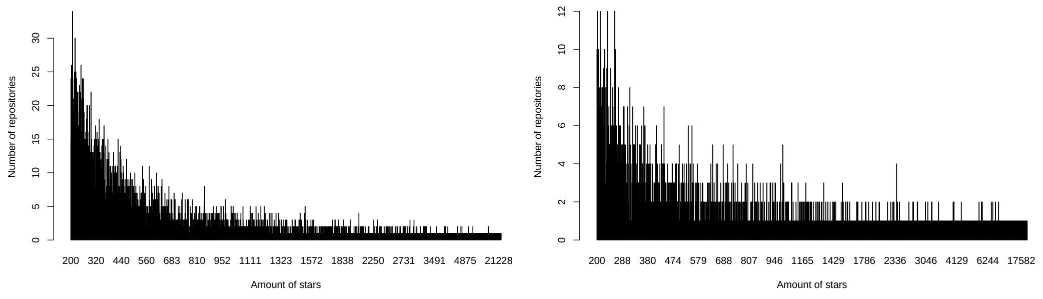


Figure 5.3: Star counts for projects without ASATs on the left, projects with ASATs on the right

limited ourselves to repositories that have received at least 100 pull requests, in order to have enough data for the results to be meaningful. This leaves us with 2353 repositories, of which 1465 do not use ASATs, and 888 do.

5.2.1 Pull Request Count

We will start off by comparing total pull request counts. We hypothesize that projects with high amounts of pull requests are more likely to have introduced ASATs at some point to make the process of handling pull requests easier and to enforce code conventions defined by the project maintainers. When not using ASATs, the median is 29 with an average 127.3. With ASATs, these numbers rise to 56 and 221.7 respectively. The test yields a p-value of $2.2 \cdot 10^{-16}$ ($U = 6926600$), showing that a larger number of total pull requests can be associated with ASAT use. Figure 5.4 shows the histograms for both groups.

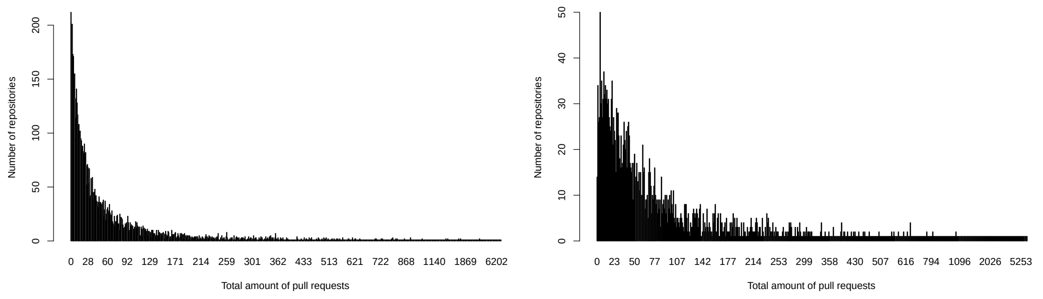


Figure 5.4: Pull request counts for projects without ASATs on the left, projects with ASATs on the right

5.2.2 Pull Request Density

Although we showed that projects using ASATs have more pull requests in total, older repositories are of course more likely to have more pull requests, and have had more time to include ASATs. Therefore, we normalize the pull request rate by using density instead.

We define the pull request density as the number of pull requests created per hour, and calculated both the lifetime density and the density during the last 100 pull requests.

The Mann-Whitney U tests for lifetime and recent density yield *p*-values of $2.2 \cdot 10^{-16}$ ($U = 6694000$) and $1.98 \cdot 10^{-6}$ ($U = 574490$) respectively, so even when normalizing for the age of the repository, use of ASATs can be associated with a higher rate of pull requests. More statistics can be found in table 5.4. Note that the recent density seems to be significantly higher; this is probably due to the fact that for recent density we only consider repositories with at least 100 pull requests, while lifetime density also includes repositories without any pull requests at all.

Timespan	Median	Mean	Max
Lifetime without ASATs	0.001	0.006	0.562
Lifetime with ASATs	0.003	0.010	0.891
Recent without ASATs	0.008	0.020	0.426
Recent with ASATs	0.010	0.026	1.111

Table 5.4: Pull request density averages

5.2.3 Unique Contributor Count

While the number of pull requests by itself provides some insight into a project's community activity, it would also be interesting to see how many different people contribute. More unique contributors could mean that it is easier for new contributors to pick up a project's style guide and/or conventions, because apparently lots of different people have done so before. ASATs would provide a way for a potential contributor to find out whether their contribution is suitable before actually submitting the pull request. Fewer contributors could indicate that the majority of pull requests are made by a core team, of which the members are familiar with the project's conventions already. This could be a reason for ASAT usage to be less likely.

However, the median number of contributors is 38 when not using, and 37 when using ASATs, with a *p*-value of 0.22 ($U = 669980$), rejecting the idea of a correlation between unique contributors and ASAT usage. This can also be observed in figure 5.5.

5.2.4 Amount of Merged Pull Requests

The raw number of incoming pull requests is a quantitative measurement, and a large number of them is still not beneficial if many of those pull requests are of low quality, and are rejected as a result. As a way to measure quality of a pull request, we look at the amount that is accepted and merged. Of the 100 pull requests, we counted the number that got merged instead of rejected in the end for each repository. We expect ASATs to increase the likelihood of a pull request being accepted, because a contributor can obtain immediate feedback about their pull request in a number of ways; the ASAT can be integrated to the IDE, they can run the build task if it is included there, or the CI check integrated into GitHub tells them as soon as the pull request has been submitted, giving the contributor time to adjust.

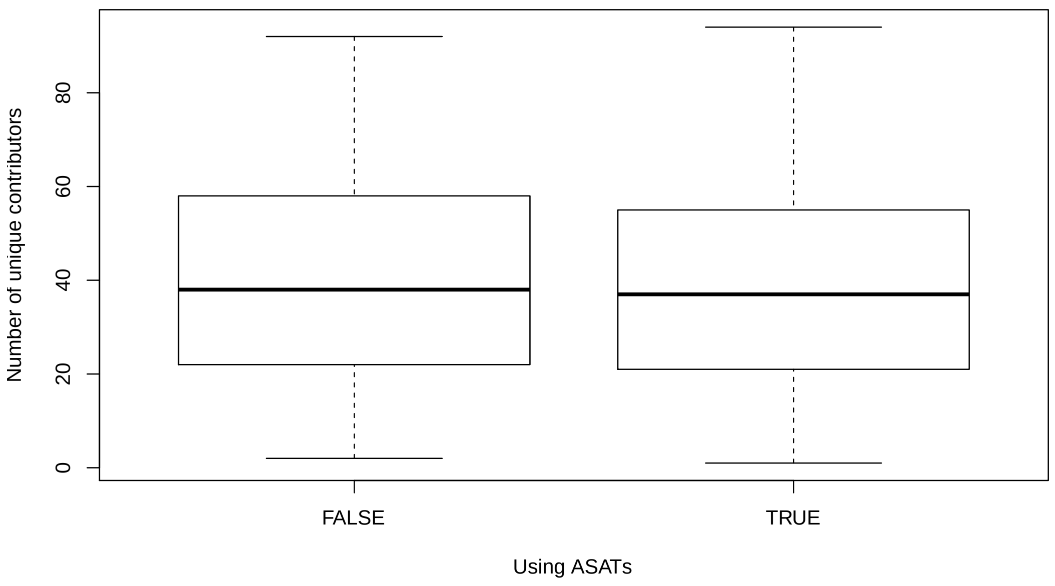


Figure 5.5: Amounts of different contributors

This seems to be supported in a survey involving project integrators[21], which shows that code quality and style are the top factors that influence the decision to accept a pull request. Without this feedback, the pull request may have been rejected due to style issues.

Contrary to this hypothesis, the median when not using ASATs is 79, versus 77 when using, with a p-value of 0.006 ($U = 694390$), hinting at an advantage without ASATs. This is visualized as a box plot in figure 5.6. A possible explanation is that integrators of projects that do not use ASATs are less concerned with code style issues – if they were, they would probably have enabled some ASAT, and are more likely to accept a pull request as a result.

There are other factors, unrelated to ASATs, influencing the decision to accept or reject a pull request. One study found that the decision to merge is mostly affected by whether the pull request targets an actively developed part of the project[20]. Another study that performed a survey among project maintainers found that without code style in the equation, the top factors for the decision become project fit, technical fit, and testing, with the main reasons for rejection being technical errors and failing tests[21]. Most contributors run tests before submitting a pull request[19], which filters out most technical errors and makes sure all tests pass. This survey also mentions that contributors want to format code according to the guidelines, as long as a project actually has said guidelines, which may play a role here.

