Tool Choice Matters: JavaScript Quality Assurance Tools and Usage Outcomes in GitHub Projects

David Kavaler
University of California, Davis
dmkavaler@ucdavis.edu

Asher Trockman
University of Evansville
asher.trockman@gmail.com

Bogdan Vasilescu
Carnegie Mellon University
vasilescu@cmu.edu

Vladimir Filkov
University of California, Davis
filkov@cs.ucdavis.edu

***Abstract*—Quality assurance automation is essential in modern software development. In practice, this automation is supported by a multitude of tools that fit different needs and require developers to make decisions about which tool to choose in a given context. Data and analytics of the pros and cons can inform these decisions. Yet, in most cases, there is a dearth of empirical evidence on the effectiveness of existing practices and tool choices.**

We propose a general methodology to model the time-dependent effect of automation tool choice on four outcomes of interest: prevalence of issues, code churn, number of pull requests, and number of contributors, all with a multitude of controls. On a large data set of *npm* JavaScript projects, we extract the adoption events for popular tools in three task classes: linters, dependency managers, and coverage reporters. Using mixed methods approaches, we study the reasons for the adoptions and compare the adoption effects within each class, and sequential tool adoptions across classes. We find that some tools within each group are associated with more beneficial outcomes than others, providing an empirical perspective for the benefits of each. We also find that the order in which some tools are implemented is associated with varying outcomes.

I. Introduction

The persistent refinement and commoditization of software development tools is making development automation possible and practically available to all. This democratization is in full swing: *e.g.*, many GitHub projects run full DevOps continuous integration (CI) and continuous deployment (CD) pipelines.

What makes all that possible is the public availability of high-quality tools for any important development task (all usually free for open-source projects). Linters, dependency managers, integration and release tools, *etc.*, are typically well supported with documentation and are ready to be put to use. Often, there are multiple viable tools for any task. *E.g.*, the linters `ESLint` and `standardJS` both perform static analysis and style guideline checking to identify potential issues with written code. This presents developers with welcome options, but also with choices about which of the tools to use.

Moreover, a viable automation pipeline typically contains more than one tool, strung together in some order of task accomplishment, each tool performing one or more of the tasks at some stage in the process. And the pipelines evolve with the projects – adding and removing components is part-and-parcel of modern software development. As shown in Table I, $38,948$ out of $54,440$ projects gathered in our study use at least one tool from our set of interest; $12,109$ adopt more than one tool over their lifetime, and $2,283$ projects switch from one tool to another which accomplishes a similar task. Having multiple tools available for the same task increases the complexity of choosing one and integrating it successfully and beneficially in a project's pipeline. What's more, projects may have individual, context-specific needs. A problem facing maintainers, then, is: *given a task, or a series of tasks in a pipeline, which tools to choose to implement that pipeline?*

Organizational science, particularly contingency theory [1], predicts that no one solution will fit all [2], [3]. Unsurprisingly, general advice on which tools are "best" is rare and industrial consultants are therefore busy offering DevOps customizations [4]. At the same time, trace data has been accumulating from open-source projects, *e.g.*, on GitHub, on the choices projects have been making while implementing and evolving their CI pipelines. The comparative data analytics perspective, then, implores us to consider an alternative approach: given sufficiently large data sets of CI pipeline traces, can we contextualize and quantify the benefit of incorporating a particular tool in a pipeline?

Here we answer this question using a data-driven, comparative, contextual approach to customization problems in development automation pipelines. Starting from a large dataset of *npm* JavaScript packages on GitHub, we mined adoptions of three classes of commonly-used automated quality assurance tools: linters, dependency managers, and code coverage tools. Within each class, we explore multiple

TABLE I
TOOL ADOPTION SUMMARY STATISTICS

		# Adoption Events Across Projects	
Tool	Task class	Per tool	Per task class
`david`	Dependency Management	$20,763$	$23,917$
`bithound`		900	
`gemnasium`		$3,093$	
`codecov`	Coverage	$2,785$	$15,249$
`codeclimate`		$2,328$	
`coveralls`		$11,221$	
`ESLint`	Linter	$7,095$	$12,886$
`JSHint`		$2,876$	
`standardJS`		$3,435$	

Note: $54,440$ total projects under study
$38,948$ projects which adopt tools under study
$2,283$ projects use different tools in the same task class

tools and distill fundamental differences in their designs and implementations, *e.g.*, some linters may be designed for configurability, while others are meant to be used as-is.

Using this data, and informed by qualitative analyses (Sec. II-B) of issue and pull request (PR) discussions on GitHub about adopting the different tools, we developed statistical methods to detect differential effects of using separate tools *within each class* on four software maintenance outcomes: prevalence of issues (as a quality assurance measure), code churn (as an indication of setup overhead), number of pull requests (for dependency managers, as an indication of tool usage overhead), and number of contributors (as an indication of the attractiveness of the project to developers and the ease with which they can adapt to project practices). We then evaluated how different tools accomplishing the same tasks compare, and if the order in which the tools are implemented is associated with changes in outcomes. We found that:

- Projects often switch tools within the same task category, indicating a need for change or that the first tool was not the best for their pipeline.
- Only some tools within the same category seem to be generally beneficial for most projects that use them.
- `standardJS`, `coveralls`, and `david` stand out as tools associated with post-adoption issue prevalence benefits.
- The order in which tools are adopted matters.

II. JavaScript Continuous Integration Pipelines with Different Quality Assurance Tools

In this work, we study how specific quality assurance tool usage, tool choice, and the tasks they accomplish are associated with changes in outcomes of importance to software engineers, in the context of CI. We focus on CI pipelines used by JavaScript packages published on *npm*: JavaScript is the most popular language on GitHub and *npm* is the most popular online registry for JavaScript packages, with over 500,000 published packages. While there is hardly any empirical evidence supporting the tool choices, there has been much research on CI pipelines. We briefly review related prior work and discuss tool options for our task classes of interest.

A. CI, Tool Configuration, and Tool Integration

CI is a widely-used practice [5], by which all work is "continuously" compiled, built, and tested; multiple CI tools exist [6]. The effect of CI adoption on measures of project success has been studied, with prior work finding beneficial overall effects [7]. As part of CI pipelines, third party tools for specific tasks are often used. Though the notion of using a pre-built solution for common development tasks is attractive, these tools are often far from being usable off-the-shelf.

Hilton *et al.* describe multiple issues that developers experience when configuring and using CI [8], including lack of support for a desired workflow and lack of tool integration: developers want powerful and highly configurable systems, yet simple and easy to use. However, Xu and Zhou [9] find that over-configurable systems hinder usability. It has also been shown that developers often don't use the vast majority of configuration options available to them [10]. Misconfigurations have been called "*the most urgent but thorny problems in software reliability*" [11]. In addition, configuration requirements can change over time as older setups become obsolete, or must be changed due to, *e.g.*, dependency upgrades [12].

The integration cost for off-the-shelf tools has been long-studied in computer science [13]–[17]. However, much of this research is dated prior to the advent of CI and large social-coding platforms like GitHub. Thus, we believe the topic of tool integration deserves to be revisited in that setting.

B. Tool Options Within Three Quality Assurance Task Classes

There are often multiples of quality assurance tools to accomplish the same task, and developers constantly evaluate and choose between them. E.g., a web search for JavaScript linters turns up many blog posts discussing options and pros and cons [18], [19]. We consider sets of tools that offer similar task functionality as belonging to the same *task class* of tools. Specifically, we investigate three task classes commonly-studied in prior work (*e.g.*, [20]–[22]) and commonly-used in the JavaScript community: linters, dependency managers, and code coverage tools.

