

Identifying Unusual Commits on GitHub

Raman Goyal^{*1}, Gabriel Ferreira², Christian Kästner² and James Herbsleb²

¹ *Indian Institute of Information Technology, Allahabad*

² *Carnegie Mellon University*

SUMMARY

Transparent environments and social-coding platforms as GitHub help developers to stay abreast of changes during the development and maintenance phase of a project. Especially, notification feeds can help developers to learn about relevant changes in other projects. Unfortunately, transparent environments can quickly overwhelm developers with too many notifications, such that they loose the important ones in a sea of noise. Complementing existing prioritization and filtering strategies based on binary compatibility and code ownership, we develop an anomaly-detection mechanism to identify unusual commits in a repository, that stand out with respect to other changes in the same repository or by the same developer. Among others, we detect exceptionally large commits, commits at unusual times, and commits touching rarely changed file types given the characteristics of a particular repository or developer. We automatically flag unusual commits on GitHub through a browser plugin. In an interactive survey with 173 active GitHub users, rating commits in a project of their interest, we found that, though our unusual score is only a weak predictor of whether developers want to be notified about a commit, information about unusual characteristics of a commit change how developers regard commits. Our anomaly-detection mechanism is a building block for scaling transparent environments. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: software ecosystems; notification feeds; information overload; transparent environments; anomaly detection

Copyright © 0000 John Wiley & Sons, Ltd.

1. INTRODUCTION

Collaborative development in open source, software ecosystems, and also industrial software systems relies increasingly on decentralized decision making [17, 20, 27, 41]. Interdependent components evolve independently and often with little explicit collaboration. Backward-incompatible changes that break modularity and produce rippling effects on downstream components are often necessary to avoid opportunity costs (not fixing mistakes, stifling change in the face of evolving requirements) and common in practice [10, 14, 19, 25, 29, 35, 38–40, 43, 47]. In addition, components may change to add new functionality that developers might want to adopt. Identifying relevant changes and reacting to them if needed can create a significant burden on developers during maintenance [3, 4, 6, 24, 35, 42, 47].

Seeds of a solution can be found in today’s *transparent environments* or *social-coding platforms* such as GitHub, LaunchPad, and Bitbucket. These environments provide mechanisms for notification and exploration, that help developers to stay abreast of activities across collections of projects without central planning [11, 12]. For example, on GitHub, developers can watch projects and receive a notification feed of activities in watched projects, such as push events or bug reports. These tools work well at small scales, but break down for large projects where imprecise and insufficiently rich notification mechanisms lead to *information overload* from notification cluttering. By inspecting publicly available events on GitHub, we found that active developers typically receive dozens of public event notifications a day and a single active project can produce over 100 notifications per day (and many more when including notifications of indirect dependencies). When we previously interviewed active GitHub users, many reported drowning in change notifications, for example stating “*I stopped with the email – now I use the GitHub notifications page. And the volume is a problem*” and “*I just wander through GitHub activity streams occasionally. [But] it is very much a crap shoot to actually get useful information from the feed*” [3, 11].

*Correspondence to: Journals Production Department, John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK.

A key to scale transparent environments is to identify relevant notifications and route them to affected or interested developers. There are many possible reasons why a change might be relevant for a developer, including the following explored in prior work:

- *Identify breaking changes:* Typically most changes are backward compatible. Notifications about the rare breaking changes are of especial importance to maintainers of affected downstream projects. Continuous integration platforms can help to highlight changes that break the system. In addition, Holmes and Walker designed a system that statically detects certain incompatible interface changes in Java to filter notifications correspondingly [24].
- *Identify critical fixes to vulnerabilities:* Patched vulnerabilities in upstream projects are typically of high importance to update the dependency to a newer version. The service *Gemnasium* tracks dependencies among Ruby packages and notifies registered package maintainers if an upstream dependency has a known vulnerability (CVE). In addition, several simple heuristics and learning approaches can identify bug fixing commits [22, 55].
- *Identify relevance based on prior activity:* In large code bases, developers may be interested only in notifications about code that relates to their own activities, such as notifications about changes in code that they have written. Padhye et al. model relevance based on simple heuristics regarding prior modifications, code ownership, and commit messages to similarly reduce information overload [42].

