2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice

Advantages and Disadvantages of a Monolithic Repository

A case study at Google

Ciera Jaspan, Matthew Jorde,
Andrea Knight, Caitlin Sadowski,
Edward K. Smith, Collin Winter
Google
ciera,majorde,aknight,supertri,edwardsmith,
collinwinter@google.com

Emerson Murphy-Hill*
NC State University
emerson@csc.ncsu.edu

ABSTRACT

Monolithic source code repositories (repos) are used by several large tech companies, but little is known about their advantages or disadvantages compared to multiple per-project repos. This paper investigates the relative tradeoffs by utilizing a mixed-methods approach. Our primary contribution is a survey of engineers who have experience with both monolithic repos and multiple, per-project repos. This paper also backs up the claims made by these engineers with a large-scale analysis of developer tool logs. Our study finds that the visibility of the codebase is a significant advantage of a monolithic repo: it enables engineers to discover APIs to reuse, find examples for using an API, and automatically have dependent code updated as an API migrates to a new version. Engineers also appreciate the centralization of dependency management in the repo. In contrast, multiple-repository (multi-repo) systems afford engineers more flexibility to select their own toolchains and provide significant access control and stability benefits. In both cases, the related tooling is also a significant factor; engineers favor particular tools and are drawn to repo management systems that support their desired toolchain.

CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**;

1 INTRODUCTION

Companies today are producing more source code than ever before. Given the increasingly large codebases involved, it is worth examining the software engineering experience provided by the various approaches for source code management. Large companies with multiple products typically have many internal libraries and frameworks, and a vast number of dependencies between projects from entirely separate parts of the organization. Successfully organizing these dependencies and frameworks is crucial for development velocity.

One approach to scaling development practices is the monolithic repo, a model of source code organization where engineers have broad access to source code, a shared set of tooling, and a single set of common dependencies. This standardization and level of access is enabled by having a single, shared repo that stores the source code for all the projects in an organization. Several large software companies have already moved to this organizational model, including Facebook, Google, and Microsoft [10, 12, 17, 21]; however, there is little research addressing the possible advantages or disadvantages of such a model. Does broad access to source code let software engineers better understand APIs and libraries, or overwhelm engineers with use cases that aren't theirs? Do projects benefit from shared dependency versioning, or would engineers prefer more stability for their dependencies? How often do engineers take advantage of the workflows that monolithic repos enable? Do engineers prefer having consistent, shared toolchains or the flexibility of selecting a toolchain for their project?

In this paper, we investigate the experience of engineers working within a monolithic repo and the tradeoffs between using a monolithic repo and a multi-repo codebase. Specifically, this paper seeks to answer two research questions:

(1) What do developers perceive as the benefits and drawbacks to working in a monolithic versus multi-repo environment?
(2) To what extent do developers make use of the unique advantages that monolithic repos provide?

To answer these questions, we ran a mixed-methods case study within a single company with a monolithic repo. We surveyed software engineers to understand their perceptions about working in monolithic repos. For engineers that also had experience working in multi-repo systems, we asked further questions to understand the benefits of each and why they might prefer one model over another. We also analyzed the logs from developer tools to study the extent to which engineers utilize their ability to view and edit all of the code in the codebase. We examined how often engineers view and edit code far afield from their team and organization, and we examined whether these views are simply to popular APIs.

Our survey results show that engineers at Google strongly

*Work completed while on sabbatical at Google

ICSE-SEIP '18, Gothenburg, Sweden
 978-1-4503-5659-6/18/05
DOI: 10.1145/3183519.3183550

prefer our monolithic repo, and that visibility of the codebase and simple dependency management were the primary factors for this preference. Engineers also cite as important the ability to find example uses of API and the ability to automatically receive API updates. Logs data confirms that engineers do take advantage of both the visibility of the codebase and the ability to edit code from other teams. Contrary to expectations, viewing popular APIs was not the primary reason engineers view code outside of their team; this provides further [18] evidence that viewing code to find examples of using an API is important, possibly more so than viewing the implementation of the API.

We also discovered many interesting tradeoffs between using monolithic and multi-repo codebases; they each had benefits that were not possible in the other system. One such tradeoff was around dependencies. Engineers note that a primary benefit of multi-repo codebases is the ability to maintain stable, versioned dependencies. This is particularly interesting because it is in direct contrast to two of the primary benefits of a monolithic repo: ease of both dependency management and of receiving API updates. Another tradeoff appeared around flexibility of the toolchain. Engineers who prefer multiple repos also prefer the freedom and flexibility to select their own toolchain. Interestingly, the forced consistency of a monolithic repo was also cited as a benefit of monolithic repos.

Finally, we saw evidence that for some engineers, the development tools were more important than the style of repo. Engineers called out favored development tools by name as a reason to use one repo over another, even though in theory, these tools could be available for any type of repo.

2 MONOLITHIC REPOSITORIES

For purposes of this paper, we define a monolithic source repo to have several properties:

(1) **Centralization:** The codebase is contained in a single repo encompassing multiple projects.
(2) **Visibility:** Code is viewable and searchable by all engineers in the organization.
(3) **Synchronization:** The development process is trunk-based; engineers commit to the head of the repo.
(4) **Completeness:** Any project in the repo can be built only from dependencies also checked into the repo. Dependencies are unversioned; projects must use whatever version of their dependency is at the repo head.
(5) **Standardization:** A shared set of tooling governs how engineers interact with the code, including building, testing, browsing, and reviewing code.