Another factor could be the number of contributors in the last 100 pull requests. If this is low, there are likely some “core contributors”, who have more knowledge of the project and how pull requests should be done, so their pull requests are more likely to be accepted. Since the number of contributors and the number of pull requests both are ratio variables, and we want to test for linear correlation, we apply the Pearson correlation test. We obtained an estimated correlation coefficient of -0.37 with a p-value of $2.2 \cdot 10^{-16}$ for the alternative hypothesis of a negative correlation. Based on this, we can say the number of contributors

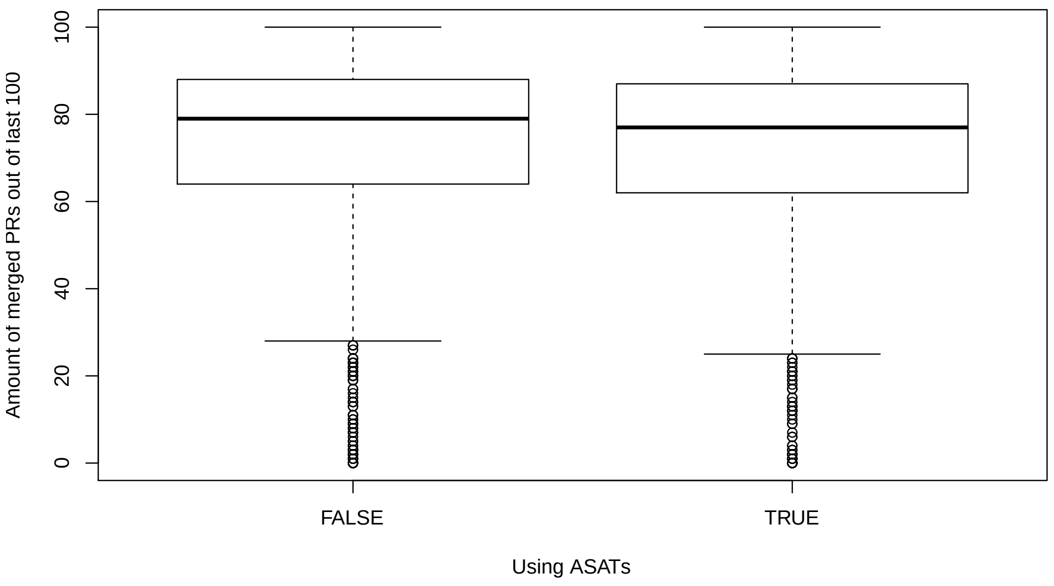


Figure 5.6: Numbers of merged pull requests (out of the last 100)

is negatively correlated with the number of merged pull requests.

In conclusion, we can say that ASATs have no demonstrable effect on the decision to accept or reject a pull request.

5.2.5 Pull Request Time to Close

For the final property, we look at how long it takes for pull requests to be accepted or rejected and the effect of ASAT usage. We presume that use of ASATs would reduce the time to respond to pull requests, because the code quality of the contribution would be higher. Previous research shows that code quality is not among the top factors in response time, and is dwarfed by reviewer availability[21]. Coming at a 5th spot, we still expect code quality to have a significant impact though; when conventions are followed, project maintainers have an easier time reviewing the changes. Other factors include the developer's track record, project size, and test coverage[20]. Per project, we look at the average time to close in seconds.

With medians 854455 (without) versus 707913 (with) seconds, means of 1674847 versus 1392057 seconds and a p-value of 0.018 ($U = 688300$), there indeed seems to be a correlation between use of ASATs and time to close. With ASATs, the time it takes to review pull requests is almost 17% lower on average. Furthermore, without ASATs the variance is 67% higher. As seen in figure 5.7, there are some large outliers within the group of projects without ASATs, explaining the difference in variance.

Like with the merge rate, we also look at the effect of the amount of contributors on the time to close. This time, we find a correlation coefficient of 0.49 with a p-value of

$2.2 \cdot 10^{-16}$, indicating that pull requests take less time to review for projects with less contributors.

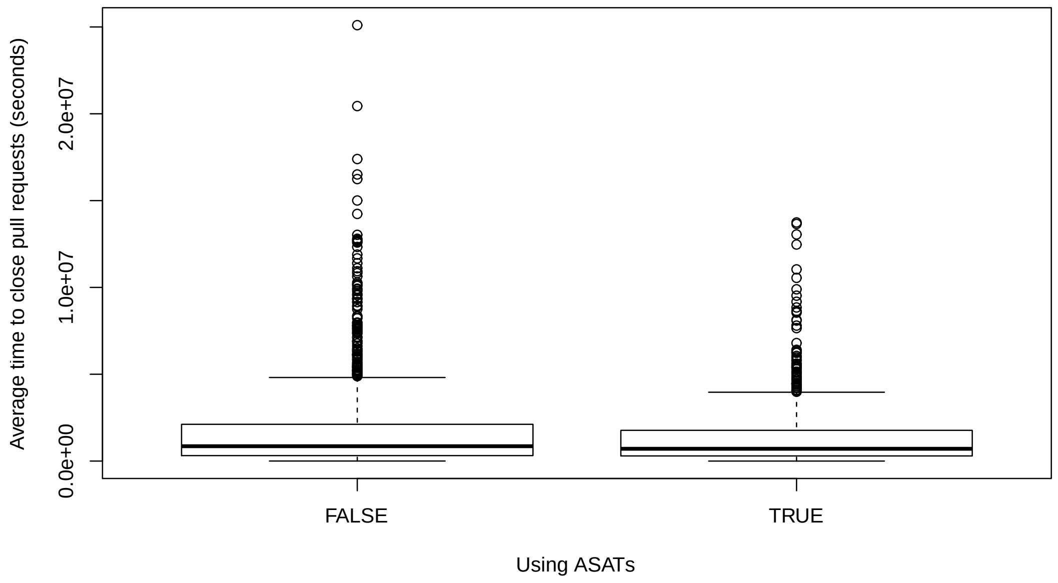


Figure 5.7: Time it takes to review pull requests on average

5.3 Warnings in Committed Code

The data obtained by running ASATs over several projects of each language forms the largest part of our results. In this section, these results are presented in different ways. First, we look at several statistics on warning count aggregates per project. Then, we present the development of total warning counts over time for several projects. Finally, we look into the different warning categories, analyzing their occurrences and solve rates.

For this study we ran the different ASATs on the code of 39 repositories. We limited the analysis to one ASAT per repository in order to compare warning counts between repositories (with multiple ASATs for some repositories, this comparison would not be valid). These are listed in table 5.5, along with their language, tool used for analysis, number of analyzed commits, and total number of warnings found. We selected the repositories with the largest amount of stars that also contain a configuration file for their ASAT. We analyzed up to 500 commits per repository, starting at the earliest commit which contained a configuration file that was compatible with the same version of the ASAT as said configuration file of the last commit. In total, we ended up with 11.863.395 warnings in the database.

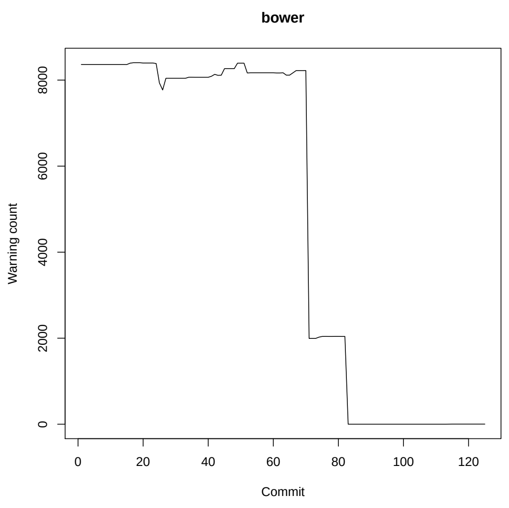
5.3.1 Warning Counts over Time

In this part, we look at how the number of warnings changes over time. For a select number of projects, we show a graph that displays the total warning count over the number of

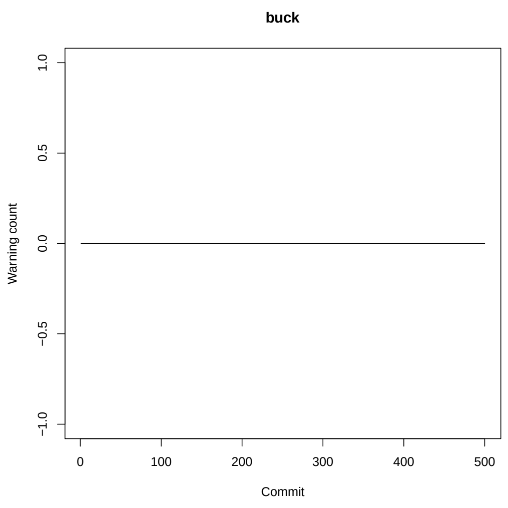
Repository	Language	ASAT	Commits	Warnings
mongodb/morphia	Java	checkstyle	64	0
square/retrofit	Java	checkstyle	236	0
bumptech/glide	Java	checkstyle	498	716
scribejava/scribejava	Java	checkstyle	83	16
google/auto	Java	checkstyle	118	1926
zeromq/jeromq	Java	checkstyle	81	120139
facebook/buck	Java	pmd	500	0
capitalone/Hygieia	Java	pmd	88	1871
checkstyle/checkstyle	Java	pmd	500	28
Netflix/servo	Java	pmd	60	4801
OpenGrok/OpenGrok	Java	pmd	419	261696
sleekbyte/tailor	Java	pmd	162	0
jashkenas/backbone	JavaScript	eslint	55	245
FreeCodeCamp/FreeCodeCamp	JavaScript	eslint	500	20206
gulpjs/gulp	JavaScript	eslint	45	45
nnnick/Chart.js	JavaScript	eslint	158	11041
jashkenas/underscore	JavaScript	eslint	52	1511
vuejs/vue	JavaScript	eslint	206	479
bower/bower	JavaScript	eslint	125	599825
remy/nodemon	JavaScript	jscs	44	7
jshint/jshint	JavaScript	jscs	248	21
requirejs/requirejs	JavaScript	jscs	65	0
gruntjs/grunt	JavaScript	jscs	60	118
hexojs/hexo	JavaScript	jscs	79	520
jquery/jquery	JavaScript	jshint	500	196
moment/moment	JavaScript	jshint	252	0
caolan/async	JavaScript	jshint	371	418
select2/select2	JavaScript	jshint	500	20
less/less.js	JavaScript	jshint	500	7611
SirVer/ultisnips	Python	pylint	257	38356
numenta/nupic	Python	pylint	500	3923892
rembo10/headphones	Python	pylint	104	14530
pyinstaller/pyinstaller	Python	pylint	500	3367394
cython/cython	Python	pylint	500	3362930
CocoaPods/CocoaPods	Ruby	rubocop	109	0
ruby-grape/grape	Ruby	rubocop	196	14838
capistrano/capistrano	Ruby	rubocop	51	421
sass/sass	Ruby	rubocop	84	316
thoughtbot/paperclip	Ruby	rubocop	116	107262