In this paper, we explore a different, complementary strategy to identify another class of relevant notifications:

- *Identify unusual changes:* We identify changes that are unusual or stand out with respect to other changes in the repository. For example, commits that are particularly large, changes to artifacts in a programming language not commonly used in the project or by that developer, or changes with exceptionally long commit messages might be worth noting. We developed an programming-language-independent anomaly detection mechanism that identifies outliers with regard to other changes in the same repository or other changes by the same developer.

We detect outliers using statistical models capturing common characteristics of commits within a project or by a developer. Based on those models, we provide an anomaly score for each commit. The anomaly score can be used to prioritize and filter notification feeds, in concert with other detection approaches, such as detecting breaking changes. In addition, anomaly scores can highlight unusual commits in the revision history to support exploration and inspection and to point out unusual characteristics during code reviews to focus the reviewer’s attention. We implemented a prototype of our anomaly detection mechanism and provide a frontend through a browser plugin that injects anomaly scores, including an explanation, into the commit history on GitHub pages.

In an evaluation, we analyze to what degree our model can predict changes developers will identify as unusual and to what degree we can identify commits about which developers want to be notified. We design an online survey with which participants rated commits in a repository of their choice. In each selected repository, we select five random commits with different anomaly scores (stratified sampling) and ask participants whether they judge the commit as unusual and whether they would want to be notified. We found that our unusual score only weakly reflects our participants’ notion of unusualness and is also only a weak predictor of whether developers want to be notified (to be expected as we capture only a subset of characteristics of important commits), but we also found that information about unusual characteristics about commits are actionable. When provided with additional information about why a commit is a statistical outlier, participants often revisited their position and identified commits as relevant for a notification.

Overall, we make the following contributions: (1) We design an anomaly model based on commit characteristics to identify unusual commits in a repository and by a developer. (2) We tailor statistical learning methods to build such models for Git repositories. (3) We integrate anomaly scores and explanations into the GitHub web page using an implementation based on a browser plugin. (4) We design an experimental setup to learn about the importance of unusual commits in a repository of the participant’s choice. (5) We evaluate our anomaly model with 173 GitHub developers, showing that despite weak predictive power, information about statistical outliers is actionable.

2. INDICATORS FOR IMPORTANT COMMITS

There are many reasons why a commit might be considered ‘unusual’ or important. In this work, we refer to commits as unusual if they are statistical outliers according to some criteria, such as commits that are substantially larger or were committed at an unconventional time of day. We intentionally used a broad and subjective term to cover a wide range of different outlier characteristics. Our mechanism is flexible enough to incorporate additional characteristics and select and weigh characteristics depending on developer preferences.

As part of a presurvey of our evaluation, which includes demographic questions about the participants’ experience and knowledge about the selected repository, we asked our participants (professional and academic GitHub users, see Sec. 4 for details) two open questions:

- *Some commits stand out among all commits in a repository. What characteristics make commits stand out?*
- *What kind of commits do you usually pay attention to?*

With both questions, we elicit commit characteristics that developers use to distinguish important commits from unimportant ones.

Among our participants, the following indicators for important commits were very commonly mentioned (at least by 30 developers):

- Commits that introduce new features (often associated with feature requests); for example, one participant claimed interest in “*commits that adds nice features to the project.*”
- Commits that signify major development steps, usually related to merging, milestones, and releases.
- Commits that are large in size (in terms of lines of code or files changed); for example, one participant wrote that changes stand out if they include “*extensive changes, lots of churn.*”
- Commits that fix bugs or security issues.
- Commits that change code about which the developers have particular knowledge or that could affect their current tasks (code ownership, dependencies).

- Commits with poor (nondescript, short) or overly complicated commit messages; for example, one participant expressed to pay particular attention to “*commits that have very detailed commit messages.*”

Many developers (at least 10) also mentioned the following indicators:

- Commits with lengthy and controversial discussions, including discussions on GitHub, in mailing lists, on IRC, and on social-media sites.
- Commits that perform major restructuring of systems or subsystems such as major shifts in an API, modifications to core functionality or abstractions, or changes affecting platform integration.
- Commits that are contributed by developers outside the regular team members or by inexperienced developers; in this case, the major concern is with the quality of the commit and, consequently, with the longevity of the project; for example, one participant expressed “*I look at the author, lines changed and description, in that order.*”
- Commits that break code, as identified by continuous integration tooling.