This definition is consistent with [21] and [29]. [1]

At the other extreme, a multi-repo system is one where code is separated by project. Notice that in a multi-repo system, it may still be true that code is viewable by all engineers, as is the case for open-source projects on GitHub or BitBucket. In theory, a multi-repo system could also have a shared set of developer tools; in practice this is rare as there is no enforcement for this to happen across repos. In a multi-repo setup, commits are not to a single head, so version skew and diamond dependencies (where each project may depend on a different version of a library) do occur.

At Google, almost all code exists in a single large, central repo, in which almost all code[2] is visible to almost all engineers. The repo is used by over 20,000 engineers and contains over 2 billion lines of code. All engineers that work in this monolithic repo use a shared set of tools, including a single build system, common testing infrastructure, a single code browsing tool, a single code review tool, and a custom source control system. The build system depends on compilers that are also checked into the codebase; this allows a centralized tooling team to update the compiler version across the company.

While engineers can view and edit nearly the entire codebase, all code is committed only after the approval of a code owner. Code ownership is path-based, and directory owners implicitly own all subdirectories as well. Engineers are limited to using a small set of programming languages, and there is a tool-enforced style for each language.

3 METHODOLOGY

To understand how engineers perceive the advantages and disadvantages of a monolithic repo, we surveyed a sample of engineers at Google. Rather than only measuring engineer satisfaction with the monolithic repo, we asked them to compare Google's monolithic repo to their prior experiences with other repo systems and to one hypothetical example. The goal was to identify the relative tradeoffs between a monolithic repo and multi-repos.

We also took advantage of our ability to log engineers' interactions with the codebase. Our common developer tools allow us to instrument not only commits to the codebase, but also file views. We used these logs to confirm some of the survey responses by showing that engineers not only say they take advantage of the visibility they get from a monolithic repo, but actively utilize this benefit.

3.1 Assumptions

We had several assumptions based on prior internal surveys and interviews about developer tools.

First, we expected our developer tools to be a major contributor to why engineers prefer our codebase. The internal tools regularly receive exceptionally high satisfaction ratings in surveys and interviews. This is a potential source of bias if we ask about satisfaction with our codebase, and we wished to separate satisfaction with the tools from satisfaction with the general concept of a monolithic repo. To mitigate this, we asked a question in the survey that attempts to hold the developer tooling stable for a comparison.

[1] Notice that there is a difference between a monolithic repo and a monolithic architecture. Linux is an example of a monolithic architecture, but it is not an example of a monolithic repo. Google's codebase is the opposite; it is a monolithic codebase but not a monolithic architecture.

[2] The primary exceptions are Chrome and Android.

Second, we expected visibility to be highly important to engineers. Engineers anecdotally cite it as a major factor for development velocity. However, we were uncertain whether engineers actually take advantage of this power. Do engineers say that visibility is important because they like the idea of being able to view code in another project, or do they actually utilize this ability on a regular basis? Because of this assumption, we planned our logs analysis to investigate this further and mitigate this risk. Additionally, in our survey, we compare monolithic repos against open-source repos. Open-source repos like GitHub are also multi-repo codebases, but with full visibility, and so provide for a useful point of reference beyond visibility benefits.

Finally, we expected complexity to be a theme. Prior internal surveys had shown that the size and complexity of the codebase overwhelms engineers. However, we were not sure how this complaint stacks up against potential benefits.

3.2 Survey Methodology

We randomly selected 1902 engineers who had worked at Google for at least three months, who had committed code to our monolithic repo in the six months prior, and who had averaged at least five hours a week in our developer tools. The population for this sample was 23,000 software engineers at Google. We constructed our survey invitation to maximize survey responses using existing best practices from the SE research community [26]. Engineers who had not completed their survey in the first 24 hours received an email reminder. None of the questions were required to complete the survey. Responses were confidential to the authors, but not anonymous. We also provided no incentives for survey completion. Of the 1902 engineers in the sample, 869 completed the survey, yielding a response rate of 46%.

Table 1 lists the survey questions, which were presented in three blocks. The first block was shown to all participants. It asks about the engineer's overall satisfaction with our monolithic codebase, their beliefs about how it impacts velocity and quality[3], and their past experience with other codebases. The second block of questions was only asked if the participant indicated that they had commercial experience with a multi-repo codebase, and the third block was only asked if the participant indicated experience in working on an open-source project.

The survey utilized several free-response questions to capture each participant's points of comparison and their motivation(s) for preferences. We employed an open coding methodology to categorize responses. Responses could receive multiple tags; the full list we used is described in [14]. We did a pass for common responses and tagged them (e.g., responses consisting entirely of the name of the internal code browsing tool were tagged "visibility" and "developer tools"). We tagged the remainder collaboratively with three authors. We resolved disagreements by re-reading the response and coming to a shared agreement. In some cases, we split tags and retagged responses to tease apart emergent themes. Finally, we did keyword searches to verify tagged responses. We used the 21 tags to create frequency graphs and narratives around five emergent themes (Section 5).

[3]The survey does not define the terms "velocity" or "code quality", but these are commonly used terms within Google and engineers have developed shared meaning around them.

3.3 Logs analysis methodology

For our logs analysis, we sought to understand the extent to which engineers view and commit files outside of their project. As a project might be defined in different ways, we chose instead to focus at the level of a Product Area (PA). As there are only 12 PAs at the company, any view or edits that are to the code of a different PA are highly likely to be outside of an engineer's project. This provides us with a lower bound for the amount of cross-project views and commits.