Table 5.5: Repositories of which we obtained warnings for a range of commits

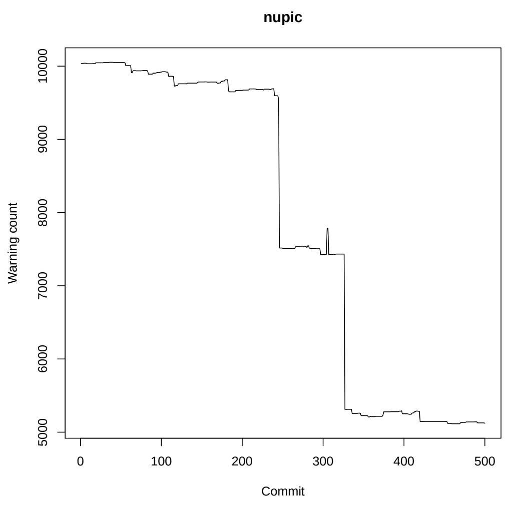
commits that we analyzed, and try to find an explanation for the graph’s characteristics. This way, we hope to find different ways for warnings to be solved or introduced.



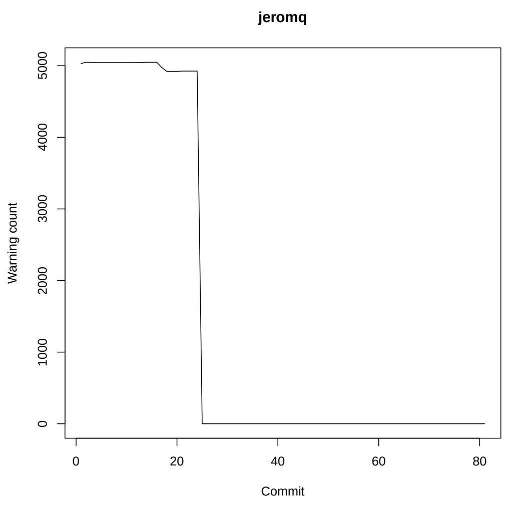
(a) bower/bower – 125 commits



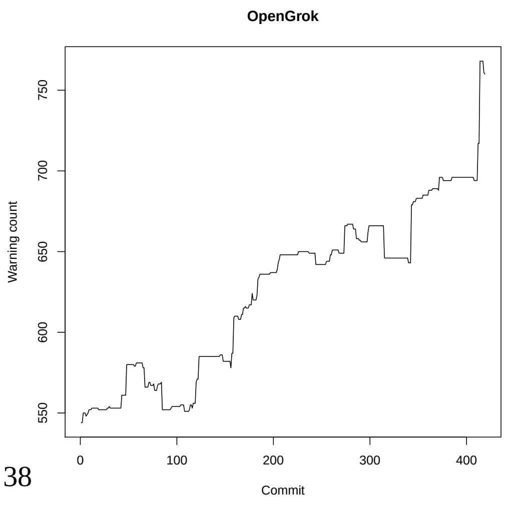
(b) facebook/buck – 500 commits



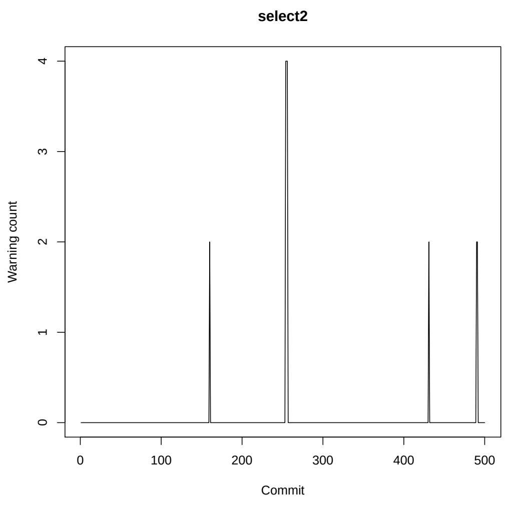
(c) numenta/nupic – 500 commits



(d) zeromq/jeromq – 81 commits



(e) opengrok/opengrok – 419 commits



(f) select2/select2 – 500 commits

Figure 5.8: Warning count graphs for several noteworthy projects

`bower/bower` (figure 5.8a) initially hovers slightly above 8000 warnings. However, around commit #70 and then again around #80, a major amount of warnings is resolved. The first drop is due to a change in ESLint settings, changing indentation size[1], and the commit of the second drop featured another overhaul of the ESLint configuration, along with a large amount of code style fixes[2]. In addition, JSCS is dropped from the repository altogether in favor of ESLint, and Grunt tasks for ESLint are added, having them run in Travis. This explains why warnings stay close to zero from that point forward.

`facebook/buck` (figure 5.8b) did not contain a single warning in 500 commits. This is achieved by running PMD during each Travis build, using Ant. The website contains a comprehensive section on contributing, including a style guide[3] which mentions that PMD checks are used.

`numenta/nupic` (figure 5.8c) starts out with a warning count at around 10000. Over time, two big drops can be observed. The first one is caused by a change in configuration, changing indentation size to a value that was being used in most of the codebase already. The second one is a commit moving a lot of code to another repository[4], so the violating code is removed rather than fixed. For this project, warnings were reduced without changing any of the code.

`zeromq/jeromq` (figure 5.8d) hovers around 5000 warnings for some time. However, at some point, they seem to have had enough, and fix them all in a single commit[5]. In this same commit, Checkstyle is added to the build process and as a result the warning count stays at 0. This case is remarkable because all warnings were reduced by solving them, rather than changing ASAT configuration or removing code.

`opengrok/opengrok` (figure 5.8e) is one of the few projects where the warning count actually increases over time. At a couple of points, drops can be observed but those are soon followed by more warnings. PMD is not run by Travis, which could be an explanation.

`select2/select2` (figure 5.8f) remains at zero warnings for the majory of its commits. However, we can observe a handful of instances where some warnings slipped through. These were all solved within a couple commits. The commit after the largest peak (4 warnings) did not only fix the existing warnings, but also added some rules to JSHint configuration and solved all violations that emerged because of that[6]. This project also runs JSHint through Grunt in Travis, so the low amount of warnings can be attributed to that. The contributing guide does not mention code conventions, however.

5.3.2 Average Warning Count

To obtain insight into the warning counts of each repository, we calculated the average number of warnings per commit. Because not all projects have an equally sized code base, we also determined the lines of code of each repository, using a command-line tool called

[1]https://github.com/bower/bower/commit/78e443db0afb1889a9ec53c2ff94cce1b1ab017b
[2]https://github.com/bower/bower/commit/53eeca97d38945751075932ac6e6841ebeaeeb0a
[3]https://buckbuild.com/contributing/codestyle.html
[4]https://github.com/numenta/nupic/pull/1866
[5]https://github.com/zeromq/jeromq/commit/14d9dc1aaae0518433944fa664a907f575e72cdf
[6]https://github.com/select2/select2/commit/081580bcd82dce16df5a5dae5b03ef1f92420191

`cloc`[7]. Using this value, we calculated the amount of warnings per 100 lines of code, which is more representative than total warning count. The result of this can be found in table 5.6, where repositories are sorted by the number of warnings per 100 lines of code. Using this table, we look for projects that have uncommon amounts of warnings per line of code, and try to find an explanation. Furthermore, we investigate if there is any relation between the total lines of code and the amount of warnings per line of code.

It turns out that only 12 out of 39 repositories averaged more than 1 warnings per 100 lines of code. 7 repositories never had any warnings at all. This tells us that most projects mostly seem to adhere to the rules they set for their ASATs. Interestingly, this seems to be especially true for projects with large code bases. 8 out of the top 10 largest projects have a negligible amount of average warnings compared to the lines of code. Both of the other two (Python) projects do not run Pylint in Travis, whereas the other 8 all do. This also seems to hold the other way around; `bower/bower`, the top offender, did not run ESLint in Travis during the commits that yielded most warnings, and when it started doing so, warnings dropped drastically. The same holds for `zeromq/jeromq`. The rest of the top 10 projects with the most warnings per line of code does not include their ASAT in Travis, with the exception of `ruby/grape` and `netflix/servo`, at the 9th and 10th spots, both under 2 warnings per 100 lines.

`thoughtbot/paperclip` stands out due to having a high amount of warnings per line of code (29.9 per 100), despite a relatively small code base of 3093 lines. In line with previous observations, we find that this project does not include RuboCop in Travis. The only evidence of RuboCop usage is a configuration file, and it seems to be integrated into Hound CI[8] instead of Travis. However, Hound does not fail upon encountering RuboCop warnings. In an issue raised on the amount of warnings[9], a maintainer answered "*This is not an issue we want to tackle all at once. All new code coming into the project should comply as best as possible with rubocop rules. Thanks for reporting.*". This, combined with the fact that RuboCop was added to the project at a relatively late moment[10] (December 2014, while the repository was created in April 2008), explains the high warning rate.