Other, but less frequent answers included:

- Commits addressing non-functional properties such as performance.
- Commits that affect essential project dependencies, potentially breaking compatibility.
- Commits made by the owner of the repository.
- Commits that use specific label, tags, or prefixes in their messages.

Overall, we can see that the reasons for paying closer attention to commits can vary widely.

Several facets can be addressed with simple checks (e.g., highlighting merge commits, identifying keywords in commit messages, counting comments on GitHub); others have already been addressed by alternative strategies (bug fixes, code ownership, breaking changes); but many indicators align well also with our strategy of modeling statistical outliers regarding commit characteristics, making it worth exploring those as a complementary detection mechanism. One can directly or indirectly relate statistical outliers to some of the indicators given by developers: for example, large commits

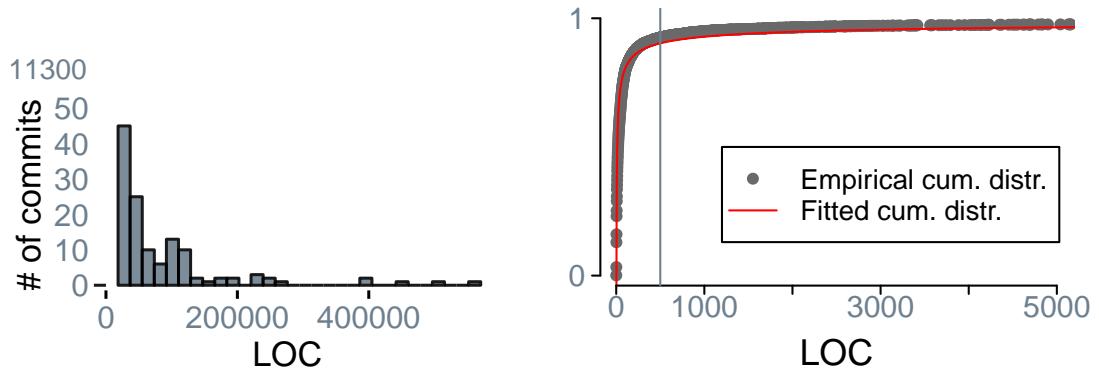


Figure 1. Commit sizes in lines of changed code in the *node.js* repository: (a) Histogram, (b) empirical and fitted cumulative distribution functions

(indicating also new features [23] or major restructuring), commits with short or long commit messages, changes in rarely changed files, and changes untypical for a developer.

3. DETECTING UNUSUAL COMMITS

3.1. Overview

We detect unusual commits as statistical outliers regarding various commit characteristics. That is, we compare characteristics of a specific commit to common characteristics of other commits in the same project or by the same developer. We build on a large body of work on anomaly detection [9,15,31,44], typically used for detecting inconsistent inputs, detecting credit-card fraud, and similar tasks.

Specifically, we build *profiles* that characterize statistical distributions of various commit characteristics (metrics). We learn concrete profiles from past commits and use the profiles to calculate an anomaly score for a new commit that describes how representative the commit is with regard to that profile. For example, we build a profile that describes the size of all past commits in a project (measured in lines of code), as the one for *node.js* shown in Figure 1b – from that distribution, we can learn that a commit with 500 changed lines of code is larger than 92 percent of all previous commits. We normalize and aggregate the anomaly scores from multiple profiles for various characteristics and derive an overall anomaly indicator. For each individual anomaly score, we can provide an explanation for why a commit receives a high score as exemplified in Figure 2.

v8: don't busy loop in cpu profiler thread

Reduce the overhead of the CPU profiler by replacing `sched_yield()` with `nanosleep()` in V8's tick event processor thread. The former only yields the CPU when there is another process scheduled on the same CPU.

Before this commit, the thread would effectively be a busy loop and consume 100% CPU time. By forcing a one nanosecond sleep period rounded up to the task scheduler's granularity (about 50 us on Linux), CPU usage for the processor thread now hovers around 10-20% for a busy application.