We analyzed two types of logs:

- **Code browsing logs.** Engineers at the company use a web-based tool to browse code. The code browsing tool logs every time an engineer views a file. While engineers can still browse code within their editor, this practice is less common, as local editors have understandable difficulty indexing and searching cross-references across a repo of this size.
- **Code commit logs.** This is simply the code that was committed into the codebase by an engineer.

To investigate the percentage of actions (views and commits) on code outside of an engineer's PA, we needed a mapping from engineer to PA, and from source file to PA. Each of the approximately 28,000 engineers[4] is assigned to one of 12 PAs for every week in our one-year study period.

To map from source file to PA, we used project metadata files. These files specify, for each project, the code directories they own and the PA the project belongs to. Code directories are not uniquely owned by a PA, and some directories are not owned by any PA. In the case that multiple projects claim ownership, we selected the majority PA. 53% of code directories were assigned a PA; 47% were unassigned either due no ownership or a tie for majority ownership.

For the remaining 47% with no clear owning PA, we looked at reviewers for commits in the directory. All code within our company must be reviewed by an engineer who owns that code, so reviewers give a good approximation of code ownership. For each source code directory, we compiled the list of reviewers who approved changes that were committed to that directory. We then looked up the PA for those reviewers and selected the majority PA of the reviewers for that directory. Majority PA is chosen because code often has non-owner reviewers from other PAs (e.g., subject-matter experts, language approvers, etc.) From this, we were able to assign PAs to 93% of directories.

The remaining 7% are directories where the code has no assigned project metadata and also has not been reviewed

[4]We only considered full-time employees with job categories that signal that software engineering is their primary task. This includes software engineers and related job categories, but not job categories such as managers, UI designers, or quantitative analysts.

Num	Text	Response type
Q1.1	Rate your satisfaction with Google's codebase as a software engineer.	7 point scale, "Extremely satisfied" to "Extremely dissatisfied"
Q1.2	Please rate how important the following are to your velocity as a developer. • I can edit source code from almost any project at Google • I can search almost all of Google's source code	5 point scale, "Extremely important" to "Not at all important"
Q1.3	Please rate how important the following are to your code quality as a developer. • I can edit source code from almost any project at Google • I can search almost all of Google's source code	5 point scale, "Extremely important" to "Not at all important"
Q1.4	Tell us more about your background: Which of the following scenarios have you experienced as a software engineer? Select all that apply, they do not need to relate to your Google employment.	Multi-select • Collaborating as an individual on open-source projects • Working at a small company with fewer than 5 software engineers • Working at a startup or a new project where you started the codebase from scratch • Working at a company with lots of engineers, who have multiple code repos • Working at a company other than Google, who has a large monolithic codebase
Q2.1	Think back to your experience with the last multi-repo codebase you used. Rate your satisfaction with that codebase as a software engineer.	7 point scale, "Extremely satisfied" to "Extremely dissatisfied"
Q2.2	Comparing your experience with the most recent multi-repo codebase you used to working in Google's codebase, which codebase did you prefer?	7 point scale, "Strongly prefer multi-repo codebase" to "Strongly prefer Google's codebase "
Q2.3	Why? Describe what motivates your preference.	Free response
Q2.4	What are some of the benefits you found to working in a multi-repo codebase?	Free response
Q2.5	What are some of the benefits you found to working in Google's single-repo codebase?	Free response
Q2.6	If you could choose to work with Google's codebase as a monolithic repo or in multiple smaller repos, which would you choose?	Single select • I would prefer to work with Google's codebase in a single repo • I would prefer to work with Google's codebase in multiple smaller repos • No preference
Q2.7	What is the single main reason you would choose to use Google's codebase (as a single repo/in multiple smaller repos)?	Free response
Q3.1	Comparing your experience with your most recent open-source project to working in Google's codebase, which codebase did you prefer?	Single select • My open-source project codebase • Google's codebase • Neither
Q3.2	What are some of the benefits you found to working in open-source codebases?	Free response
Q3.3	What are some of the benefits you found to working in Google's codebase?	Free response

Table 1: Survey questions

(and thus not modified) in over a year. We excluded these code directories from our analysis.

To analyze the extent of views/commits that occur cross-PA, we calculated the percentage of an engineer's interactions with files in a different PA for each week in 2016. We then averaged each engineer's percentage across all weeks. Finally, we computed distribution statistics across all engineers.

3.4 Threats to Validity

Our survey may suffer from selection bias; software engineers' codebase preferences may impact where they work, such that those who prefer multi-repo codebases may choose to work at a company with such a codebase. Other companies could rerun our survey to determine whether their workforce prefers their respective codebase model. Our survey may also suffer from nonresponse bias, though we acheived a good response rate of 46%.

Our survey results may have a major confounding factor from Google's internal developer tools. Engineers may have rated the monolithic codebase highly when, in fact, it was the developer tools that they had a strong preference for. Indeed, the developer tools did come up as a major benefit, though certainly not the only one. To mitigate this confound, Q2.6 and Q2.7 ask about whether particpants had a preference for Google's codebase in a monolithic repo or a (hypothetical) multi-repo codebase where tooling could be considered to be equal. This appears to be a successful mitigation; fewer respondents cited developer tools in these questions compared to Q2.2 and Q2.3.