Below, to better illustrate fundamental differences in tools within the same task class, and note tradeoffs when choosing between tools, we intersperse the description of the tools with links to relevant GitHub discussions. To identify these discussions, we searched for keyword combinations (*e.g.*, "eslint" + "jshint") on GitHub using the GitHub API (more details in Section IV), identifying 47 relevant issue discussion threads that explicitly discuss combinations of our tools of interest. One of the authors then reviewed all matches, removed false positives, and coded all the discussions; the emerging themes were then discussed and refined by two of the authors.[1]

Linters. Linters are static analysis tools, commonly used to enforce a common project-wide coding style, *e.g.*, ending lines with semicolons. They are also used to detect simple potential bugs, *e.g.*, not handling errors in callback functions. There has been interest in the application of such static analysis tools in open-source projects [23] and CI pipelines [24]. Within *npm*, common linters include `ESLint`, `JSHint`, and `standardJS`. Qualitative analysis of 32 issue threads discussing linters suggests that project maintainers and contributors are concerned with specific features (16 discussions) and ease of installation (15), with some recommending specific linters based on personal preferences or popularity, favoring those that they have used before without much justification (8), *e.g.*, [25]–[27]. Two categories of linters emerge: First, `ESLint` and `JSHint` are both meant to be configured to suit a project's specific needs and preferences; among them, `ESLint` is more highly configurable as it allows for custom plugins. Second, `standardJS` is meant to be a "drop-in" solution to linting: it is not meant to be further configured.

[1]Surprisingly, across $38,948$ projects in our data which adopted tools, only 47 issue threads explicitly discuss combinations of our tools of interest. This relatively low frequency suggests a lack of deliberateness in choosing tools, which underscores the need for this study, especially given our findings.

ESLint's *high configurability* makes it a common choice (9 discussions [28]–[33]). Ten projects switched to ESLint or JSHint after using older linters which lack this functionality; in fact, four view ESLint as a combination of older linters [31], [34]–[36]. On the other hand, increased configurability comes with increased overhead, as noted in one discussion: "*ESLint has so many options, [which] often leads to insignificant debates about style... This makes the project harder to maintain and hinders approachability*" [37].

In contrast, standardJS addresses the problem of optionality, encouraging projects to adopt a *standard configuration*. One developer notes it may be more convenient for contributors, who are more likely to be familiar with the "standard" style [38]. However, another developer finds it less appealing as they "*simply don't agree with those default settings*" [39]. Overall, ten of the found issues included some form of discussion between contributors and maintainers. This indicates, at least partly, that some developers are interested in discussing the pros and cons of linter choice.

Based on these anecdotes, we hypothesize that a complex project with project-specific code style requirements may be more likely to use ESLint or JSHint, while a smaller project may opt for standardJS because it is easier to set up. While highly configurable tools like ESLint and JSHint require more effort in configuration, they may nullify disputes over code formatting and facilitate code review. standardJS requires less initial effort from project maintainers and may be appealing to new contributors who are familiar with the common code style. However, since its adoption likely entailed less configuration effort, maintainers may need to spend more time on code review or address more issues about code styling.

Coverage Tools. Coverage tools, executed locally or as part of a CI pipeline, compute and report code coverage (typically line coverage), a measure of test suite quality. Popular JavaScript coverage tools include Istanbul and JSCover. If used as part of a CI pipeline, the coverage results are often sent to a third-party service such as coveralls or codeclimate, which archive historical coverage information and offer dashboards. Third-party tools require some additional configuration over simply using a coverage reporter locally, and may result in more issues and pull requests as code coverage information is made available or reacted to by the community.

Out of ten issue discussions about coverage tools, in four developers talked about not wanting to spend too much time on configuration [32], [40], [40], [41]. One developer states that the sole purpose of such a tool is to get the coverage badge, *i.e.*, to signal that the project cares about test coverage [32]. As expected, two developers state that they use a certain coverage tool simply because they are more familiar with it [42], [43]. It seems developers want to invest minimal effort in coverage services, but seem unaware that some services require more overhead than others: "*a yml config file seems silly when all that you want is coverage*," states a developer [32].

coveralls and codecov are considered very focused services, generally providing, as per the issue discussions, only coverage [32], [40], [44], [45], without additional features. Two developers claim that coveralls is unreliable [45], [46], inspiring a switch to codecov, which is said to have a better user experience [45]. codeclimate provides other services besides coverage, *e.g.*, linting, and developers may be confused by its high configurability [32], [45]: "*CodeClimate isn't sure if it wants to be a CI or not*," states a developer [32].

Dependency Managers. Dependency management services help keep dependencies up-to-date and secure. They make the status of dependencies visible to project maintainers, encouraging updates after new releases through automated notifications. Dependency management is a popular research topic [47]–[50]. david, gemnasium, and snyk are examples of such tools that require manual intervention; their results are often displayed as badges [21], [51] on a project's README file. Greenkeeper, a GitHub bot, provides *automatic* dependency updates at each new release. It creates a PR with the updated dependency, which in turn triggers the CI service to run the build. This allows the project maintainers to immediately see if an upgrade will incur significant work. If the build does not break, then it is likely safe to merge the update. Hence, projects must choose if they would like manual or automatic dependency upgrades. Intuitively, manual upgrades may require more work for developers due to upgrading and testing. However, automatic upgrades increase the number of PRs to review, and may cause notification fatigue [52].

Dependency managers face similar trade-offs between configurability and ease of use, although we found only five issues discussing them. Our analysis suggests that developers value ease of installation, *e.g.*, low configuration overhead, over other concerns: *e.g.*, one developer prefers david since it's a specific, easy-to-install service [53], as opposed to bithound, which, as another developer notes, provides dependency management as well as numerous extra features [54].

In summary, most projects decide which tool to select based on personal preference, popularity, or implementation cost. However, we find very little explicit discussion of tool choice within issue discussions, suggesting that tool choice may not be very deliberate.

III. Research Questions

We investigate three main research questions:

RQ$_1$: How often do projects change between tools within the same task class?

Rationale. GitHub projects have varying needs, and thus likely require configurable tools. However, given the complexity of tool configuration [8], [10], [55] in general and the requirement of keeping configurations up-to-date [12], developers may need to change between tools within a given task class over time. RQ$_1$ studies the stability of the projects' quality assurance pipelines.

RQ$_2$: Are there measurable changes, in terms of monthly churn, pull requests, number of contributors, and issues, associated with adopting a tool? Are different tools within an equivalence class associated with different outcomes?

Rationale. Despite great research interest in CI pipelines, there is a distinct lack of empirical evidence regarding tool choice and associations with outcomes of interest. Prior work has shown that there are costs when integrating pre-built tools in general [13]–[17], but the empirical evidence for differential costs *within task classes* is lacking, especially in the choice-laden GitHub open-source setting. RQ$_2$ focuses on these costs.

To drill into our second research question, we investigate specific subquestions:

RQ$_{2.1}$: Is linter tool adoption associated with lower monthly churn?

Rationale. Prior work has reported that JavaScript developers use linters primarily to prevent errors [20]. We expect a decrease in monthly churn after linters start being used, because the linter may catch issues that would have otherwise had to be changed after the initial commit.

RQ$_{2.2}$: Is `standardJS` associated with more monthly contributors?

Rationale. As discussed in Section II, `standardJS` is meant as a "drop-in" solution, implementing standard JavaScript style guidelines, while `ESLint` is more customizable but more complex. Developers' potential surprise or frustration with unusual style guidelines may impact how engaged they remain with a project. Therefore, we expect that other variables held constant, projects using more standard JavaScript style guidelines would be more attractive to developers, who, in turn, may remain engaged longer.

RQ$_{2.3}$: Are coverage tools associated with immediate overhead, measured by monthly churn and PRs?

Rationale. Test coverage has been described as important for software quality [56]. However, achieving full coverage is difficult [57], and relies on an appropriate test suite. Thus, we hypothesize that the adoption of a coverage reporting tool is associated with an immediate overhead cost, as coverage tools likely require moderate to large infrastructure and test suite changes to facilitate their reporting.

RQ$_{2.4}$: Are dependency managers associated with more monthly churn and PRs?

Rationale. Prior work [21] found that dependency management is (according to one developer) "*one of the most significantly painful problems with development*". The authors report that developers identified strategies that can be roughly categorized as "quick", "scheduled", or "reactive". We note that the existence of a dependency management tool may not affect monthly churn and PRs for projects with a "scheduled" update policy (*i.e.*, a systematic method for reviewing dependencies), but may affect those with "quick" or "reactive" policies, as they are more likely to be sensitive to such a tool's warnings; dependencies should be updated, and developers may do this through PRs, additionally reflected in churn.

RQ$_{2.5}$: Are the studied tools associated with issue prevalence?