PR-URL: <https://github.com/joyent/node/pull/8789>
 Ref: <https://github.com/strongloop/strong-agent/issues/3>
 Reviewed-by: Trevor Norris <trev.norris@gmail.com>

bnoordhuis authored on 2014-11-27

1 parent fe20196 commit 6ebd85e10535dfa9181842fe73834e51d4d3e6c

Show Details

Use "Show details" button to show commit details.

ADDITIONAL INFORMATION FOR THIS COMMIT

- Changes were committed at **6am UTC** -- **bnoordhuis** **rarely** commits around that time. (fewer than **0.7%** of all commits by bnoordhuis are around that time)
- .gyp** files were changed -- such files are **rarely** changed in this repository. (fewer than **2%** of all file types changed)
- .cc and .gyp** files were changed in the same commit -- this combination of files is **rarely changed together**. (in fewer than **2%** of all commits)
- .cc and .gyp** files were changed in the same commit -- this combination of files is **rarely changed together** by **bnoordhuis**. (in fewer than **3%** of all commits by bnoordhuis)
- .gyp** files were changed -- such files are **rarely** changed by **bnoordhuis**. (fewer than **3%** of all file types changed by bnoordhuis)

Figure 2. Node.js commit with textual explanation about its unusualness

Based on the use case, we then rank the commits within a time frame by their anomaly indicator, filter commits in a notification feed with a given threshold, or merely show the indicator and explanation to developers.

In the following, we describe the analyzed characteristics, as well as the modeling and learning approaches used to build statistical models for these characteristics, and the derivation and aggregation of anomaly scores.

3.2. Commit Characteristics (Metrics)

While the measured commit characteristics can easily be extended, we selected 10 initial easy to measure characteristics regarding size of the change, size of the commit message, time committed, and changed file types. We summarize all measured commit characteristics in Figure I.

Size of the commit	Size of the commit message
Lines of code added	Length of the commit message (in words)
Lines of code removed	Time
Lines of code added or removed (sum)	Time of day of the commit (hours since midnight/UTC)
Number of files added	File types changed
Number of files removed	List of file types changed (file extensions)
	Number of files changed per file type

Table I. Commits characteristics

Size metrics measure the size of a change (a) in terms of added or removed lines, according to a text-based *diff* between the two revisions, and (b) in terms of files added, removed, or changed. Renamed files are considered both as removed or added in our current implementation. We use these simple size metrics, because very large commits tend to stand out as also indicated by participants in our study. Furthermore, Hindle et al. found in a manual investigation of commits that large commits tend to perform rare architectural changes and tend to signify perfective rather than corrective changes [23]. We specifically distinguish between adding and removing code, as they cover different change scenarios (e.g., new feature or cleanup) that may be more or less representative of change in a repository or by a developer.

We measure the size of the commit message in words. Our intuition is that exceptionally long commit messages tend to explain non-routine commits. At the same time, in projects with strong commit message discipline, short or missing messages may be considered as outliers.

We measure the time of day of a commit to detect commits at an unusual time (for that developer or that project). This might, for example, identify unusual late-night commits for a developer with regular working hours, which could be an indicator for an urgent change or a change under time pressure. Since we compare the times only against other commits by the same developer or in the same repository but not against external policies or expectations, we can ignore time zones and simply measure the time since midnight in UTC.

Finally, we track the types of files changed (as detected by their file extension) and the number of files changed per type in a commit. The intuition is that commits may change files of types that

are rare in that repository (such as .c files in a project focused on JavaScript) or that are rarely ever changed (such as license files). Similarly, for a developer typically working on .html and .css files, a change to the Java part of the repository might be unusual. We build profiles both with regards to the distribution of file types in the repository and the absolute and relative distribution of file types in typical commits.

3.3. Profiles

For each of the metrics described in Table I, we build one or more profiles. For most metrics, we build a profile per project and a profile per developer.[†] We skip developer profiles if the number of prior commits by that developer does not exceed a configurable threshold (set to 20 in our evaluation). Depending on the commit characteristics and their typical distributions, we use different kinds of statistical models with different learning and detection steps, as we will explain next.