Our survey results may also suffer from priming. In particular, Q1.2 and Q1.3 ask the participant to think about the relationship between their ability to see and edit the entire codebase with velocity and code quality. It is likely that this primed participants later to think about the visibility of a codebase, velocity, and code quality when considering potential benefits to different types of code repos.

We used an open-coding methodology to classify developer survey responses into thematic areas. This process is inherently subjective. We used a collaborative open-coding process with three coders to ensure no single coder had unmitigated influence over the coding. We do not claim this to be a complete or singular way of coding this data.

The primary threat to validity for the quantitative logs analysis is the heuristic for assigning source code to a PA. 53% of the code directories were mapped to a PA using metadata files. These metadata files may be inaccurate if a project moves from one PA to another PA, though this is rare. If this happens, we would have swapped whether the project was within an engineer's PA or outside of it. We also could not calculate a PA for the 7% of directories which had no metadata file and no commits in the last year. However, it is likely that these are dead projects. Finally, we assign only a single PA to each directory. 3.9% of directories are claimed by multiple PAs. 13.6% of users had at least one interaction matching a file's minority PA (a *potential* false positive). However, the impact was small; only 6 engineers had 1% or more of their interactions affected.

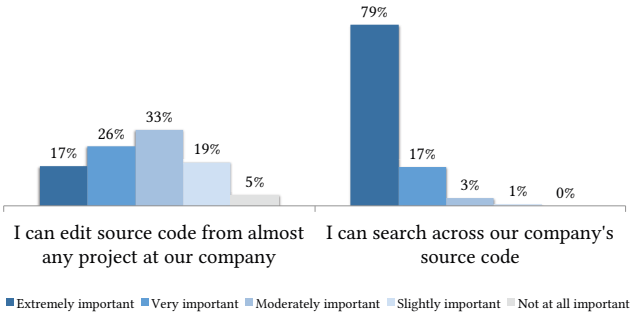

Figure 1: Impact on velocity. (Q1.2)

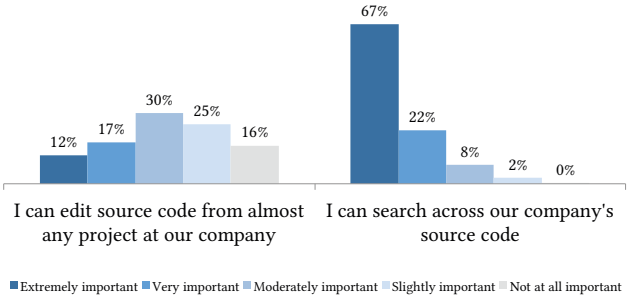

Figure 2: Impact on code quality. (Q1.3)

4 RESULTS

Of the 869 engineers who completed the survey:

- 455 (52%) had prior experience where they started a codebase from scratch.
- 379 (44%) had prior experience in a corporate multi-repo codebase.
- 337 (39%) had prior experience using open-source codebases.
- 321 (37%) had prior experience working at a company with fewer than 5 software engineers.
- 205 (24%) had prior experience using a monolithic codebase at a different company.

Survey questions Q1.2 and Q1.3 asked all 869 engineers to evaluate how their ability to search/edit code impacted their velocity and code quality. Figures 1 and 2 show the results for these questions. Participants overwhelmingly reported that the ability to search code is important to both velocity and code quality. Participants had mixed opinions on whether the ability to edit code across the codebase is important to velocity and code quality.

The logs analysis confirmed that engineers regularly view code outside of their PA. The first two rows of Table 2 show

Interaction	1st%	5th%	10th%	20th%	Median	80th%	90th%	95th%	99th%	Exclusions
Code view	1%	3%	6%	11%	28%	54%	69%	82%	94%	None
Code commit	0%	0%	0%	0%	5%	32%	60%	79%	100%	
Code view	0%	2%	3%	6%	18%	44%	61%	75%	92%	Common files
Code commit	0%	0%	0%	0%	3%	26%	54%	74%	100%	
Code view	1%	2%	3%	6%	17%	41%	57%	71%	89%	Common files and low activity engineers
Code commit	0%	0%	0%	0%	4%	25%	51%	70%	92%	

Table 2: Percentage of cross-PA interactions for engineers.

the percentage of cross-PA interactions for engineers. For example:

- The median code view value of 28% indicates that for 50% of engineers, over 28% of file views are outside of the author's PA.
- The 90th percentile code commit value of 60% means that for 10% of engineers, over 60% of their commits are outside of their PA.

As engineers at our company, we were surprised by how much cross-PA activity actually occurs in practice. We hypothesized that many of the code views may be coming from engineers viewing the APIs of common libraries, including core libraries like collections, distributed databases, and distributed computing frameworks [5, 6, 11]. To account for this, we compiled a set of files to be excluded from analysis due to their common nature:

- A hand-curated list of 26 very common directory prefixes (including projects like Guava [11])
- All directories that had more than 10,000 cross-PA code views in 2016. This constitutes 63,500 source code directories out of 1,817,000 (3.5%).
- All build configuration files, as these may be looked at to see what code is available for reuse.
- All interface files for service-level APIs [22].

The second two rows of Table 2 exclude the files described above from the analysis. Despite these exclusions, there is only a minor change in the distributions. This signals that most of the cross-PA views are not for common libraries.

We also examined the people who only commited code outside of their PA. Most of these came from engineers who had not contributed much code at all. The last two rows of Table 2 exclude both common files and engineers who authored fewer than 20 commits in 2016.[5] This had the effect of removing outliers at both ends of the distribution.