Rationale. Since the tools studied are quality assurance tools, developers may hope that adopting a tool will decrease the prevalence of reported issues, either right away or over time. A reduction in the number of issues can give developers more time to provide new features, maintain existing ones, *etc.* If we observe such an association, it can provide evidence for the argument that, all else being equal, the tool associated with fewer issues should be adopted.

RQ$_3$: Are certain tool adoption *sequences* more associated with changes in our outcomes of interest than others?

Rationale. Lastly, we investigate whether integration cost is incurred differently given prior tool adoptions. For example, implementing a coverage tool before a linter may incur less integration cost than in the opposite order, as coverage tools are accompanied by large test suites; if there are systematic issues with a test suite, a linter may point them out. Thus, if the test suite is not initially written according to a linter's guidelines, there may be additional cost associated with integrating a linter as the improper formatting must be fixed.

IV. Data and Methods

Here, we discuss practical matters of data collection, including extraction of tool adoption events, task class creation, negative control data, model building, and interpretation.

A. Data Collection

To determine when projects incorporate linters and coverage services into their CI pipelines, we downloaded build information from Travis CI, the most popular CI service on GitHub. While such tools may be declared in a project's `.travis.yml` configuration file, they are often delegated to shell scripts, task runners, Makefiles, or the *npm* `package.json` file, making discovery non-trivial. Hence, instead of scanning repositories for evidence of tool use, we scan the Travis CI build logs.

Travis CI allows projects to define a *build matrix*, which specifies the configuration for numerous *jobs*. The build matrix could specify that only one, or more than one jobs involve running linters or coverage services. Hence, for the 92,479 most popular *npm* projects we had access to, we queried the Travis CI API to download build information, *e.g.*, date, status, and if the build was started via push or pull request, as well as all associated jobs. This resulted in 7.03 million rows of build metadata and 18.6 million rows of corresponding jobs.

To determine the time of tool adoption, we downloaded all associated job logs per project at *monthly intervals*: it would have been intractable to download all 18.6 million logs. The granularity in adoption dates in our data is one build per month per project. In total, we downloaded logs for 80,558 projects.

We say that a project is using a given tool in a given month if *any* of the job logs contain corresponding traces. We scanned the job logs case insensitively for instances of the tool names ("eslint", "jshint", "standard", "coveralls", "codecov", "codeclimate") occurring on lines starting with > or `$`, *i.e.*, lines corresponding to the execution of a command. This heuristic is necessary, since otherwise we would detect many false positives, *e.g.*, for projects that include dependencies with names including "eslint" that do not actually run `ESLint` as part of the CI build. The adoption date of a tool is the first month that has a job log containing an execution of that tool.

Dependency managers, however, are not typically included in the CI pipeline. Instead, we determine their adoption date from the corresponding badge on a project's README [51].

For project-level GitHub measures (*e.g.*, number of commits, number of PRs), we use data from the March 2017 dump of GHTorrent [58], aggregated monthly, and join this to our tool adoption data. For our monthly authors outcome, we look at the authors for all commits in a given month and project, recording how many are unique. We also mined 1.4 million issues for 62,548 *npm* projects from GHTorrent (the remaining projects had no issues) and the corresponding 5.3 million issue comments from GHTorrent's MongoDB raw dataset.

B. Choosing Tools That Do Not Overlap

We want to model both tool adoption and tool switching within task classes, if it occurs. Identifying precise tool removals from Travis CI job logs is prone to false positives, as projects may (and according to manual review of our data, do) move a given tool run outside of the Travis CI job itself, instead running it as, *e.g.*, a pre-CI job step. The difficulty of parsing CI job logs has been noted by prior work [59]. Instead, to infer tool removal, we carefully chose the task classes and popular (competing) tools within them, which should not be used simultaneously. Thus, we say that a tool is removed when another tool within the same task class is adopted. We believe this is a reasonable decision, as, *e.g.*, we find it unlikely that a project will use two different linters simultaneously; thus, the adoption of a new linter likely coincides with the removal of the old one. This is partially supported by our qualitative analysis. This assumption affects most data shown in Table I, therefore we acknowledge it as a threat to validity.

C. On Negative Controls and Interventions

In standard experimental design with intervention, *e.g.*, clinical trials, there are often two basic groups: a *negative control* group, which receives no intervention, and a *treatment* group, which receives an intervention. However, with observed data as on GitHub, generally these two groups do not exist *a priori*; the intervention is observed, rather than induced in a specific group with proper prior randomization. In order to view tool adoption as an intervention, we first form an analogous negative control group; otherwise, our data would consist of only those projects that receive an intervention. Lacking a proper negative control reduces statistical power, as there exists no group to compare to the treatment group. Thus, our data also consists of projects which do not adopt any of the tools under study, providing a form of negative control.

D. Time Series Modeling

We use *linear mixed-effects regression* [60] (LMER) to measure the relationship between our outcomes (dependent variables) and our explanatory variables of interest, under the effect of various controls. LMER can be seen as an extension of standard ordinary least squares (OLS) linear regression, allowing for the addition of *random effects* on top of standard *fixed effects*. Fixed effects in an LMER can be interpreted the same way as coefficients in an OLS regression. Random effects are used to model (often sparse) factor groupings that may imply a hierarchy (nesting) within the data, or to control for *multiple observation* of the same subjects in time. In OLS regression, multiple observation can lead to multicollinearity, which can limit inferential ability [61]. LMER explicitly models correlation within (and between) random effect groupings, thus reducing the serious threat of multicollinearity when modeling longitudinal data using other methods.

In our data, we use a random effect grouping for each project. Ideally, one would be able to fit a separate OLS regression to each project, inspecting individual coefficients to test hypotheses. However, data within project groups are often sparse, *i.e.*, many projects have too few data points to model individually in a statistically robust manner. In an OLS regression, one may attempt to combine data from all projects, and model project groups as a factor. However, this approach is also vulnerable to sparsity, for similar reasons. Thus, random effects (or, *e.g.*, LASSO regression [62]) can be used as a form of *shrinkage*. Each random effect group level is shrunk towards its corresponding "grand mean", analogous to standard regularization strategies [63].

One can *interpret* this as fitting a separate OLS regression to each project, with a constraint: the project-level intercept for each individual OLS regression is a deviation from the mentioned "grand mean". This is distinctly different than fitting a standard OLS regression to each project, as this random-effect constraint allows us to model projects with far fewer data points than would be allowed when fitting separate OLS regressions, due to the aforementioned constraint.

We model the effect of tool adoption over time across GitHub projects for multiple outcomes of interest. To do this, we use a model design analogous to regression discontinuity design (RDD) [64], [65] using mixed-effects regression [66]. In this work, we treat each tool adoption event as an *intervention*, analogous to administering a drug treatment in a clinical trial. RDD is used to model the extent of a discontinuity in a group at the moment of intervention, and lasting effects post-intervention. The assumption is that if an intervention has no effect on an individual, there would be no significant discontinuity in the outcome over time; the post-intervention trajectory would be continuous over the intervention time. Thus, RDD allows us to assess how much an intervention changes an outcome of interest immediately and over time (via change in the trajectory slope). Fig. 1 shows real data for monthly pull requests, centered at the time of `ESLint` adoption. Note the discontinuity at the time of intervention, which serves as motivation for the applicability of RDD here.

Fig. 2 depicts the four theoretical cases for an intervention in our model design. Prior to an intervention, there is some trend, depicted as a dashed line. At the moment of intervention, there can be a positive or negative immediate effect (discontinuity), in addition to a positive or negative change in slope. Thus, our design can be expressed as:

$$\begin{aligned} y_{ij} = \; & \alpha_i + \beta_i time_{ij} + \gamma_i intervention_{ij} + \\ & \delta_i time\_after\_intervention_{ij} + \pi_i controls_{ij} + \epsilon_{ij} \end{aligned}$$

where j is a particular project and i indexes the set of observations for a given project, with the ability to have

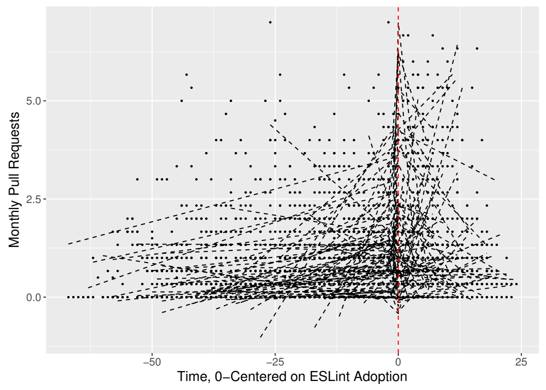

Fig. 1. ESLint intervention for PRs, centered at 0, 100 projects sampled. Basic OLS trajectories fitted before / after intervention shown as dashed lines.