Size-Based Profiles. Sizes of commits and commit messages tend to follow a long-tail distribution with most commits being fairly short and only few very large commits. For example, we show the distribution of commits in the *node.js* repository in Figure 1. As most commits are short, we only detect anomalies at the long end of the distribution. We sampled over 100 popular GitHub repositories and, after our evaluation, we also checked the commit sizes of the 173 repositories we investigated, confirming that all follow such long-tail distribution.

Given a learning set of commits, a simple profile could consist of an empirical cumulative distribution function (*ecdf*) that computes which percentage of commits is smaller than a given commit. We could then consider a commit larger than, say, 95 percent of all commits as anomalous. For example, we may learn in *node.js* that 97 percent of all commits in the learning set are smaller than 2770 changed lines of code (see Figure 1b) and thus report an anomaly score of 0.97 for a commit changing 2770 lines. However, an empirical cumulative distribution function is based exactly

[†]At this point, we build developer profiles per project, but it would easily be possible to build developer profiles based on their commits across multiple projects, e.g., collecting projects the developer contributes to from the developer's public events on GitHub.

on the learning set (danger of overfitting) and cannot distinguish between two commits that are both larger than the largest observed commit; for example commits with 600,000 and 6,000,000 changed lines would both receive the same anomaly score of 1 in *node.js*.

Instead, we abstract the actual observation by fitting a cumulative distribution function (*cdf*) that describes the observations in the learning set with few parameters. Specifically, we learn an exponential probability distribution with a *cdf* in the form

$$F(x) = 1 - \left(\frac{\alpha}{x}\right)^\beta$$

We estimated the parameters α and β using linear regression on the *ecdf* (technically encoded as $\log(1 - y) = \beta * \log(\alpha) - \beta * \log(x)$, which has the form $Y = A + BX$, such that we can learn A and B and subsequently α and β). In our *node.js* example, we learn the *cdf* $F(x) = 1 - (2.5597/x)^{0.44139}$ as plotted over the *ecdf* in Figure 1b, which yields two high but distinguishable anomaly scores 0.9957 and 0.9984 for 600,000 and 6,000,000 lines respectively.

All size-based profiles return an anomaly score of 0.5 for an average commit and values close to 0 and 1 for outliers. Due to the long-tail distribution, we focus only on outliers at the long end.

Time-Of-Day Profiles. To build a profile of typical times at which commits are created in a repository or by a developer, we build a histogram counting the number of commits per time of day (in 1 h intervals since midnight/UTC): function *tod*(*x*) returns the number of commits in that hour. We do not learn a function, but to avoid relying too much on noisy empirical data, we take a three hour average and compare it to all commits to get the probability of a commit at that hour:

$$p(x) = \frac{tod((x - 1) \bmod 24) + tod(x) + tod((x + 1) \bmod 24)}{3 \cdot \sum_{i=0}^{23} tod(i)}$$

If all commits were randomly distributed we would expect 1/24 of all commits per hour. As for size-based profiles, we want our anomaly score to report 0.5 for an average commit and 0 and 1 for outliers. Again, we only care about outliers on one end: regarding commits at times with few other commits. We therefore transform the probability value, such that the baseline 1/24 probability yields a 0.5 anomaly score:

$$F(x) = 1 - p(x)^{\log_{24} 2} = 1 - p(x)^{0.2181}$$

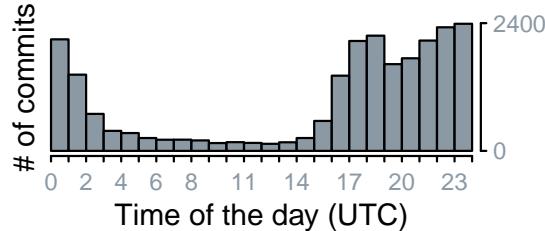


Figure 3. Time-of-day distribution of commits in the *atom* repository

Further adjustment is possible in the aggregation and normalization process to give larger weight to smaller outliers (see below).

In Figure 3, we show as example the time-of-day profile for the *atom* text editor repository. In this repository, we can observe fairly regular commit times. In the past only 0.7 percent of all commits were made around 10am UTC (3-hour average), yielding an anomaly score of 0.67, whereas commits around 11pm UTC are common and yield an anomaly score of 0.40.