[5]Why 20? We examined the number of commits for our 1902 surveyed engineers, all of whom had averaged at least 5 hours a week working within our developer tools and had submitted code in the prior 6 months. We found that all but 4 of the 1902 surveyed engineers had more than 20 CLs.

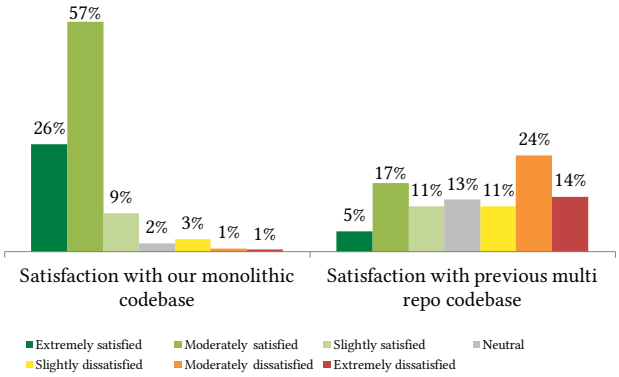

Figure 3: Relative satisfaction ratings of engineers with commercial multi-repo experience. (Q1.1 and Q2.1)

4.1 Participants with corporate multi-repo codebase experience

379 of the 869 participants reported experience working at a company that used multiple repos. Q2.1 asked these participants to rate their satisfaction with the latest such codebase, and Q1.1 asked participants for their satisfaction with Google's codebase. Figure 3 compares the satisfaction rates for these 379 participants on these two questions. While satisfaction with Google's codebase is high overall, satisfaction with multi-repo codebases is mixed.

The survey also explicitly asked these 379 participants which codebase they prefer (Q2.2) and why (Q2.3). 326 particpants preferred Google's codebase, 22 prefered their most recent monolithic codebase, and 31 had no preference. The authors open-coded the responses for the 232 participants who provided a reason for their preference, which resulted in 544 codes (some responses received multiple codes). Figure 4 shows the counts of reasons for their preference, split by which repo they preferred. The list of codes is in [14], and relevant codes are discussed further in Section 5. The top reasons for preferring Google's codebase centered around code reuse, including the ability to see all the code and the ease of dependency management. Since there were only 22 participants who preferred their most recent multi-repo

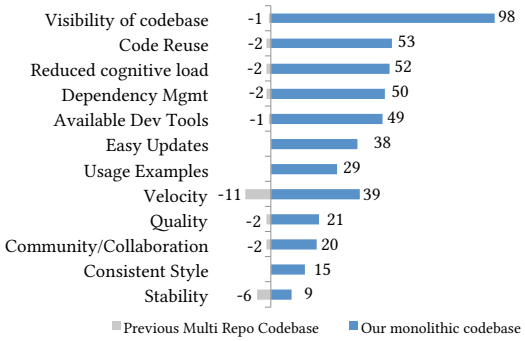

Figure 4: Reasons for preferring Google's monolithic repo or most recent multi-repo codebase, as stated by participants with commercial multi-repo experience. (Q2.2 and Q2.3)

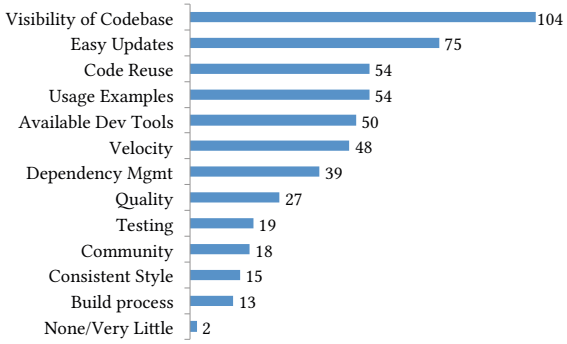

Figure 5: Benefits to Google's monolithic repo, according to participants with commercial multi-repo experience. (Q2.4

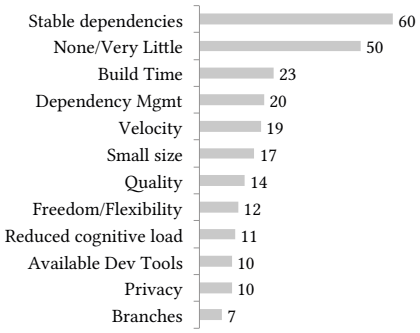

Figure 6: Benefits to multi-repo codebases, according to participants with commercial multi-repo experience. (Q2.5)

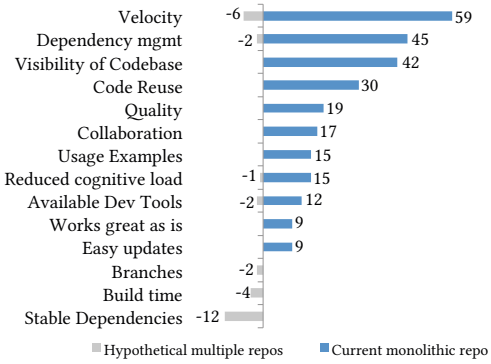

Figure 7: Reasons for preferring Google's codebase either as a monolithic repo or as a hypothetical multi-repo codebase, as stated by engineers with multi-repo experience. (Q2.6 and Q2.7)

experience, it is hard to infer many patterns. The primary reason provided was velocity; interestingly, this was also a common reason for preferring the monolithic repo.