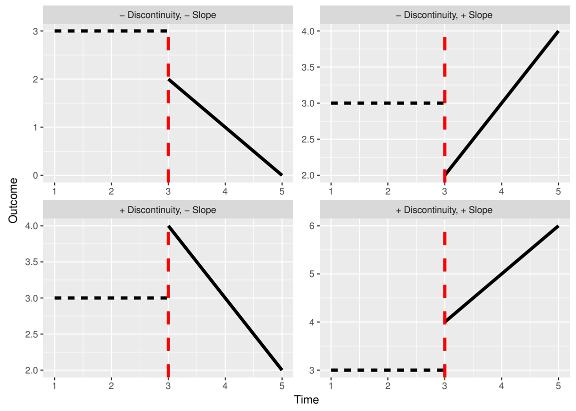

Fig. 2. Four theoretical basic cases for longitudinal mixed-effects models with interventions. Intervention times marked in red.

multiple *time-varying* controls. Note that we are not limited to a single intervention as expressed above; we can (and do) have multiple tool adoption events within the same model. The time$_{ij}$ variable is measured as months from the start of a project to the end of our observation period; intervention$_{ij}$ is an indicator for time occurring before (0) or after (1) tool adoptions; and time_after_intervention$_{ij}$ is a counter measuring months since the tool adoption. In mixed-effects regression, these parameters are aggregated across all groups (projects), yielding a final coefficient useful for interpretation. This RDD formulation, along with the usage of mixed-effects regression, allows for the simultaneous analysis of many projects, all with varying observation periods (*i.e.*, differing project lifetimes), with interventions occurring at different points in time (*i.e.*, not necessarily aligned). This flexibility allows for complex and statistically robust analysis, with simple interpretations.

For points before the treatment, holding controls constant, the resulting regression line has a slope of β_i; after the treatment, $\beta_i + \delta_i$. The effect of an intervention is measured as the difference between the two regression values of y_{ij} pre- and post-treatment, equal to γ_i. This design is used in 9 different models to examine RQ$_2$.

To examine RQ$_3$, we build 3 additional longitudinal models, with observed tool adoption sequences modeled as a factor, along with the same controls used in our prior models. The general form of the model for each outcome of interest is:

$$y_{ij} = \alpha_i + \beta_i time_{ij} + \kappa_i current\_tool\_sequence_{ij} + \pi_i controls_{ij} + \epsilon_{ij}$$

where current_tool_sequence$_{ij}$ is a cumulative factor indicating what tools have been adopted for a given project until then, including a random intercept for each project j. E.g., if project j adopts tool `A` in month 1, `B` in month 2, and `C` in month 4, with `A` and `C` in the same task class, our data would have 4 observations for project j, with a tool sequence of {`A`, `A`→`B`, `A`→`B`, `B`→`C`}, where `A`→`B` is repeated in month 3 and `A` is removed in month 4 as `C` belongs to the same task class. Using this formulation, we can analyze whether an association exists between a given tool sequence and our outcomes of interest. In both formulations, we include additional fixed effects (multiple π_i estimated) as controls, *e.g.*, project size, popularity, and age.

The idea of variable significance in LMER is greatly debated in statistics, mostly due to the lack of classical asymptotic theory as used for inference in, *e.g.*, OLS regression [67]–[70].[2] These issues can be dampened by large sample sizes (as we have), but not completely avoided. We report significance using Satterthwaite approximation for denominator degrees of freedom for regression variable t-tests, implemented using the *lmerTest* R library [71]. This is considered a reasonable approach when sample sizes are large [69].

We account for multicollinearity by considering only fixed-effect control variables with VIF (variance inflation factor) less than 5 [61], as having many fixed effects along with a complex design structure can introduce issues in model estimation. In addition, LMER is sensitive to extreme-valued parameters, and having many fixed effects can require heavy re-scaling, making it difficult to interpret results. To avoid extreme values in the fixed effects, we *log* transform the *churn* variable, which has high variance. We trim outliers by removing the top 1% of values in each considered variable to further avoid potential high-leverage issues.

Due to the large variance in our outcomes of interest, there is a risk of spurious significant discontinuities; a significant discontinuity at intervention time may be found due to mere noise in the signal. Thus, we smooth each outcome using *moving average* smoothing, with a window of 3 months.[3] In addition, we remove data for the exact month in which the tool is adopted. These two steps act to combat the risk of spurious discontinuities, and should not negatively affect our results as we are interested in the trend of each outcome over time, rather than the absolute, high variance value for a particular month.

We report psuedo-R^2 values as described by Nakagawa and Schielzeth [72], called *marginal* R^2 and *conditional* R^2. The marginal R^2 can be interpreted as the variance described by the fixed effects alone; conditional R^2 as the variance described by both fixed and random effects. As our data set consists of 17,137 projects after filtering for described issues, we expect

[2]Examples of this debate abound; listed references are a small sample.
[3]No discernible difference was found with windows of size 3 to 6.

project-specific idiosyncrasies to play a large part in model fit. Thus, we expect our marginal R^2 to be much lower than our conditional R^2. We perform standard model diagnostic tests to evaluate model fit [66]; our models pass these tests. As we build many models, we also perform multiple hypothesis testing (p-value) correction by the Benjamini and Hochberg method [73]; this correction balances false positives and false negatives, unlike the more conservative Bonferroni correction which aims to lower only false positives. The reported p-values are the corrected ones.

E. Threats to Validity

If multiple tools are adopted in the same month, we do not know their ordering. However, we do not believe this is a large threat, as this happens in only 12% of all tool adoption events. In addition, as stated in Sect. IV-B, we assume that when a new tool is adopted in an existing task class, the old tool is removed. The effect of this assumption is likely small, as vast majority of projects adopt only one tool in a class. There is the potential issue of multiple aliases; the same developer using multiple usernames, which we do not account for. As our primary interest is at the project level, having multiple instances of the same person per project will not affect project-related conclusions.

Recall that we use mixed-effects models. The notion of goodness-of-fit in these models is highly debated [72], with many available metrics for assessment [74]–[77]. We note that our models for RQ2 have relatively low marginal R^2 values. However, our conditional R^2 are much higher (44.8% to 58.9%), suggesting appropriate fit when considering project-level differences. We also note relatively small effect size for tool interventions and post-intervention slopes. We believe this is expected, as we have controls for multiple covariates that have been shown to highly associate with our outcomes; thus, these controls likely absorb variance that would otherwise be attributed to tools, leading to smaller effect size for tool measures. Finally, as with any statistical model, we have the threat of missing confounds. We attempted to control for multiple aspects which could affect our outcomes and made a best-effort to gather data from as many projects as possible.

V. Results and Discussion

Recall that we are interested in three specific task classes of tools: linters, dependency managers, and coverage reporters. Here, we discuss the results of data analysis and model fitting.