Following these findings, we investigate the relation between project size and warnings per line of code. It looks like the large projects are more likely to make efforts to keep the warning counts low. Figure 5.9 shows a scatter plot of the lines of code versus the amount of warnings per 100 lines. This shows us that most projects have both low amounts of code and warnings. As mentioned earlier, the projects with large code bases are low on warnings, but any correlation seems unlikely. Indeed, a Pearson correlation test yields a *p*-value of 0.6. To draw any conclusions, more data would be needed.

Considering the previously mentioned top 10 projects with most warnings per line of code, we also hypothesize that integrating the ASAT into Travis and failing the build upon warnings can be associated with low warning counts. Projects that include an ASAT in their Travis script have an average of 2 warnings per 100 lines of code, versus 12.5 warnings for

[7]https://github.com/AlDanial/cloc
[8]https://houndci.com
[9]https://github.com/thoughtbot/paperclip/issues/2062
[10]https://github.com/thoughtbot/paperclip/pull/1733

Repository	Average warnings/commit	Lines of code	Warnings/100 LoC
bower/bower	4798.6	10285	46.66
pyinstaller/pyinstaller	6734.79	18083	37.24
thoughtbot/paperclip	924.67	3093	29.9
numenta/nupic	7847.78	31163	25.18
cython/cython	6725.86	52787	12.74
zeromq/jeromq	1483.2	23621	6.28
SirVer/ultisnips	149.25	3469	4.3
OpenGrok/OpenGrok	624.57	27508	2.27
ruby-grape/grape	75.7	4367	1.73
Netflix/servo	80.02	6746	1.19
rembo10/headphones	139.71	12128	1.15
nnnick/Chart.js	69.88	6395	1.09
jashkenas/underscore	29.06	4336	0.67
capistrano/capistrano	8.25	1398	0.59
FreeCodeCamp/FreeCodeCamp	40.41	8087	0.5
capitalone/Hygieia	21.26	4726	0.45
jashkenas/backbone	4.45	1166	0.38
less/less.js	15.22	9272	0.16
gulpjs/gulp	1	719	0.14
google/auto	16.32	19590	0.08
gruntjs/grunt	1.97	3371	0.06
sass/sass	3.76	13104	0.03
caolan/async	1.13	7431	0.02
vuejs/vue	2.33	18575	0.01
remy/nodemon	0.16	1657	0.01
bumptech/glide	1.44	31657	0
scribejava/scribejava	0.19	7871	0
hexojs/hexo	6.58	371236	0
jquery/jquery	0.39	31809	0
jshint/jshint	0.08	8069	0
select2/select2	0.04	17456	0
checkstyle/checkstyle	0.06	29808	0
mongodb/morphia	0	31095	0
moment/moment	0	39157	0
sleekbyte/tailor	0	4663	0
facebook/buck	0	156462	0
requirejs/requirejs	0	181720	0
square/retrofit	0	7316	0
CocoaPods/CocoaPods	0	8798	0

Table 5.6: Average warning counts per commit for each repository

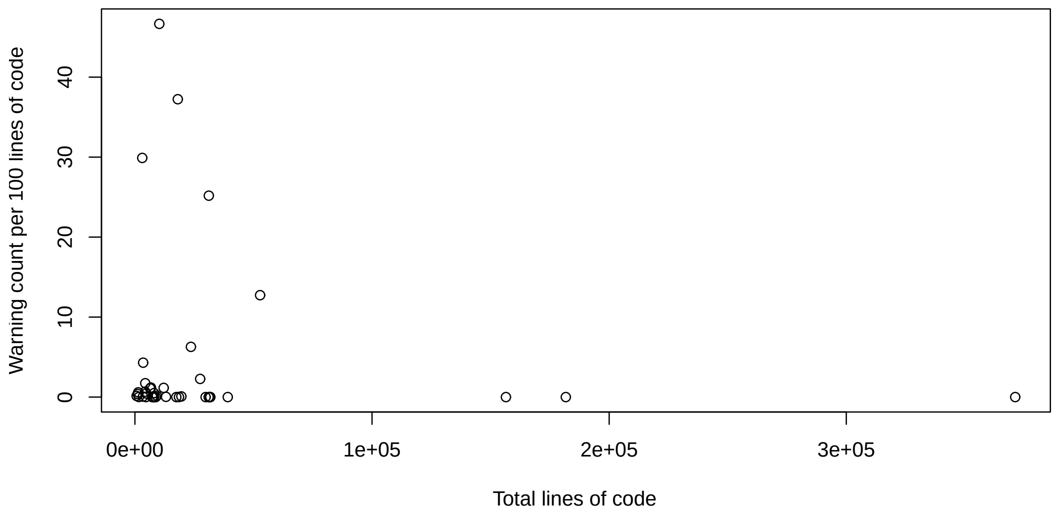


Figure 5.9: Scatter plot of lines of code and warnings per 100 lines

projects that do not. This looks like a significant difference and indeed, the Mann-Whitney U test yields a p-value of 0.003 (two-tailed, $U = 222.5, n_1 = 9, n_2 = 30$).

5.3.3 Prevalence per Warning Category

To say something about which kind of warnings show up most often, we want to analyze the total counts of different warning types. However, since each ASAT has its own set of rules, we cannot directly compare rules across ASATs. Therefore, we use the General Defect Classification(GDC)[8]. Using this classification, we mapped rules of each ASAT to one of the 18 more general categories. These categories have descriptive names, and are generally easier to understand than the rule names of ASATs themselves. Figure 5.10 shows the total amount of warnings we encountered across the study for each category.

Not all projects influence this figure equally. In table 5.5, we can see that Python projects have generated most warnings, with a total amount of over 10 million. Considering the data contains less than 12 million warnings in total, we also counted the categories for each language separately to account for this imbalance. Figure 5.11 displays these category counts per language.

For Ruby, most warnings are from the Metric category. Coincidentally, rules of this category are often enabled in RuboCop[8], explaining this observation. A large majority within this category belongs to the `LineLength` RuboCop rule, which defines a maximum amount of characters a line can have. Upon closer inspection, we find that most of the offending lines are either comments or string literals, and likely not considered a problem. Other than that, line length could also be considered to be a style convention, but because the rule has a configurable number, it has been classified as Metric; this could be an indication that some rules can be in multiple GDC categories.

For Python, the Logic category is most numerous. Upon closer inspection, we find that

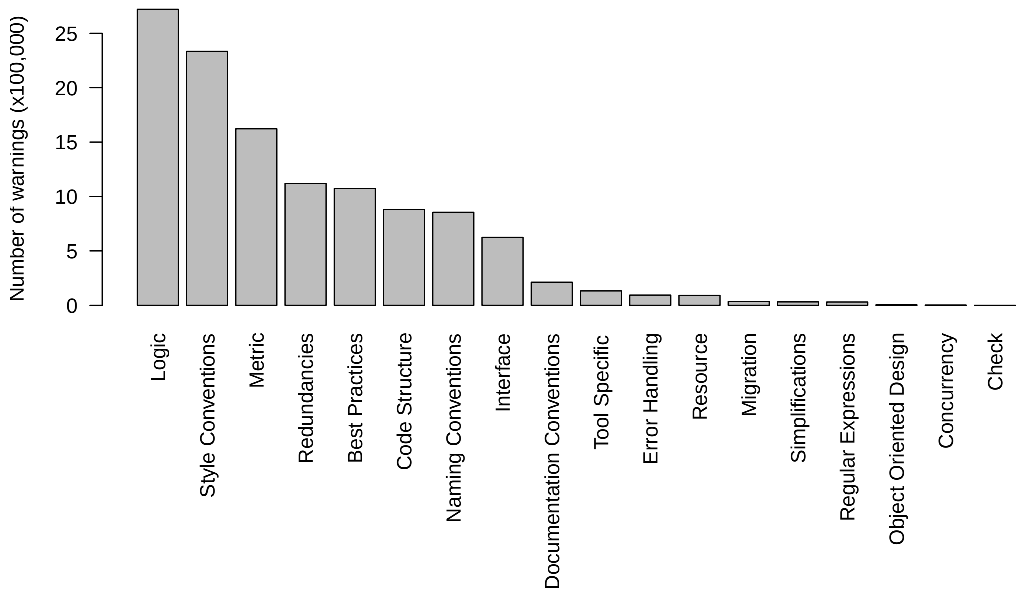


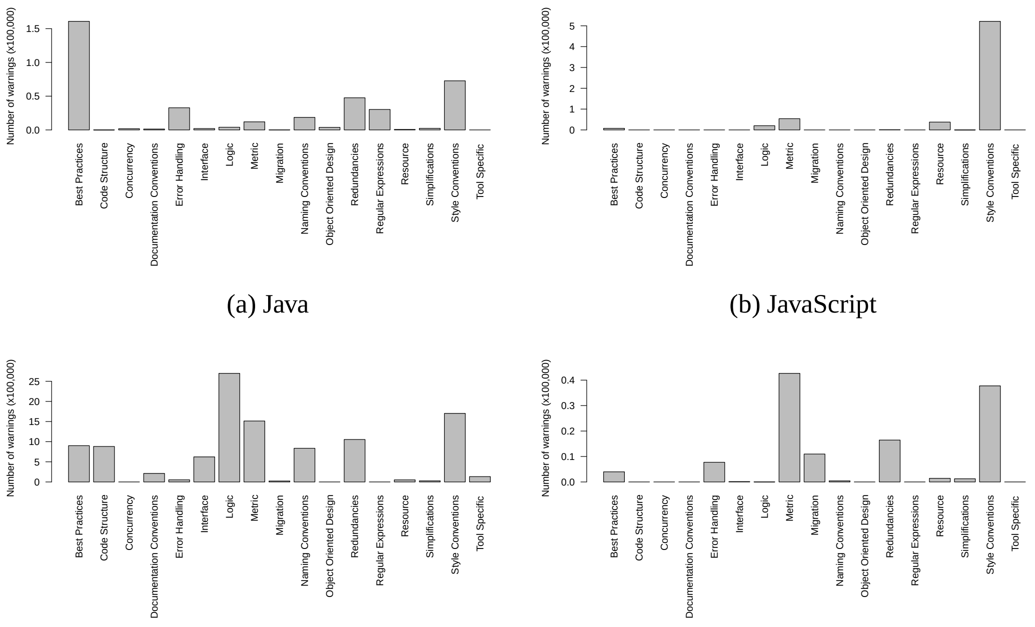
Figure 5.10: GDC warning category counts overall

87% of these warnings originate from the `bad-continuation` Pylint rule. A warning is shown when function arguments on a new line do not align. This actually seems to be a misclassification in the GDC, as it has no effect on the correctness of the program, and should probably be in the Style Conventions category. Apparently, these issues are not considered to be important enough to solve quickly; it is possible that they were present before the ASAT's introduction, and solving them has low priority.