Regardless of their preferred codebase, all 379 participants were asked to provide benefits to working in both Google's codebase (Q2.4) and their prior multi-repo codebase (Q2.5). 238 participants provided 579 benefits to using Google's codebase (Figure 5), and 227 participants provided 298 benefits to using their past multi-repo codebase (Figure 6). If a participant explicitly provided no benefit (by either stating this outright, using profane language, or providing obviously sarcastic remarks), we coded it as "None/Very little".

The 379 participants with experience using a corporate multi-repo codebase were also asked whether they would prefer to work in Google's codebase using either the current monolothic repo or a multi-repo system (Q2.6) and why (Q2.7). 294 participants chose the current monolithic system, 26 would prefer it to be a multi-repo system, and 52 had no preference. 234 participants provided a reason for their choice which resulted in 325 codes, as shown in Figure 7. The primary reasons for preferring a multi-repo codebase were for stability of dependencies and velocity. Velocity was also the primary reason for preferring the monolithic repo, and was followed by ease of depedency management and codebase visibility. Despite attempting to control for the developer tools, we did still see some participants cite developer tools as a reason to prefer one over the other. These participants emphasized the uniformity and consistency of the tools as important to their velocity and cognitive load. Engineers cited build time as a reason they would prefer our company's codebase in a multi-repo system, which is consistent with build times being a benefit to open-source and multi-repo systems. Build speed can be slow within a large monolithic repo because building a project requires building its dependencies. If the dependencies are versioned and available as binaries, as is the case for most multi-repo systems, build speeds improve significantly.

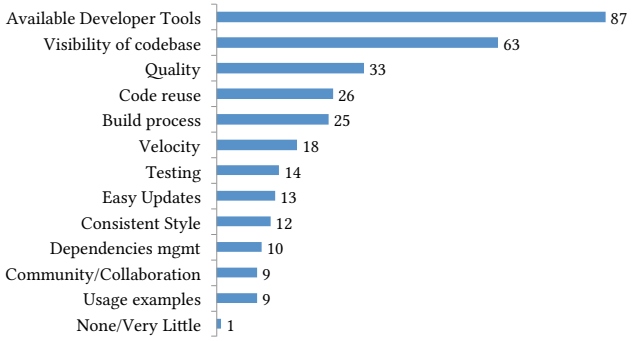

Figure 8: Benefits to Google's monolithic repo, according to participants with open-source experience. (Q3.2)

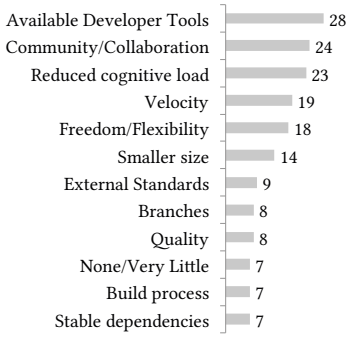

Figure 9: Benefits of open-source codebases, according to participants with open-source experience. (Q3.3)

4.2 Participants with open-source experience

337 of the participants had experience with open-source repos. These repos are of particular interest to us because they, like the monolithic repo, allow for full visiblity into the codebase. We first asked participants whether they preferred the codebase of their most recent open-source codebase, or Google's codebase (Q3.1). 192 participants preferred Google's codebase, 33 preferred their most recent open-source codebase, and 40 had no preference.[6]

Regardless of their preference, these 337 participants were asked to describe the relative benefits of each (Q3.2 and Q3.3). 149 participants provided 379 benefits for using Google's codebase (Figure 8) and 129 participants provided 181 benefits for using open-source codebases (Figure 9). As expected, the visibility of the codebase took a lesser role, though we were surprised that it still came up as a benefit for Google's codebase in this comparison. We notice that code reuse, easy updates, and usage examples all appeared within the stated benefits for monolithic repos over open-source repos. We hypothesize that either the open-source codebases are not large enough to make these benefits apparent, or that the tooling is not sufficient to support these benefits. Developer tools was the primary benefit for both, indicating perhaps that either both provide excellent tooling or that engineers are strongly split about tools. The second-most cited benefit for the open-source repo was being a member of the larger community. Participants enjoyed giving back to others and having their work visible to other people.

5 DISCUSSION

The results in this study highlight five themes about the advantages and disadvantages of a monolithic repo when compared to multi-repo codebases.

[6]This question is flawed in that it assumes these cannot be the same thing. Several open-source projects at Google develop code in the monolithic repo and then export to open-source repos.

The visibility of a monolithic repo is highly impactful. Engineers report that the ability to easily search code has a large positive impact on both velocity and code quality. Our quantitative logs analysis confirms that engineers do view code from across the organization. The logs analysis also showed that these views are not limited to common libraries; the most common libraries made up only 3.5% of the cross-organization page views for 50% of engineers.

Visibility was the top benefit of Google's monolithic repo for engineers with prior experience using multi-repo codebases (Figure 5). Furthermore, it was also the primary driver behind the next three benefits, each of which are enabled through a visible codebase: looking up the documentation and implementations of APIs (coded as Code Reuse), searching for examples of how to use APIs (coded as Usage Examples), and migrating clients of an API to the latest version (coded as Easy Updates).

The visibility of the monolithic repo also enables engineers to edit code across the organization, although this happens less frequently and is rated as less important to velocity and quality than searching for code. This is primarily important for the use case of migrating clients of an API, a task which is undertaken only by the owners of an API.