A. RQ$_1$: Patterns of Tool Adoption

Table I provides tool adoption summary statistics, per task class. It shows that some tools dominate in a class, *e.g.*, `david`, and that there are numerous adoptions of each tool. Next we study sequential tool adoptions, *i.e.*, adoption of a tool after one was already adopted, within the same task class. Fig. 3 depicts per-project adoption sequences with alluvial diagrams. Specifically, we take the full tool adoption sequence and plot how many projects "flow" from one tool to the next (x-axis) in sequential order. The insets depict a "zoomed-in" view of

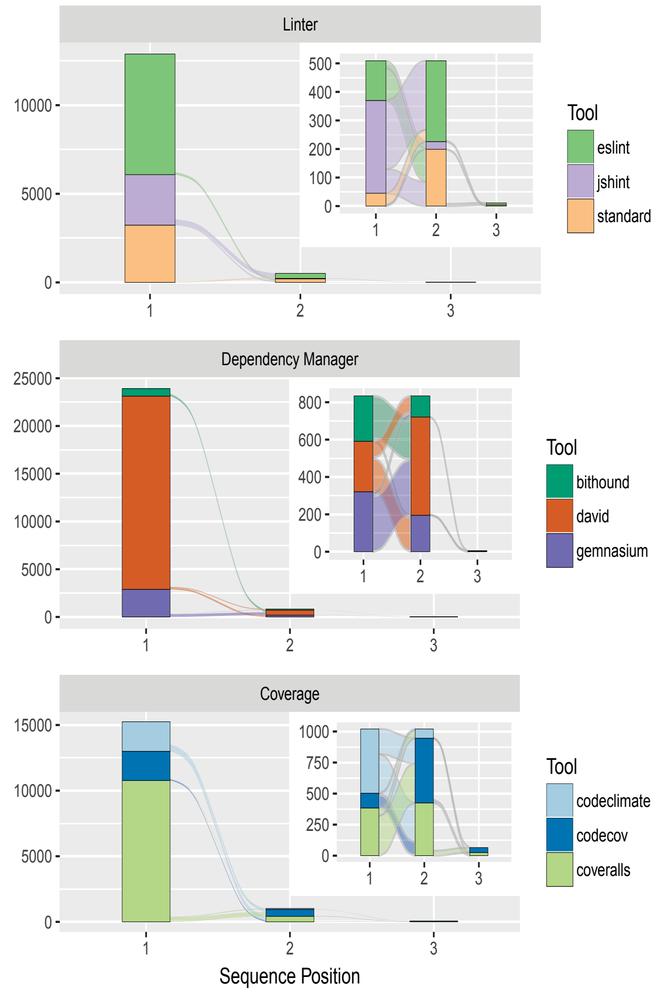

Fig. 3. Alluvial diagrams for per-project adoption sequences for tools within the same task class. Insets: among projects which changed tools.

the subset of projects which have changed to another tool after adopting the first one. We augment these empirical results with findings from our qualitative analysis (Section II-B).

It is apparent that the majority of projects appear to adopt one tool and stick with it. This may be due to a prevalence of a "set it and forget it" mentality; developers wish to use a tool and not have to deal with the additional configuration work of changing that tool [32], [40]. This is reflected in a statement by one commenter: "*there are too many tools to manage*" [78]. It is also possible, though not plausible, that most projects get their tool choice right, from the start.

We also see that a large number of projects (2,283; Table I) choose to adopt another tool within the same task class after their adoption of the first. For linters, among projects which adopt additional tools (inset figure), most projects first adopt `JSHint`, and later `ESLint`. This could be due to apparent popular opinion that `ESLint` does a better job than `JSHint` for the same tasks, is more highly configurable, or provides a superset of features, as evidenced by multiple blog posts [18],

TABLE II
LINTER MODELS

	Dependent variable (monthly):			
	log(Churn + 1)	PRs	Unique Authors	Issues
	Coeffs (Err.)	Coeffs (Err.)	Coeffs (Err.)	Coeffs (Err.)
Authors	0.454*** (0.007)	0.377*** (0.004)		0.051*** (0.004)
Commits	0.052*** (0.001)	0.014*** (0.000)	0.009*** (0.000)	0.015*** (0.000)
PRs	−0.077*** (0.003)		0.067*** (0.001)	
Churn		−0.039*** (0.001)		0.009*** (0.001)
time	−0.049*** (0.000)	0.010*** (0.000)	−0.005*** (0.000)	0.013*** (0.000)
eslint_int	0.194*** (0.026)	0.428*** (0.015)	0.105*** (0.006)	0.101*** (0.018)
eslint_after	−0.055*** (0.004)	0.022*** (0.002)	−0.000 (0.001)	0.003 (0.002)
jshint_int	−0.109*** (0.031)	0.250*** (0.018)	0.057*** (0.007)	0.279*** (0.021)
jshint_after	−0.008*** (0.002)	−0.011*** (0.001)	−0.002*** (0.001)	−0.013*** (0.002)
standard_int	−0.235*** (0.038)	0.439*** (0.022)	0.097*** (0.009)	0.053* (0.026)
standard_after	−0.044*** (0.005)	0.031*** (0.003)	0.004*** (0.001)	−0.010** (0.003)
Intercept	4.543*** (0.014)	0.264*** (0.009)	1.000*** (0.003)	0.418*** (0.011)
Marginal R^2	21.6%	8.9%	13.0%	2.4%
Conditional R^2	54.3%	58.4%	45.0%	56.7%
Note:				*p<0.05; **p<0.01; ***p<0.001

TABLE III
COVERAGE MODELS

	Dependent variable (monthly):			
	log(Churn + 1)	PRs	Unique Authors	Issues
	Coeffs (Err.)	Coeffs (Err.)	Coeffs (Err.)	Coeffs (Err.)
Authors	0.453*** (0.007)	0.381*** (0.004)		0.051*** (0.004)
Commits	0.052*** (0.001)	0.014*** (0.000)	0.009*** (0.000)	0.015*** (0.000)
PRs	−0.077*** (0.003)		0.068*** (0.001)	
Churn		−0.040*** (0.001)		0.009*** (0.001)
time	−0.048*** (0.000)	0.011*** (0.000)	−0.005*** (0.000)	0.014*** (0.000)
coveralls_int	−0.131*** (0.020)	0.289*** (0.012)	0.102*** (0.005)	0.128*** (0.014)
coveralls_after	−0.027*** (0.002)	0.003** (0.001)	−0.005*** (0.001)	−0.013*** (0.001)
codeclimate_int	0.185*** (0.050)	0.124*** (0.029)	0.068*** (0.012)	0.037 (0.035)
codeclimate_after	−0.036*** (0.005)	0.002 (0.003)	−0.007*** (0.001)	−0.005 (0.004)
codecov_int	0.257*** (0.051)	0.452*** (0.029)	0.173*** (0.012)	−0.029 (0.035)
codecov_after	−0.101*** (0.009)	0.032*** (0.005)	−0.002 (0.002)	−0.011 (0.006)
Intercept	4.561*** (0.014)	0.264*** (0.009)	0.994*** (0.003)	0.422*** (0.011)
Marginal R^2	21.6%	7.7%	12.7%	2.4%
Conditional R^2	54.2%	58.8%	44.8%	56.8%
Note:				*p<0.05; **p<0.01; ***p<0.001

[79], [80] and our qualitative analysis [31], [34]–[36].

For dependency managers, we see that `david` is the predominant choice. This may be due to its ease of installation [53], as compared to, *e.g.*, `bithound`, which provides dependency management and many other features [54]. We also see that among projects which switch from `david`, there is a bifurcation in the next tool adopted, between `bithound` and `gemnasium`. We were unable to find any explicit issue discussions regarding `gemnasium` to illuminate this switch.

For coverage, we see that `coveralls` is the most popular tool by far (11,221 adoptions, compared to 2,328 for `codeclimate` and 2,785 for `codecov`). However, some online discussions [81]–[83] and issue discussions [45], [46] suggest that project communities may be starting to favor `codecov` due to its superset of features and its direct integration with GitHub, BitBucket, and other git-related services, and due to its better support. This is supported by Fig. 3; among those who switch from `coveralls`, almost all later move to `codecov`. In addition, we see that projects which switch from `codeclimate` next adopt either `codecov` or `coveralls` with a fairly even split. This switch may be due to the stated confusion regarding the wide array of varying features that `codeclimate` provides [32], [45]. In summary:

Answer to RQ$_1$: *Most projects choose one tool within a task class and stick to it for their observed lifetime. However, when projects adopt additional tools within the same task class, they often move in the same direction as other projects,* e.g.*,* `JSHint` *to* `ESLint`*.*

B. RQ$_2$: Comparing Tools

Starting from the trace data of tool adoptions, here we compare the effect of tool adoption on our outcomes of interest. To examine the effects of each task class individually, we build a separate set of models for each task class and for each outcome, resulting in 12 models.