For JavaScript, the large majority of warnings is from Style Conventions, of which 83.5% comes from the `indent` rule, available in both ESLint and JSHint, and 16% from `space-before-function-paren` available in ESLint. The former requires indentation of 4 (by default) spaces per level, and the latter requires a space after the `function` keyword. Like with Python, these are not major maintainability issues, and solving them is probably of low priority.

For Java, Best Practices comes out on top. Within this category, rules are slightly more evenly distributed than for the other languages: 49% is from `FieldDeclarationsShouldBeAtStartOfClass`, 18% from `GuardLogStatementJavaUtil`, 10% from `ConsecutiveAppendsShouldReuse`, all of which are PMD rules. It turns out that all instances of `FieldDeclarationsShouldBeAtStartOfClass` (which requires class fields to be declared before any class methods) are found in `OpenGrok/OpenGrok`. The instances where this happens, the class field is grouped with the method where it is used, but the field is not used anywhere else. This may be a violation of the Single-Responsibility Principle, so instead of solving the warning by moving the field declaration, a better solution would be to split the class up into multiple classes. An example is the `Configuration` class, which is over 950 lines long, has many different methods, and a constructor of 45 lines. These are indications of a God Class.

We also see that warnings of the Style Conventions category are common for all languages. In addition, the rules that are violated most often in Ruby and Python could be considered to be Style Conventions as well. An explanation for this could be that Style Conventions are categorized under Maintainability Defects, meaning they have no effect on the correctness of the program. Compared to other Maintainability Defects, Style Conventions should have little impact on readability, because they mostly are about indentation, line breaks and whitespace. Possibly, project maintainers would ideally like to see these issues resolved, but other issues take precedence.



(a) Java (b) JavaScript

(c) Python (d) Ruby

Figure 5.11: GDC warning category counts per language

5.3.4 Solve Rate per Warning Category

Finally, we present the solve rates per GDC category. We opted to express this as the number of commits instead of real time because we feel the number of commits is a better representation of the actual time spent working than number of days passed; one project may be only worked on in the weekend, while another is someone's full time job. We only considered warnings that have been introduced after the first commit we analyzed, and solved before the last one, otherwise we cannot properly determine how long a violation had been present. Figure 5.12 displays the average of all solve time measurements per category and figure 5.13 shows the medians. In the remainder of this section, we will explore whether the solve times of different categories significantly differ from one another, and try to order the categories in a way that is supported by the data.

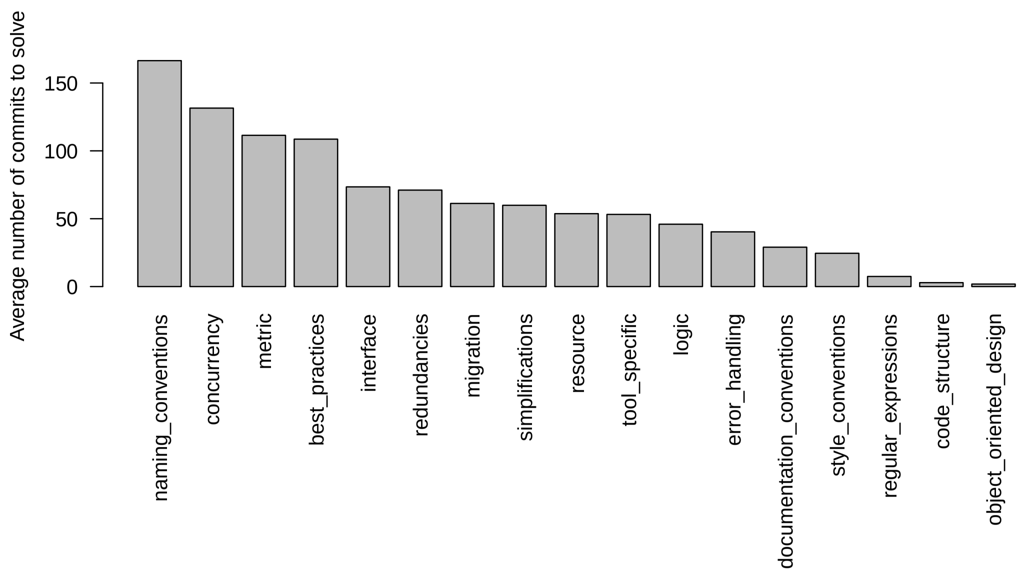


Figure 5.12: Average solve times for each GDC category

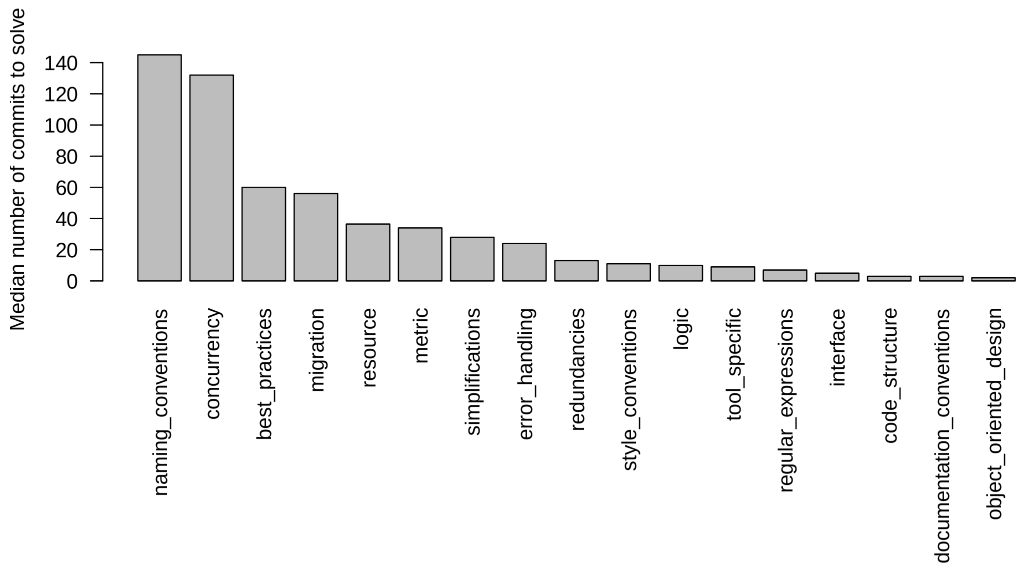


Figure 5.13: Median solve times for each GDC category

We excluded solve times of one Python repository, `numenta/nupic`, from the results. The reason for this is a series of commits that first "solves" a lot of warnings by moving code to another directory, hence changing the file path. Later, this change is reverted[11] because of some complications, and shortly after, it is applied again[12], presumably in the correct way. Because of this, a lot of warnings were re-introduced in one commit, then removed

[11]https://github.com/numenta/nupic/pull/2583
[12]https://github.com/numenta/nupic/pull/2585

again shortly after, making the data biased.

The average solve times do seem to differ from one another, as can be seen in figure 5.12. However, the order is quite different from figure 5.13, which could mean the averages are affected by outliers. To find out if there is a statistically significant difference between the category solve times, we perform a one-tailed Mann-Whitney U test (by means of the Shapiro-Wilk test, we determined that none of the solve times have a normal distribution) to each pair of categories. Table 5.7 lists the *p*-values of each test. Each cell is the result of testing its row category to be smaller than its column category. Assuming a significance level of 0.05, cells with values that indicate a statistically significant difference are colored white, versus grey cells for test results where the *p*-value is higher; for each white cell, the row category solve times are significantly smaller than the column category solve times, and vice versa. For a complete picture, tables 5.8 and 5.9 show the corresponding U-values, and table 5.10 shows the number of data points (solve times) obtained for each category.