Visibility also came up in the context of velocity. While engineers cited velocity as a benefit of both the monolithic codebase and their prior multi-repo codebase, it was cited for different reasons. When providing benefits for the monolithic repo, engineers tied velocity back to visiblity and finding example code for APIs. When providing benefits for multi-repo codebases, engineers tied velocity back to smaller code sizes, build speed, and not being limited in their tool choice.

Developer tools may be as important as the type of repo. Engineers commonly cited developer tools as a benefit to both our monolithic repo and to open-source repos. Two internal tools were frequently called out by name; our code browsing tool had 42 mentions in reponses to Q3.2, and the code reviewing tool had 14. Likewise, Git was mentioned 16 times by name in the responses to Q3.3. This signals that

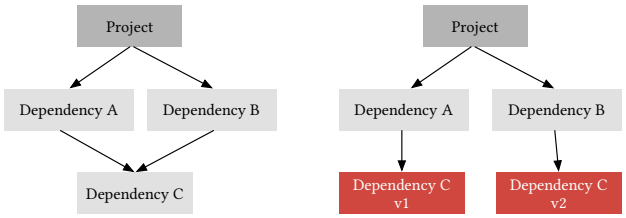

Figure 10: Diamond dependences occur when a project has two dependencies which depend on the same underlying library. When a developer upgrades a dependency, they run the risk of breaking a diamond in the dependency graph.

engineers tend to be very tied to their preferred tools. It also highlights a recognized tradeoff of Google's monolithic codebase: we have developed specific tools to maintain our monolithic repo, but in doing so have lost the ability to use industry-standard tools that engineers are familiar with.

Developer tools also appear to play a significant role when comparing commercial multi-repo codebases. In the results for Q2.1, we saw that 88% of engineers preferred Google's monolithic repo over their past multi-repo setup, and developer tools was highlighted as one benefit of Google's codebase. In the results for Q2.6, we held the tooling the same, and the preference for the monolithic repo went down to 79%. While visibilty is extremely important, there are some engineers who have strong preferences for their development tools.

There is a tension between consistent style and tool use with freedom and flexibility of the toolchain. One of the provided benefits of the monolithic repo is a consistent style; many engineers also associated this with higher-quality code. Likewise, engineers reported that having shared tooling increased their velocity. However, the benefits for open-source projects included the freedom and flexibility to select their own tools, styles, and programming languages. Engineers felt empowered to decide based on the best fit their project.

There is a tension between having all dependencies at the latest version and having versioned dependencies. Two of the primary benefits cited for a monolithic repo were the ease of dependency management (coded as Dependency Mgmt) and the ease of updating dependent code (coded as Easy Updates). Google engineers stated how much they loved not having to deal with the diamond dependency problem [9] (see Figure 10), and they praised the ability to receive code updates as their dependencies migrated. These two benefits were found by engineers with either multi-repo experience (Figure 5) or open-source experience (Figure 8), though it was more frequently called out by engineers with multi-repo experience. Both of these benefits are enabled by building all code off the latest version of their dependencies.

This aspect of the monolithic model was also cited as a disadvantage. Engineers felt the effects of the code churning underneath them, and they were frustrated if their dependencies broke. The primary cited benefit of the multi-repo codebase is having a set of dependencies that do not change until the project owner choses (Figure 6, coded as Stable Dependencies). This ensures that project owners update to a stable version of their dependency that works well with their other dependencies. Dependency mangement also came up as benefit for a multi-repo codebase, but in a different context than for monolithic repos: engineers who cited this wanted to limit their dependencies and control when they created versions of their own libraries for others to use.

A small minority of engineers believed that the difficulty of dependency management with versioned dependencies led to improved code quality through two good practices. First, they believed that if the process is onerous, engineers would not add dependencies unless absolutely necessary. Using a smaller set of dependencies has numerous benefits such as improving build speeds and reducing binary sizes. Second, they believed that difficulty in updating dependencies would force engineers to write higher quality APIs, as there would be no second chance. As soon as the API was released, someone would depend on it, and one would not be able to fix it later.

Reducing cognitive load is important, but there are many ways to achieve this. Reducing cognitive load on the engineer also emerged as a theme in both monolithic and multi-repos. Reduction of cognitive load was cited as a reason for preferring Google's codebase over multi-repo codebases (Figure 5 and Figure 8), yet it was also cited as benefit of both commercial multi-repo and open-source codebases (Figure 6 and Figure 9). The difference was in how engineers believe this could be achieved. For multi-repo and open-source codebases, engineers noted that the source of the reduced cognitive load was a smaller codebase size (coded as Small size), and that this resulted in other benefits as well, like improved build times and faster development velocity. For Google's monolithic repo, reduced cognitive load came up in the context of visibility and developer tools. Engineers found that having access to both APIs and example code made it easier to understand their own code.

6 RELATED WORK

Monolithic repos are a relatively new trend within industry [10, 21], and there has been little opportunity to study them in practice. A study at Google [25] examined how developers use code search in Google's monolithic codebase. The study found that code search is a key component in development workflows; engineers used the ability to search the entire codebase to understand how to use an API, understand what an implementation is doing, or to debug failures. Our work reinforces this finding, as engineers in our survey and observational datasets reported and were repeatedly observed using our monolithic repo for code search.