Tables II, III, and IV show our model results for linters, coverage tools, and dependency managers, respectively. The *_int* naming convention identifies effects of the intervention;

TABLE IV
DEPENDENCY MANAGER MODELS

	log(Churn + 1)	PRs	Unique Authors	Issues
	Coeffs (Err.)	Coeffs (Err.)	Coeffs (Err.)	Coeffs (Err.)
Authors	0.456*** (0.007)	0.383*** (0.004)		0.052*** (0.004)
Commits	0.052*** (0.001)	0.014*** (0.000)	0.009*** (0.000)	0.015*** (0.000)
PRs	−0.078*** (0.003)		0.068*** (0.001)	
Churn		−0.040*** (0.001)		0.009*** (0.001)
time	−0.049*** (0.000)	0.011*** (0.000)	−0.005*** (0.000)	0.013*** (0.000)
gemnasium_int	−0.168*** (0.043)	−0.022 (0.025)	0.048*** (0.010)	0.061 (0.030)
gemnasium_after	−0.003 (0.003)	0.009*** (0.002)	0.001* (0.001)	−0.001 (0.002)
david_int	−0.182*** (0.021)	0.199*** (0.012)	0.082*** (0.005)	0.138*** (0.015)
david_after	−0.009*** (0.002)	0.005*** (0.001)	−0.001 (0.000)	−0.011*** (0.001)
bithound_int	0.002 (0.107)	0.418*** (0.061)	0.088*** (0.026)	0.078 (0.073)
bithound_after	−0.028* (0.014)	0.037*** (0.008)	0.000 (0.003)	−0.014 (0.009)
Intercept	4.567*** (0.014)	0.289*** (0.009)	0.998*** (0.003)	0.415*** (0.011)
Marginal R^2	21.6%	7.1%	12.5%	2.4%
Conditional R^2	54.1%	58.9%	44.8%	56.8%
Note:				*p<0.05; **p<0.01; ***p<0.001

_after refers to post-intervention slopes. For models with *log*-transformed outcomes, coefficients are interpreted as a percent increase or decrease, *e.g.*, if a variable coefficient is reported as 0.123, this reflects a $e^{0.123} - 1 = 13.1\%$ increase in the outcome per unit increase in the given variable. For models with no transformation of the outcome, a unit increase in an explanatory variable is associated with the corresponding coefficient increase in the outcome variable.

1) RQ$_{2.1}$: Linter Usage vs. Monthly Churn: We find that the results are mixed in terms of discontinuity in churn: adopting `standardJS` is associated with a 20.9% reduction in monthly churn; `JSHint` is also associated with a reduction (10.3%). However, `ESLint` is associated with a 21.4% *increase* in monthly churn. This difference might be explained by `ESLint`'s higher configurability as compared to `JSHint` and `standardJS`. It is possible that developers are initially more familiar with "standard" JavaScript coding practices, but are interested in adding rules to their style guidelines. Thus, when `ESLint` is adopted, they must alter a significant amount of the code base to adhere to these new rules, seen through increased churn. It is interesting to note that this potential overhead is referred to by one developer in our qualitative analysis [37]. However, all post-intervention slopes are negative, indicating that this initial churn cost for `ESLint` is eventually paid (in 4 months, all other variables held constant), and monthly churn is lowered after the adoption event. Thus, for churn, our expectation is supported by the data.

2) RQ$_{2.2}$: StandardJS vs. Monthly Authors: Here, in apparent contrast to our expectations, we find that adopting `standardJS` (0.097) associates with a lower positive intervention discontinuity in the monthly unique authors than adopting `ESLint` does (0.105), while higher than `JSHint` (0.057). However, when it comes to slope changes, we see that `standardJS` is associated with a slight increase in post-intervention slope (0.004) and `ESLint` has an insignificant post-intervention slope coefficient. Thus, though `standardJS` is associated with a lower immediate increase in monthly unique authors than `ESLint`, after 2 months (all other variables held constant), the monthly unique authors trend for `standardJS` usage will meet that of `ESLint`, and proceed to exceed it over time.

We note that `JSHint` shows a slight negative post-intervention slope (-0.002). Though the effect is small, our qualitative findings show that many developers have moved to prefer `ESLint` and `standardJS` over `JSHint`. Thus, this negative trend associated with `JSHint` might be explained by developers becoming more attracted to alternative options.

3) RQ$_{2.3}$: Coverage vs. Immediate Overhead: Our results here are mixed; `coveralls` is associated with a 12.3% discontinuous decrease in monthly churn, while `codeclimate` (20.3%) and `codecov` (29.3%) are associated with discontinuous increases in monthly churn, all with significant negative post-intervention slopes.

Our model may be detecting a negative discontinuity for `coveralls` due to no controls for prior adopted tools. In other words, this negative effect may be due to projects which switch from, *e.g.*, `codecov` to `coveralls`. However, we find this unlikely; the number of projects which switch tools is low compared to those which stick to their initial choice. As stated in Section II-B, we have found instances in which developers explicitly state a lack of trust in `coveralls`, due to unreliability. One explanation could be that, upon adoption of `coveralls`, knowledgeable and experienced developers may be immediately driven away, as they have negative opinions of this tool. This potential departure of top developers would be reflected by lower monthly churn. The exact reasons behind this negative discontinuity would likely be best explained by, *e.g.*, a qualitative study, left to future work.

We see that all three coverage tools are associated with a discontinuous increase in monthly PRs (`coveralls` 0.289; `codeclimate` 0.124; `codecov` 0.452), with significant positive post-intervention slopes for `coveralls` (0.003) and `codecov` (0.032). This could be due to the additional overhead incurred immediately due to necessary changes in the code base to interface with a coverage reporter, and further issues down the line regarding coverage. In summary, our expectation is mostly supported by the data, with the exception of `coveralls` and monthly churn.

4) RQ$_{2.4}$: Dependency Managers vs. Monthly Churn and PRs: For monthly churn, surprisingly all significant fitted discontinuities are negative (`gemnasium` -15.5%, `david` -16.6%), and all significant post-intervention slopes are negative (`david` -0.90%, `bithound` -2.8%). We assumed that developers listen to dependency manager warnings, and alter relevant code as a result. However, dependency management tools may cause developers to suffer from *notification fatigue* [52], when they issue too many automated warnings. Such fatigue due to false alarms is a serious issue across many disciplines [84]–[86]. Thus, prior to adopting a dependency manager, developers may update dependencies appropriately. However, upon tool adoption, developers may experience notification fatigue, causing them to ignore dependency warnings and thus update dependencies less often than before.

For monthly PRs, we see positive discontinuity coefficients when significant (`david` 0.199, `bithound` 0.418), and significant positive post-intervention slopes (`gemnasium` 0.009, `david` 0.199, `bithound` 0.418). This increase in PRs may be due specifically to warnings brought forth by the tool, as expected. However, given potential notification fatigue, this increase in PRs may be due to individuals having issues with the new warning proliferation, submitting code to handle this new problem. Thus, we do not believe this increase in PRs fully supports our initial expectation. However, investigation of these findings brought the issue of notification fatigue to our attention, which may be what our models are actually detecting for coverage tools. We believe this newly formed hypothesis is worthy of further investigation.

5) RQ$_{2.5}$: All Tools vs. Monthly Issues: For all tools with significant discontinuities, we see positive effects, *i.e.*, a jump in the number of issues immediately post adoption. Among the linters, interestingly, the adoption penalty ranges from the heaviest, for `JSHint`, 0.279, to the lightest for `standardJS`, 0.053. For those two, the post-intervention slopes are significant and negative, indicating a decrease over time, and have similar values (`JSHint` -0.013, `standardJS` -0.010).

Among coverage tools and dependency managers, only two have significant discontinuities, `coveralls` 0.128 and `david` 0.138, respectively. For those two, their post-intervention slopes are significant, negative, and similar.

This result has nuanced implications. In all significant cases, tool adoption is associated with an immediate increase in monthly issues, which may be seen as a negative (especially for `JSHint` due to its magnitude). However, with the exception of `ESLint`, these tools are associated with negative post-intervention slopes, suggesting that this initial “cost” of increased issues turns into a benefit later. This may be due to the tool itself; *e.g.*, the use of a linter may decrease problems in the long run. Further (qualitative) research is needed to investigate the exact reason for this negative slope in time.