	BP	CS	C	DC	EH	I	L	M	M	NC	OOD	R	RE	R	S	SC	TS
Best Practices		1	0.48	1	1	1	1	0.9	0.46	0	1	1	1	1	0.89	1	1
Code Structure	0		0	0.07	0	0	0	0	0	0	1	0	0	0	0	0	0.04
Concurrency	0.52	1		0.99	0.74	0.93	0.91	0.6	0.56	0.35	1	0.87	0.57	0.7	0.83	0.88	0.93
Documentation Conventions	0	0.93	0.01		0	0	0	0	0.01	0	1	0	0	0	0	0	0.07
Error Handling	0	1	0.26	1		1	1	0	0.02	0	1	0.78	1	0.1	0.25	1	0.94
Interface	0	1	0.07	1	0		0.78	0	0.11	0	1	0.01	0	0	0.11	0.44	0.86
Logic	0	1	0.09	1	0	0.22		0	0.05	0	1	0	0.99	0	0.04	0.74	0.53
Metric	0.1	1	0.4	1	1	1	1		0.31	0	1	1	1	0.92	0.73	1	1
Migration	0.55	1	0.56	0.99	0.98	0.89	0.95	0.69		0.13	1	0.86	1	0.93	0.75	1	0.87
Naming Conventions	1	1	0.65	1	1	1	1	1	0.87		1	1	1	1	1	1	1
Object Oriented Design	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0.06
Redundancies	0	1	0.13	1	0.22	0.99	1	0	0.14	0	1		1	0.1	0.4	1	0.98
Regular Expressions	0	1	0.43	1	0	1	0.01	0	0	0	1	0		0	0	0	0.15
Resource	0	1	0.31	1	0.9	1	1	0.08	0.07	0	1	0.9	1		0.37	1	0.96
Simplifications	0.11	1	0.2	1	0.75	0.89	0.96	0.27	0.28	0	1	0.6	1	0.63		0.98	0.91
Style Conventions	0	1	0.12	1	0	0.56	0.26	0	0	0	1	0	1	0	0.02		0.5
Tool Specific	0	0.96	0.08	0.93	0.06	0.14	0.47	0	0.14	0	0.94	0.02	0.85	0.04	0.09	0.5	

Table 5.7: *p*-values of one-tailed statistic tests for each pair of classification categories

	BP	CS	C	DC	EH	I	L	M	M
Best Practices		5908.5	395	22746.5	61049	24377	52714.5	68951.5	386.5
Code Structure	1091.5		0	2192.5	2170	2345	4397.5	5293.5	0
Concurrency	405	140		483.5	1139	541	1076.5	1394.5	8
Documentation Conventions	6853.5	2987.5	108.5		17813	11391	22538	27791.5	79
Error Handling	35151	14665	785	53375		53071	109278	137115	372
Interface	13423	4270	215	16581	37838		38309	48358.5	244
Logic	25285.5	9252.5	483.5	35182	78312	35401		94291	416
Metric	61048.5	17456.5	1205.5	68408.5	175535	74491.5	159209		1113.5
Migration	413.5	140	8	513	1552	512	1144	1486.5	
Naming Conventions	41291	10475.5	751.5	41165.5	117199.5	45721	100406	136550	893.5
Object Oriented Design	259	22	0	396	405	485	1141	1321	0
Redundancies	25549	8722	437	32614.5	75443.5	34184	72595.5	91831.5	442
Regular Expressions	36913	25798.5	1450	89457.5	86750	83547	134282	165649.5	0
Resource	9148.5	3485	197	12952.5	30024	13056	27173.5	34595.5	129.5
Simplifications	1396.5	511.5	24	1860	4477.5	1896	4133	5044.5	27
Style Conventions	40854.5	19026	951.5	68777	128355	68308.5	136764.5	168677.5	218.5
Tool Specific	2977.5	964	51.5	3802.5	9300	3823	8721.5	10509.5	60

Table 5.8: *U*-values of one-tailed statistic tests for each pair of classification categories, part 1

	NC	OOD	R	RE	R	S	SC	TS
Best Practices	26109	2141	39251	112487	14051.5	2003.5	102745.5	6022.5
Code Structure	1319.5	398	2618	346.5	575	83.5	6104	611
Concurrency	596.5	48	859	1538	267	44	1920.5	128.5
Documentation Conventions	8710.5	1380	15337.5	21098.5	4215.5	656	37487	2857.5
Error Handling	44897.5	5367	80400.5	272557	25772	3699.5	217003	12345
Interface	17972	1783	27052	57636	8868	1317	67393.5	4682
Logic	31024	3539	53764.5	157048	18066.5	2497	143255.5	8828.5
Metric	82500	6479	118768.5	319900.5	40804.5	6005.5	298022.5	18740.5
Migration	454.5	48	854	2988	334.5	41	2653.5	120
Naming Conventions		3843	74997	199229	27411	3930.5	188205	11501.5
Object Oriented Design	201		574	45	169	16	1433	192
Redundancies	34191	3314		141076.5	17274.5	2650	133436.5	8629.5
Regular Expressions	52510	8919	100951.5		14644.5	4453	231454.5	15768.5
Resource	11681	1223	20309.5	72007.5		937	54839	3065.5
Simplifications	1798.5	188	2858	8246	1035		7851.5	466
Style Conventions	53761	7183	99195.5	304891.5	28449	4354.5		16152.5
Tool Specific	3663.5	348	5950.5	17846.5	2154.5	299	16157.5	

Table 5.9: U-values of one-tailed statistic tests for each pair of classification categories, part 2

Category	Number of Solve Times
Best Practices	200
Code Structure	35
Concurrency	4
Documentation Conventions	148
Error Handling	481
Interface	189
Logic	390
Metric	650
Migration	4
Naming Conventions	337
Object Oriented Design	12
Redundancies	324
Regular Expressions	747
Resource	116
Simplifications	17
Style Conventions	718
Tool Specific	45

Table 5.10: Solve time counts for each category

To obtain a more reliable order of categories than the averages, we count the sum of significant results (white cells) for each row and column of table 5.7. Per category, we add the numbers of its row and column counts to obtain the total number of categories that are significantly different. Since we have 17 different categories, each category is paired with 16 others in the significance tests. To reliably establish an ordering, we discard categories that significantly differ from less than half (8) of the other categories. This leaves out Concurrency, Migration, Simplifications and Tool Specific. The counts can be found in table 5.11, where the discarded categories are colored gray.

We can sort the remaining categories in two ways: by the “less than” count ascending or “greater than” count descending. The other count can be used as tiebreaker, e.g. Best Practices and Metric both have significantly smaller solve times than 1 other category, but Best Practices has significantly larger solve times than 11 categories versus 7 of Metric, so Best Practices comes first. In the first two columns of table 5.12, we see that these orderings differ slightly; the Metric and Interface categories have shifted two places down in the “greater than” order. Otherwise, the orderings are the same. Looking back at table 5.7, we see that Metric solve times are significantly longer than Redundancies. Compared to Resource, the *p*-value was 0.08 (only slightly above significance level 0.05), so putting Metric above those two in the ordering seems fair, agreeing with the “less than” ordering. As for Interface, since this category is significantly faster than Regular Expressions, which is in turn significantly faster than Style Conventions, we consider Interface to be below the other two, which follows the “greater than” ordering. Following this reasoning, the third column shows the ordering that we finally decided upon. At the top, we see Naming Conventions and Best Practices, while Object Oriented Design and Code Structure have the lowest solve

Category	Significantly smaller than	Significantly larger than	Total
Best Practices	1	11	12
Code Structure	14	1	15
Documentation Conventions	13	1	14
Error Handling	4	7	11
Interface	7	3	10
Logic	7	4	11
Metric	1	7	8
Naming Conventions	0	14	14
Object Oriented Design	15	0	15
Redundancies	3	8	11
Regular Expressions	10	4	14
Resource	2	8	10
Style Conventions	8	4	12
Concurrency	0	3	3
Migration	0	6	6
Simplifications	1	6	7
Tool Specific	5	1	6

Table 5.11: Significant difference counts per category

time. A possible explanation for this is the relation these categories have with maintainability; the slow categories have little impact on program correctness or maintainability, while the fast ones do.

Order by “less than” count	Order by “greater than” count	Final decided order
Naming Conventions	Naming Conventions	Naming Conventions
Best Practices	Best Practices	Best Practices
Metric	Resource	Metric
Resource	Redundancies	Resource
Redundancies	Metric	Redundancies
Error Handling	Error Handling	Error Handling
Logic	Logic	Logic
Interface	Style Conventions	Style Conventions
Style Conventions	Regular Expressions	Regular Expressions
Regular Expressions	Interface	Interface
Documentation Conventions	Documentation Conventions	Documentation Conventions
Code Structure	Code Structure	Code Structure
Object Oriented Design	Object Oriented Design	Object Oriented Design

Table 5.12: Category ordering for solve times, largest to smallest

Next, we will look at the solve times of the two top and two bottom categories, to find out which projects they originated from, and which commits introduced and solved the warnings. This way we seek to explain why categories have high or low solve times by anecdotal evidence.

At the bottom, we find Object Oriented design. In total, we found 12 solve times for this category, 10 of which came from `checkstyle/checkstyle`. The warnings were introduced with a commit that removed a suppression[13], and solved just 2 commits later[14]. Upon closer inspection, checkstyle turns out to be a project that aims to keep warnings out at all times, so the low solve time for the category may be attributed to the fact that most of the solve times came from checkstyle, rather than the nature of the category itself.

Warnings categorized as Code Structure are a close second. Most the solve times originate from `rembo10/headphones`, where a single wildcard import caused 29 warnings (one for each unused module imported). Soon after, a pull request was submitted that solved a couple of Pylint warnings[15]. Because Pylint generates so many warnings for a single violation, the real amount of solve times is actually lower than the data says. With this instance counted as one, only 7 data points would remain. This weakens the evidence of Code Structure's place in the ranking.

The warnings of the Naming Conventions category—and more importantly, their high solve times—mostly originate from `pyinstaller/pyinstaller`. We analyzed the warning counts per commit and found one commit that removed around 5000 warnings. The accompanying pull request[16] is a major overhaul of the Pylint configuration file, explaining the drop in warning count. Many of these warnings had been present for a long time, resulting in a long solve time, and because of the large number, causing the average of Naming Conventions to be the highest. The solve times of other projects for this category are much lower, however. Therefore we cannot assume Naming Conventions to be solved slowly in general.