Version control has been a focus of toolbuilders since the release of Source Code Control System in 1972 [19, 20, 24, 27]. While most early version control systems were based on central per-project repos, we are unaware of instances of

monolithic repos during the early history of version control systems. The rise of distributed version control has provoked some research into the differential impact of centralized and distributed version control on development [3, 7]. This is an orthogonal question to monolithic vs. multi-repo models; either centralized or distributed version control systems can host monolithic repos (although most distributed version control systems have faced scalability problems until recently [10, 12]). The rise of distributed version control has provoked a number of development models based on branches within distributed repos; the research community has studied branching models intended to facilitate development velocity [1, 2]. Distributed repos necessitate merging between these repos or branches; researchers have contributed tools to make developers aware of the merged state of multiple development repos [4], as well as developing novel merge algorithms [13].

Desouza [8] discusses how dependency management occurs through communication channels, for example, when and how engineers notify dependent components that a change is imminent. Monolithic repos make a significant portion of this communication unnecessary, as changes to dependencies can be simultaneous with an update to clients, and all clients are contained in the repo. In a multi-repo system, there is no canonical source of truth that enumerates all of a component's reverse dependencies.

Github and other open-source codebases do have many similarities to a monolithic repo, and these open-source codebases and communities have been well-studied [15, 16, 23, 28]. There are two significant differences between them and a monolithic codebase: there is not a shared set of developer tools, and the dependencies are not kept up to date at a single repo head. In some languages, such as Go, Rust, and Node, some developer tools and dependency management systems are standardized, but as they still presume specific versions, the diamond dependency problem remains.

7 CONCLUSION

Like virtually all choices in computing, the choice of whether to use a monolithic or multi-repo model is a comparison of tradeoffs. Monolithic repos may seem inflexible to engineers accustomed to having more autonomy over the tools and dependencies they use in their projects. However, the standardization of toolchains and dependencies found in monolithic repos eliminates whole classes of versioning problems, encourages consistent and high-quality code, and empowers engineers to study and learn from the institutional knowledge of their company, crystallized in the form of source code.

REFERENCES

[1] Earl T. Barr, Christian Bird, Peter C. Rigby, Abram Hindle, Daniel M. German, and Premkumar Devanbu. 2012. Cohesive and Isolated Development with Branches. In *FASE*.

[2] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-if Analysis. In *FSE*.

[3] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. 2014. How Do Centralized and Distributed Version Control Systems Impact Software Changes?. In *ICSE*.

[4] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *FSE*.

[5] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[7] Brian de Alwis and Jonathan Sillito. 2009. Why Are Software Projects Moving from Centralized to Decentralized Version Control Systems?. In *Proceedings of Cooperative and Human Aspects on Software Engineering*.

[8] Cleidson RB de Souza and David F Redmiles. 2008. An empirical study of software developers' management of dependencies and changes. In *ICSE*.

[9] Mark Florisson and Alan Mycroft. 2017. Towards a Theory of Packages. http://www.cl.cam.ac.uk/~mbf24/packages.pdf. (2017). Accessed: 2017-2-17.

[10] Durham Goode and Siddarth P Agarwal. 2014. Scaling Mercurial at Facebook. https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/. (2014). Accessed: 2017-2-17.

[11] guava 2014. Guava: Google Core Libraries for Java. https://code.google.com/p/guava-libraries/. (2014). Accessed: 2014-11-14.

[12] Brian Harry. 2017. Scaling Git at Microsoft. https://blogs.msdn.microsoft.com/bharry/2017/02/03/scaling-git-and-some-back-story/. (2017). Accessed: 2017-2-17.

[13] Judah Jacobson. 2009. A formalization of darcs patch theory using inverse semigroups. ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf. (2009). Accessed: 2018-2-17.

[14] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K. Smith, Collin Winter, and Emerson Murphy-Hill. 2017. Codes for survey free responses. goo.gl/ZNvYMe. (2017).

[15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[16] Dawn Nafus. 2012. 'Patches don't have gender': What is not open in open source software. *New Media & Society* 14, 4 (2012).

[17] Saeed Noursalehi. 2017. GVFS at Microsoft. https://blogs.msdn.microsoft.com/visualstudioalm/2017/02/03/announcing-gvfs-git-virtual-file-system/. (2017). Accessed: 2017-2-17.

[18] Chris Parnin, Christoph Treude, and Lars Grmmel. 2012. *Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow*. Technical Report. Georgia Institute of Technology.

[19] Santiago Perez De Rosso and Daniel Jackson. 2013. What's wrong with git?: a conceptual design analysis. In *Proceedings of the intl. symposium on New ideas, new paradigms, and reflections on programming & software*. ACM.

[20] Santiago Perez De Rosso and Daniel Jackson. 2016. Purposes, concepts, misfits, and a redesign of git. In *OOPSLA*.

[21] Rachel Potvin and Josh Levenburg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* (2016).

[22] protobuf 2014. Protocol Buffers. http://code.google.com/p/protobuf/. (2014).

[23] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *FSE*.

[24] Marc J Rochkind. 1975. The source code control system. *IEEE Transactions on Software Engineering* 4 (1975), 364–370.

[25] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *FSE*.

[26] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *Proceedings of Cooperative and Human Aspects on Software Engineering*.

[27] Walter F Tichy. 1985. RCS — a system for version control. *Software: Practice and Experience* 15, 7 (1985).

[28] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. 2016. The sky is not the limit: multitasking across GitHub projects. In *ICSE*.

[29] Wikipedia. 2017. Codebases — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Codebase. (2017). Accessed: 2017-10-23.