In general, our models suggest that tool choice may matter in terms of our outcomes of interest. In addition, though our models are observational in nature (being regressions), they provide practical knowledge - for each task class, effects are mostly in the same direction for each tool intervention. Thus, one can draw general conclusions across tools within task classes with respect to our modeled outcomes. The modeling methods presented can also be used in a predictive setting; a developer can provide monthly data for their own project(s) and identify potential future outcomes given our models.

There are very few instances of explicit discussion regarding tool choice in GitHub issues. Thus, we suggest that developers consider tool choice as a reasonable topic of discussion in their projects moving forward. Our qualitative analysis shows that developers often struggle with tool choice; decisions are often ad-hoc. Our models provide another data point to be used in their decision making process, which may be useful when coming to a conclusion as to which tool to implement.

Answer to RQ$_2$: *We find that there are measurable, but varied, associations between tool adoption and monthly churn, PRs, and unique authors for both immediate discontinuities and post-intervention slopes. We find that tools within a task class are associated with changes in outcomes in the same direction, with the exception of* `ESLint` *and* `coveralls` *for monthly churn. For issues, in all significant cases, tools are associated with a discontinuous increase in monthly issues; however, all significant post-intervention slopes are negative (decreasing issues over time). Regarding issue prevalence,* `standardJS`, `coveralls`, *and* `david` *stand out as tools with significant and negative post-intervention slopes. We find that some of our initial expectations regarding tool adoption are supported, while the investigation of others brought new hypotheses to light.*

C. RQ3: The Order of Tool Adoptions

To examine the potential effects of the sequence of tool adoptions, we build 3 additional models, described in Section IV-D. As there are 152 unique tool adoption sequences in our data and 161 significant comparisons, for sake of space, Fig. 4 depicts randomly selected comparison groups for each outcome of interest. Comparison groups are created by looking at tool adoption sequences of the same length which are permutations of each other; *e.g.*, `bithound`→`standardJS` vs. `standardJS`→`bithound`. Each bar is a significant estimated factor level coefficient, fit by the mixed-effects model described above; standard error bars are also shown.

In most cases, the direction of the effect is the same across comparison groups. In addition, in some cases, the difference between comparison groups is quite small, *e.g.*, `david`→`codecov`→`ESLint` and `david`→`ESLint`→`codecov` for the monthly authors model. However, multiple tool sequences have large relative differences, *e.g.*, `david`→`ESLint`→`codecov` and `codecov`→`ESLint`→`david`, with a percent change of 134.26% and absolute change of 0.32 monthly authors. Further, we see one case in which effects are in different directions for monthly log churn. There are 5 total significant cases in which tool sequences have effects in opposite directions. In summary, we find that there exists a significant effect of tool adoption order in 161 sequence comparisons, which could be explained by several reasons. Perhaps tool integration effort is lower for particular orders as compared to others; we gave the reasoning behind the case

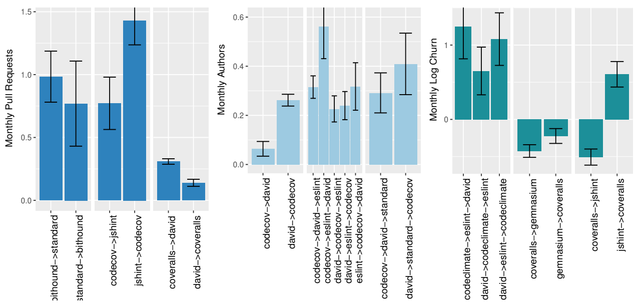

Fig. 4. Randomly selected significant fitted tool sequence factor comparisons. We compare tool sequences which are permutations of each other.

for $linter \rightarrow coverage$ *vs.* $coverage \rightarrow linter$ in Section III, and found a significant example in the data. We note, however, that other examples of this ordering do not show significant differences. Alternatively, projects which follow particular adoption sequences may be inherently different than projects which adopt the same tools, but in a different sequence, and these inherent differences are not adequately controlled. In any case, though effects are small in raw value, this result is noteworthy as it focuses the effects of tool adoption order, to our knowledge hitherto not investigated. Future work should follow this direction to find where such adoption sequences may matter for other software engineering outcomes.

Answer to RQ$_3$: *Some sequences of tool adoptions are more associated with changes in our outcomes of interest than others. We find that some tool adoption sequences, compared to others consisting of the same tools but in a different order, are associated with changes in opposite directions.*

VI. Conclusion

In this study we sought to learn if tool choice matters for similar projects that have adopted different quality assurance tools for the same task.

We expected to find much discussion of the pros and cons of different tools for the same task. On the contrary, very sparse discussion goes on on GitHub prior to tool adoption, perhaps because developers do not think it is worthwhile. On the other hand, we expected to find small if any differences in outcomes related to tool choice. But this intuition also did not fully pan out: we found that tool choice matters, and in some cases significantly, *e.g.*, `ESLint` and `coveralls`.

The statistical technology for analyzing longitudinal sequences of events also allowed us to consider and asses the effect of sequential adoption of tools from different categories. That in turn enabled us to compare adoption sequences of tools across task categories, and to identify sequences that associate with more beneficial effects than others. This technology paves the way for predictive tools to aid projects in improving or perfecting their automation pipelines, by offering bespoke advice based on analytics, as in this paper, from similarly contextualized projects. We encourage future work on this.

References

[1] S. W. Richard, "Organizations: Rational, natural, and open systems," *Aufl., Englewood Cliffs (NJ)*, 1992.

[2] T. E. Burns and G. M. Stalker, "The management of innovation," *University of Illinois at Urbana-Champaign's Academy for Entrepreneurial Leadership Historical Research Reference in Entrepreneurship*, 1961.

[3] J. Woodward, *Industrial organization: theory and practice*. Oxford University Press, 1965.

[4] Q. Wiki, "Are there any companies focused on doing devops consulting?" Jul. 2017. [Online]. Available: https://www.quora.com/Are-there-any-companies-focused-on-doing-DevOps-consulting

[5] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 358–368.

[6] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[7] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 805–816.

[8] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 197–207.

[9] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 70, 2015.

[10] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 307–319.

[11] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 255–265.

[12] S. Zhang and M. D. Ernst, "Which configuration option should I change?" in *Proc. International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 152–163.

[13] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft, and S. E. Condon, "Investigating and improving a COTS-based software development process," in *International Conference on Software Engineering (ICSE)*. IEEE, 2000, pp. 32–41.

[14] C. Abts, B. W. Boehm, and E. B. Clark, "COCOTS: A COTS software integration lifecycle cost model-model overview and preliminary data collection findings," in *ESCOM-SCOPE Conference*, 2000, pp. 18–20.

[15] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of software engineering*, vol. 1, no. 1, pp. 57–94, 1995.

[16] M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon, "COTS-based software development: Processes and open issues," *Journal of Systems and Software*, vol. 61, no. 3, pp. 189–199, 2002.

[17] D. Yakimovich, J. M. Bieman, and V. R. Basili, "Software architecture classification for estimating the cost of COTS integration," in *International Conference on Software Engineering (ICSE)*. ACM, 1999, pp. 296–302.

[18] J. Hartikainen, "A comparison of javascript linting tools," Mar. 2015. [Online]. Available: https://www.sitepoint.com/comparison-javascript-linting-tools/

[19] D. Sternlicht, "Thoughts about JavaScript linters and "lint driven development"," Aug. 2017. [Online]. Available: https://medium.com/@danielsternlicht/thoughts-about-javascript-linters-and-lint-driven-development-7c8f17e7e1a0

[20] K. F. Tómasdóttir, M. Aniche, and A. v. Deursen, "Why and how JavaScript developers use linters," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 578–589.

[21] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 84–94.

[22] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[23] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 470–481.

[24] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 334–344.

[25] papandreou, "Fixed accidentally global 'browserlist var." https://github.com/browserslist/browserslist/pull/7, January 2015.

[26] dasilvacontin, "add a more strict linter," https://github.com/janl/mustache.js/issues/433, March 2015.

[27] jack guy, "Feature: choose between eslint + airbnb and jshint?" https://github.com/feathersjs/generator-feathers/issues/112, April 2016.