We also look at Best Practices, which turns out to have a high solve time for multiple projects. `cython/cython` and `pyinstaller/pyinstaller` contributed to the high solve times with averages of 270 and 197 commits respectively, but `OpenGrok/OpenGrok` yielded most solve times for this category (62), so we decided to look at this project. It turns out that most warnings were about class fields not coming before methods, which must have been low priority for maintainers. However, eventually, a large amount was fixed in a single commit[17]. Since such cleaup commits do not happen very often, warnings go unsolved for a while.

For reliable results, solve times should come from several different projects for each category, otherwise individual projects will have too much influence on the statistics. Table 5.13 shows the number of projects that contributed to each category, along with the share of the project that contributes most to the category's solve times. The share is the percentage of solve times that originates from a single project for a category; for example, although Metric has solve times of 12 different projects, half these times come from a single project. Ideally, this fraction is as close as possible to $\frac{1}{n}$ where n is the number of projects for the category.

[13]https://github.com/checkstyle/checkstyle/commit/f79ab476b036a185383a1c7f28dc8cc02869e5e3

[14]https://github.com/checkstyle/checkstyle/commit/7450f13dfcf2f222c4bac344f21c5105f7430b61

[15]https://github.com/rembo10/headphones/pull/2020

[16]https://github.com/pyinstaller/pyinstaller/pull/1907/files

[17]https://github.com/OpenGrok/OpenGrok/commit/eb1776903fd1f998009e97470a65fba8a499a0d9

Given these results, we believe more data would be needed to actually support the claim of a significant ordering of categories; especially the shares of the most contributing projects are too high. This results in solve times of several categories mostly being dominated by a single project.

Category	Number of projects	Biggest share
Best Practices	11	31%
Code Structure	3	83%
Concurrency	1	100%
Documentation Conventions	4	74%
Error Handling	4	93%
Interface	6	56%
Logic	9	49%
Metric	12	50%
Migration	1	100%
Naming Conventions	4	81%
Object Oriented Design	2	83%
Redundancies	19	40%
Regular Expressions	1	100%
Resource	8	41%
Simplifications	4	53%
Style Conventions	22	62%
Tool Specific	3	82%

Table 5.13: For each category, the number of projects with solve times for that category, and the highest share of a single project

The skewed solve time data prevents us from making confident statements about differences between categories in general. However, we can look at the solve times per category in the context of individual projects. Our hypothesis is that there is no large difference between solve times per category within a single project, but solve times between projects do differ; i.e. the project has a bigger influence on the rate at which a warning is solved than the category which the warning belongs to. Tables 5.14 and 5.15 shows the median solve times and sample sizes for each category for a select number of projects, as well as average and median for all solve times of the project in the bottom rows. We do not show all projects because the full table is very sparse; for 10 projects we do not have any solve times at all, and 15 projects yielded solve times for only 3 categories or less. The remaining 13 are displayed.

	FreeCodeCamp	OpenGrok	ultisnips	bower	glide	cython	less.js	Chart.js
Best Practices		67.5 (62)	3 (21)	30.5 (12)		336 (18)	1 (3)	
Code Structure			3 (1)			3 (5)		
Concurrency		132 (4)						
Documentation Conventions			3 (109)		3 (22)	372 (1)		
Error Handling		20 (446)	14.5 (2)					
Interface			3 (106)			1 (7)		
Logic	107 (6)	12 (4)	29 (191)			1 (141)	1 (12)	10 (10)
Metric	33.5 (4)	28 (5)	6 (63)	16 (126)		1 (99)		63.5 (2)
Migration								
Naming Conventions			3 (59)			1 (3)		
Object Oriented Design					1 (2)			
Redundancies	3 (13)	22 (46)	3 (25)		1 (3)	82 (23)	2 (1)	53 (1)
Regular Expressions								
Resource				38 (48)		8 (1)	241 (17)	
Simplifications		43 (3)	4 (4)					
Style Conventions	91 (1)	41 (7)	6 (63)	22 (448)	1 (5)	131 (6)	81 (3)	33 (5)
Tool Specific			20 (37)			1 (5)		
Total Average	44.33	47.09	22.51	23.64	22.38	41.58	113.69	22.78
Total Median	12	22	5	22	1	1	111	10

Table 5.14: Median solve times with sample sizes in brackets per category per project, and overall average and median solve times per project, part 1

	pyinstaller	headphones	grape	paperclip	jeromq
Best Practices	169 (44)	50 (29)	50 (6)		33 (2)
Code Structure		3 (29)			
Concurrency					
Documentation Conventions	122 (16)				
Error Handling	5 (1)		47 (32)		
Interface	170 (68)	1 (3)	28 (1)		
Logic	144 (21)	2.5 (2)			
Metric	181 (325)	1 (1)		48 (11)	
Migration				56 (4)	
Naming Conventions	177 (272)				
Object Oriented Design					
Redundancies	162 (131)	3 (23)	28 (3)	28 (4)	13 (1)
Regular Expressions					7 (747)
Resource	70 (2)		17 (23)		
Simplifications			149 (9)		
Style Conventions	321 (13)	4 (25)	8 (3)	56 (19)	23 (24)
Tool Specific	6 (3)				
Total Average	190.27	14.09	60.75	43.24	7.8
Total Median	177	3	44	55.5	7

Table 5.15: Median solve times with sample sizes in brackets per category per project, and overall average and median solve times per project, part 2

For bower, less.js, pyinstaller, paperclip and jeromq, the average and median values are quite close to each other, but only paperclip has category solve times that are somewhat close to each other. To a lesser extent, this also holds for pyinstaller, since its outliers (Error Handling, Resource, Tool Specific) are obtained from small samples. We see no strong evidence that projects have a consistent solve time across categories. In many projects we see large differences between overall average and median (FreeCodeCamp, ultisnips, glide, cython, headphones), hence no real conclusions can be drawn from these tables either.

5.4 Threats to Validity

In this section we discuss the potential threats to validity of our research.

5.4.1 Internal Validity

We compared several properties between the groups of projects that use ASATs and those that do not. However, we cannot really say which was the cause and which was the effect. For example, projects using ASATs have more stars, but that does not mean that using ASATs gets a project more stars, or that a high star count leads a project to use ASATs. Furthermore, all tests were performed on the same set of projects. This reduces the strength of the conclusions drawn from these tests.

5.4.2 External Validity

With millions of repositories hosted on GitHub, a sample size of under 10,000 may seem low. However, these are all the popular and active repositories. We believe that compromising in this regard may result in a larger dataset, but as we found out, ASAT usage is higher among repositories with more stars, and we still ended up with less than 30% ASAT usage overall. By decreasing the required amount of stars, this percentage would go even further down. Including less active repositories would mean we end up getting abandoned projects as well, which may not be representative for open source projects as a whole. A viable solution may be to include more ASATs, specifically those for languages that we did not cover in this study.

100 pull requests per repository may be enough for properties like recent density, but the merged count and unique contributors would probably need more data to be accurate. However, we believe that by averaging these values, the effect of edge cases is reduced.

Nearly twelve million warnings in the database may seem like an impressive number. However, the distribution of warnings over projects is heavily skewed towards a couple of Python projects. This mainly impacts results such as warning counts per category, where these projects simply have a much bigger impact. To counter this, we looked at the results for each language separately.

5.4.3 Construct Validity

Our tool has some shortcomings regarding the measurement of solve times. To elaborate on this, first recall the way we compute solve times, depicted in figure 4.6. Although this approach seems rather straightforward, it does depend on a reliable definition of equality between two warnings (when deciding which counters to increment). As it turns out, this is a nontrivial task. For each warning found by an ASAT, we save its location (filename, line and column), the human-readable message that the ASAT provides, the rule ID, and the line of code where the warning occurred. Intuitively, one might require all of these properties to be equal for two warnings to be equal. However, an unrelated addition in a warning's file above the line of the warning will cause the line number to change, and this would wrongly be picked up as the warning being fixed, with another warning having been introduced in the same commit. Should said warning really be resolved at a later point, we would end up with two incorrect solve times which add up to the real value. The same thing happens when changing indentation size, but then with the column number. This would result in invalid data.

To deal with the issue of the position of the warning, we only check for the filename, rule ID, and the actual line of code. In most cases, this works, but there are still some scenarios where different warnings could be registered as duplicates:

1. **A single line of code resulting in multiple warnings with the same rule ID.** This can happen when there are two instances of a warning within the line, in which the column number would be the tie breaker. However, we also discovered an edge case for this in Pylint: when a wildcard import is used, a warning is reported for each

module that is imported as a result, but not used. In this case, the warning message is the tie breaker as it states the name of the unused module.

2. **Duplicate lines of code.** Some lines of code are rather generic, for example the first line of a catch block: `} catch (Exception e) {`.
PMD has a rule, `AvoidCatchingGenericException`, that checks for catch blocks using the base `Exception` class. However, such lines can occur more than once in a file. Furthermore, we encountered a case with duplicate docblock lines, describing the same parameter in different methods, which caused a warning for being too long. Finally, lines containing just whitespace sometimes caused a warning, for example when the empty line was trailing whitespace of a string literal. As a result, the code line ended up being an empty string, because code lines are trimmed before being saved. In cases of duplicate lines, the line number could be used as tie breaker.