[28] valorkin, "Add mocha to mocha environment," https://github.com/sindresorhus/globals/issues/34, January 2015.

[29] ——, "added all globals for full jshint compatibility," https://github.com/sindresorhus/globals/pull/17, January 2015.

[30] jasonkarns, "Easy require," https://github.com/sindresorhus/jshint-stylish/pull/20, May 2015.

[31] seanpdoyle, "Configure hound to lint with jscs," https://github.com/ember-cli/ember-cli/issues/5106, November 2015.

[32] jamesplease, "Consider switching to coveralls," https://github.com/babel/generator-babel-boilerplate/issues/409, September 2016.

[33] cscott, "Lint source code," https://github.com/CSSLint/parser-lib/pull/179, January 2016.

[34] magawac, "Stop using jscs," https://github.com/caolan/async/issues/1111, April 2016.

[35] ai, "Csslint + stylelint," https://github.com/CSSLint/csslint/issues/668, July 2016.

[36] Herst, "Update link to .jscrc file," https://github.com/Mottie/tablesorter/issues/1227, June 2016.

[37] martijnrusschen, "Switch to ESlint (what kind of tooling should we use for linters)," https://github.com/Hacker0x01/react-datepicker/issues/367, February 2016.

[38] rtablada, "Add eslint config for standard to project," https://github.com/poppinss/adonis-fold/pull/4, July 2016.

[39] emmby, "Use standard.js for code formatting," https://github.com/futurice/pepperoni-app-kit/issues/50, June 2016.

[40] henrjk, "feat(travis): add codeclimate integration," https://github.com/anvilresearch/connect/pull/275, October 2015.

[41] emmby, "Use standard.js for code formatting," https://github.com/futurice/pepperoni-app-kit/issues/50, June 2016.

[42] KrysKruk, "Add code coverage statistics," https://github.com/Neft-io/neft/issues/31, April 2016.

[43] yagpo, "Refactorized websocket api," https://github.com/bitfinexcom/bitfinex-api-node/pull/6, December 2015.

[44] isaacs, "Breaking change rfc: Remove codecov.io support," https://github.com/tapjs/node-tap/issues/270, June 2016.

[45] boneskull, "code coverage in ci," https://github.com/mochajs/mocha/issues/2351, July 2016.

[46] ariya, "Integrate codecov.io," https://github.com/jquery/esprima/issues/1215, June 2015.

[47] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.

[48] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 109–118.

[49] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" 2015.

[50] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[51] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem," in *International Conference on Software Engineering (ICSE)*. ACM, 2018.

[52] M.-A. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2016, pp. 928–931.

[53] ghaiklor, "Full refactoring," https://github.com/building5/sails-hook-bunyan/issues/9, October 2015.

[54] jhwohlgemuth, "Update minimatch version," https://github.com/nightwatchjs/nightwatch/pull/1184, September 2016.

[55] M. Sayagh, Z. Dong, A. Andrzejak, and B. Adams, “Does the choice of configuration framework matter for developers? empirical study on 11 Java configuration frameworks,” in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 41–50.

[56] H. Pham and X. Zhang, “Nhpp software reliability and cost models with testing coverage,” *European Journal of Operational Research*, vol. 145, no. 2, pp. 443–454, 2003.

[57] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[58] G. Gousios and D. Spinellis, “GHTorrent: GitHub’s data from a firehose,” in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 12–21.

[59] M. Beller, G. Gousios, and A. Zaidman, “TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration,” in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 447–450.

[60] R. H. Baayen, D. J. Davidson, and D. M. Bates, “Mixed-effects modeling with crossed random effects for subjects and items,” *Journal of Memory and Language*, vol. 59, no. 4, pp. 390–412, 2008.

[61] J. Cohen, *Applied multiple regression/correlation analysis for the behavioral sciences*. Lawrence Erlbaum, 2003.

[62] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[63] P. Bühlmann and S. Van De Geer, *Statistics for high-dimensional data: methods, theory and applications*. Springer, 2011.

[64] R. Hyman, “Quasi-experimentation: Design and analysis issues for field settings (book),” *Journal of Personality Assessment*, vol. 46, no. 1, pp. 96–97, 1982.

[65] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, “The impact of continuous integration on other software development practices: a large-scale empirical study,” in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 60–71.

[66] J. D. Singer and J. B. Willett, *Applied longitudinal data analysis: modeling change and event occurrence*. Oxford University Press, 2003.

[67] Z. Feng, T. Braun, and C. McCulloch, “Small sample inference for clustered data,” in *Proceedings of the Second Seattle Symposium in Biostatistics*. Springer, 2004, pp. 71–87.

[68] M. L. Bell and G. K. Grunwald, “Small sample estimation properties of longitudinal count models,” *Journal of Statistical Computation and Simulation*, vol. 81, no. 9, pp. 1067–1079, 2011.

[69] A. Gelman and J. Hill, *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

[70] J. C. Pinheiro and D. M. Bates, “Linear mixed-effects models: basic concepts and examples,” *Mixed-effects models in S and S-Plus*, pp. 3–56, 2000.

[71] A. Kuznetsova, P. B. Brockhoff, and R. H. Christensen, “lmertest package: Tests in linear mixed effects models,” *Journal of Statistical Software*, vol. 82, no. 13, pp. 1–26, 2017.

[72] S. Nakagawa and H. Schielzeth, “A general and simple method for obtaining r2 from generalized linear mixed-effects models,” *Methods in Ecology and Evolution*, vol. 4, no. 2, pp. 133–142, 2013.

[73] Y. Benjamini and Y. Hochberg, “Controlling the false discovery rate: a practical and powerful approach to multiple testing,” *Journal of the royal statistical society. Series B (Methodological)*, pp. 289–300, 1995.

[74] A. Tom, T. A. S. R. J. Bosker, and R. J. Bosker, *Multilevel analysis: an introduction to basic and advanced multilevel modeling*. Sage, 1999.

[75] R. Xu, “Measuring explained variation in linear mixed effects models,” *Statistics in medicine*, vol. 22, no. 22, pp. 3527–3541, 2003.

[76] H. Liu, Y. Zheng, and J. Shen, “Goodness-of-fit measures of r 2 for repeated measures mixed effect models,” *Journal of Applied Statistics*, vol. 35, no. 10, pp. 1081–1092, 2008.

[77] J. G. Orelien and L. J. Edwards, “Fixed-effect variable selection in linear mixed models using r2 statistics,” *Computational Statistics & Data Analysis*, vol. 52, no. 4, pp. 1896–1907, 2008.

[78] thiagogcm, “[suggestion] greenkeeper,” https://github.com/jhipster/generator-jhipster/issues/3159, March 2016.

[79] L. Hilsøe, “Linting javascript in 2015,” Jul. 2015. [Online]. Available: http://tech.lauritz.me/linting-javascript-in-2015/

[80] Slant, “Jshint vs. eslint,” 2018. [Online]. Available: https://www.slant.co/versus/8627/8628/~jshint_vs_eslint

[81] G. jazzband/django-model utils, “Use codecov instead of coveralls,” Jun. 2015. [Online]. Available: https://github.com/jazzband/django-model-utils/pull/175

[82] G. chaijs/chai, “Coveralls/coverage badge not working,” Feb. 2017. [Online]. Available: https://github.com/chaijs/chai/issues/927

[83] G. vavr-io/vavr jackson, “Move code coverage from coveralls.io to codecov.io,” Nov. 2015. [Online]. Available: https://github.com/vavr-io/vavr-jackson/issues/14

[84] M. Burdon, B. Lane, and P. von Nessen, “The mandatory notification of data breaches: Issues arising for australian and eu legal developments,” *Computer Law & Security Review*, vol. 26, no. 2, pp. 115–129, 2010.

[85] M. Cvach, “Monitor alarm fatigue: an integrative review,” *Biomedical Instrumentation & Technology*, vol. 46, no. 4, pp. 268–277, 2012.

[86] J. Welch, “An evidence-based approach to reduce nuisance alarms and alarm fatigue,” *Biomedical Instrumentation & Technology*, vol. 45, no. s1, pp. 46–52, 2011.