We have taken a couple of measures to counter these issues. For the first one, we add the warning message to the comparison whenever the rule is `unused-wildcard-import`. Two warnings on the same line will still cause incorrect duplicate warnings, however. For the second case, we add the warning line number to the comparison whenever the code line is an empty string, or the warning rule is `AvoidCatchingGenericException`. However, this will cause warnings to be incorrectly marked as solved when their line number changes. So far, we did not find a robust method of dealing with this.

To illustrate the consequences of this issue, we look at some instances of warning solving in `zeromq/jeromq`, which has defined a number of blank line rules as regular expressions. For example, there should be no blank line before a closing brace, represented as `\n\n}`, and no consecutive blank lines, represented as `\n\n\n`. However, it seems like no warnings were actually fixed until the commit that solved everything (see figure 5.8d). Instead, most of these fixes were registered because code was added, changing the line numbers of all violations below the insertion, so they were moved rather than solved. Since many files had warnings in them, almost every commit caused some of those warnings to be "fixed". Therefore, we cannot consider these solve times to be viable, and by extension, the entire Regular Expressions category's solve times, because there were no other instances of this category.

These are probably just a few of the potential issues. Because of that, solve time data may not be accurate. Under the assumption that this phenomenon occurs equally often in different warning categories, the order of categories regarding solve times should still be valid though. To mitigate this issue further, we leave out solve times larger than the total amount of commits for a repository, because these are clearly the result of different warnings being recognised as the same, and as a result, the counter is wrongly incremented multiple times per commit.

Chapter 6

Conclusion

In this chapter, we review the research questions and answer them based on the results that we found in chapter 5. After that, we give an overview of the study's contributions. Finally, we provide some ideas for future work based on this study.

6.1 Answers to Research Questions

RQ1: Which factors influence ASAT prevalence?

To answer this question, we look at the results as described in section 5.1. Here we found that projects that have JavaScript as main language have the highest ASAT adoption rate by a wide margin(RQ1.1), and JavaScript-related ASATs (JSHint, ESLint, JSCS) are most prevalent(RQ1.2). Ruby comes second, with Java and Python lagging relatively far behind. When using a build tool, JavaScript projects have configured an ASAT as build task in 62% of the cases, compared to an overall use rate of 48%. This was not observed in other languages, which are less likely to use ASATs as build tasks than they are to use ASATs in general (RQ1.3).

When using Travis, projects seem more likely to use ASATs as well(RQ1.4). Repositories with ASATs are younger on average, but this may well be due to the bias toward JavaScript projects(RQ1.5). Finally, projects with higher star counts seem more likely to use ASATs(RQ1.6).

RQ2: How does community activity affect ASAT usage?

When looking at the pull request data, we see that projects with ASATs have a higher number of total pull requests on average, and also received more pull requests over time(RQ2.1). We did not find a significant relation between ASAT usage and unique contributor count(RQ2.2), but the contributor count did significantly affect both the time to close a pull request and the likelihood for a pull request to be accepted, with a positive and negative correlation respectively. ASAT usage does not increase the chance for a pull request to be accepted(RQ2.3), but it does reduce time to close, with close times reduced by 17% on average and lowering the variance of close time by 67%(RQ2.4).

RQ3: What is the prevalence of rule violations reported by ASATs?

We attempted to answer this question by running ASATs on a number of projects. For several projects, we looked at the amount of warnings their code generated over up to 500 versions. For most projects, this amount decreased over time or roughly stayed at the same level. In some cases, the warning count increased. No universal trends were discovered, but we found that the reason for a drop in warnings is often a change in configuration or moving/deleting part of the code base, rather than fixing violations directly(RQ3.1). In general, projects with more lines of code had fewer warnings per line of code than smaller projects(RQ3.2). The most warnings overall were found in Python projects.

When looking at warning categories, we found that the counts per category significantly differ between languages, but Style Conventions is highly represented across all languages, and many of the warnings that caused other categories to have high counts could also be considered Style Conventions, but fitted another category description slightly better(RQ3.3). We also found a distinct ordering of categories with respect to the time it takes to solve warnings on average by means of statistic tests; warnings of the categories Naming Conventions and Best Practices, which do not play a big role in maintainability, took the longest time to solve, while Code Structure and Object Oriented Design were solved the fastest, possibly because they do have an impact on maintainability(RQ3.4). However, data on solve times is heavily influenced by individual projects. Therefore, a bigger dataset would be desirable, or one that is more evenly distributed over different projects.

6.2 Contributions

By performing this study, we have made the following contributions to the research on ASATs in general:

- **A look at characteristics associated with ASAT usage.** By comparing several properties groups of projects with and without ASATs, we have found some statistically significant differences.

- **A link between ASAT and CI usage.** We found that ASATs and CI, specifically Travis, often go together in open source projects. We also found that CI usage can greatly reduce the amount of warnings in committed code.

- **The effects of ASATs on community activity.** We found that ASAT usage is associated with several favorable changes in community activity. However, we did not find which of these is the cause and which the effect.

- **The prevalence and solve rate of warning categories.** We looked at both the warning counts and solve times per category, and established an ordering backed by statistical tests.

6.3 Future work

We have encountered three ways for a warning to be removed:

1. The corresponding line of code is changed to comply with the corresponding rule
2. The configuration for the corresponding rule is changed, or the configuration is changed to exclude the rule altogether
3. The corresponding code is moved or deleted

Because we want to learn about solve rates, we are mostly interested in the first way. In addition, it would also be interesting to find out how often the second way occurs. This could be linked to previous research by Beller et al about changes in ASAT configurations[8]. Ideally, we should be able to detect the third case happening and act accordingly: for moved code, find the new location, and keep counting commits from there, and for deleted code, mark the corresponding warning as a separate case, rather than "solved after *x* commits", which happens now. However, our current tool cannot distinguish between these cases.

As mentioned, we had some issues properly detecting generic warnings (such as empty lines) that were not resolved, in a commit where the file of the warning was modified. In such cases, the warning was incorrectly marked as solved and a new counter was started for the same warning. This may be approached using Clone Region Descriptors[14], a technique to find code duplication, because the issue arises in non-unique lines of code. Using CRD, one could keep track of all "cloned" regions, find the ones where warnings occurred, and mark each warning with a unique identifier based on the CRD findings.

Individual projects often dominated the solve time data. Instead of merging all solve times together, a way to normalize results per project would be to average the solve time of each category per project. This way, each project only contributes up to a single solve time per category, reducing the bias introduced by projects that perform lots of violation solving. A downside to this is the significantly reduced size of the resulting dataset. To counter this, ASATs would need to be run on a much larger number of projects.

Bibliography

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: an effective verification process. *IEEE software*, 6(3):31, 1989.

[2] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.

[3] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[5] Thomas Ball and Sriram K Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[6] Victor R Basili and Richard W Selby. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12):1278–1296, 1987.

[7] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.

[8] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 470–481, 2015.

[9] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[10] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. *PeerJ Preprints*, 2016.

[11] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294, 1989.

[12] Andrea Capiluppi, Maurizio Morisio, and Juan F Ramil. Structural evolution of an open source system: A case study. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 172–182. IEEE, 2004.

[13] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[14] Ekwa Duala-Ekoko and Martin P Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1):3, 2010.

[15] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[16] Eirini Kalliamvakou et al. The promises and perils of mining github. *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014.

[17] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[18] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, page 122, 2006.

[19] Georgios Gousios and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. *IEEE Software*, 32(1), 2015.

[20] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[21] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press, 2015.

[22] Seth Hallem, David Park, and Dawson Engler. Uprooting software defects at the source. *Queue*, 1(8):64–71, 2003.

[23] Jesper Holck and Niels Jørgensen. Continuous integration and quality assurance: A case study of two open source projects. *Australasian Journal of Information Systems*, 11(1), 2003.

[24] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[25] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[26] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[27] Stephen C. Johnson. *Lint, a C program checker*. Citeseer, 1977.

[28] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[29] Sunghun Kim and Michael D Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[30] John C Knight and E Ann Myers. An improved inspection technique. *Communications of the ACM*, 36(11):50–61, 1993.

[31] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*, pages 295–315. Springer, 2003.

[32] Johnny Martin and Wei Tek Tsai. N-fold inspection: A requirements analysis technique. *Communications of the ACM*, 33(2):225–232, 1990.

[33] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[34] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[35] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[36] Allen P Nikora and John C Munson. Understanding the nature of software evolution. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 83–93. IEEE, 2003.

[37] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170. IEEE, 2015.

[38] David L Parnas and David M Weiss. Active design reviews: principles and practices. In *Proceedings of the 8th international conference on Software engineering*, pages 132–136. IEEE Computer Society Press, 1985.

[39] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 411–416. IEEE, 2012.

[40] Adam Porter, Harvey Siy, and Lawrence Votta. A review of software inspections. *Advances in Computers*, 42:39–76, 1996.

[41] Peter C Rigby and Daniel M German. A preliminary examination of code review processes in open source projects. Technical report, Technical Report DCS-305-IR, University of Victoria, 2006.

[42] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.

[43] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 8. ACM, 2010.

[44] Daniela Steidl. *Cost-Effective Quality Assurance For Long-Lived Software Using Automated Static Analysis*. PhD thesis, München, Technische Universität München, Diss., 2016, 2016.

[45] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and software*, 81(5):823–839, 2008.

[46] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from github.** updated version with corrections. *arXiv preprint arXiv:1512.01862*, 2015.

[47] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[48] Lawrence G Votta Jr. Does every inspection need a meeting? *ACM SIGSOFT Software Engineering Notes*, 18(5):107–114, 1993.

[49] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229. IEEE, 2008.

[50] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering*, 32(4):240–253, 2006.