File-Type Profiles. In line with Herraiz et al. [21], we observed that most repositories are dominated by files of few types, for example in *node.js*, 76 percent of commits in the repository affect C/C++, JavaScript, or HTML files, whereas Ruby files are only touched in 0.1 percent of all commits. We consider changes to file types that are rarely changed in the repository as unusual. We distinguish file types $t \in T$ by their file name's extension, but this could easily be refined by analyzing the files' content or merging different file extensions describing the same class of file types (e.g., .c and .h). We actually build multiple profiles based on file types, counting different aspects. Since we do not have a null hypothesis of how file types would be distributed across random commits (the set of file types T is open ended), we work directly with the empirical distributions and we do not have an expected anomaly score for an average commit.

In a first profile, we collect how frequently files of a given type were changed across all commits in a learning set. Function $ft(t)$ returns how many files of type t have been changed in a commit, summed over all commits in the learning set. We derive an anomaly score for each filetype based on the relative frequency of changes to this filetype in the repository as follows:

$$F(t) = 1 - \frac{ft(t)}{\sum_{t \in T} ft(t)}$$

For a given commit, we return the maximum anomaly score among all file types occurring in that commit.

In a second profile, we collect the percentage of commits that modify a file of a given file type. The anomaly score is simply the relative number of commits that do *not* modify a file of that type. That is, in contrast to the previous profile, we do not distinguish how often files have been changed in those commits. This profile is similar to the first, but less biased toward individual commits that change many files of one type. For example, with Ruby files occurring only in 0.1 percent of all commits, a Ruby file in a commit to the *node.js* repository would receive an anomaly score of 0.999.

In a third profile, we characterize which files are commonly changed together. This profile is created by grouping list of file types changed. We derive an anomaly score for every pair of file types ($F(t, u)$). The intuition is that certain kinds of files may be common in the repository but are rarely changed together, such as JavaScript and markdown files. The anomaly score is computed by comparing the number of commits that change both file types against the number of commits of the file type that is changed less often. For example, in *node.js* Javascript files are changed in 49 percent and markdown files in 12 percent of all commits, they are changed together in only 4 percent of all commits, yielding an anomaly score of $1 - 0.04/0.12 = 0.67$. Again, the highest score for any pair of file types contained in a commit is reported.

3.4. Normalizing, Aggregating, and Explaining Results

We described a number of profiles, and additional profiles can easily be added for other commit characteristics, including more sophisticated and language-dependent ones (e.g., conformity with the common vocabulary used in the project or with common syntactical structures [45]). It is also straightforward to learn profiles for other baseline sets, such as all commits of a developer (across multiple projects) or all commits within a set of projects. Which profiles to use and how to weigh them may be configured by personal preference. We design the anomaly detection system with a general normalization and aggregation framework that combines the anomaly scores of several

profiles into a single indicator. In addition, we provide facilities to explain the scores, which can help user acceptance in many usage scenarios.

The aggregation function should support two scenarios: On the one hand, if a single high anomaly score is found in the project, the aggregated anomaly indicator should have a high value. That is, it should not be possible to hide an anomaly regarding one characteristic with typical parameters of other characteristics. On the other hand, if several profiles indicate anomalous results, the aggregated indicator should be higher than the individual scores. To that end, instead of an arithmetic mean, we use an associative and commutative aggregation operator $\oplus : a \oplus b = a + b - (a * b)$, which supports both scenarios. Within the range $[0, 1]$, $a \oplus b \geq \max(a, b)$ and within the range $]0, 1[$, $a \oplus b > \max(a, b)$, i.e., two anomaly scores support each other. For missing profiles, e.g., when the developer has not sufficient prior commits to build a profile, we substitute the default anomaly score 0.5 of an average commit in the aggregation process.

To adjust individual profiles with weights α_i , we transform each anomaly score x_i using the transformation $1 - (1 - x_i)^{\alpha_i}$. We use this transformation, since it produces well distinguishable values especially for the high end range of anomaly scores between 0.95 and 1. In most profiles that gave high end range of anomaly scores, we *normalize* values using the transformation to emphasize that range. Without normalization, the aggregation operator \oplus leads to values close to 1 that are nearly indistinguishable, even for relatively low anomaly scores as 0.8. In our evaluation on the basis of preliminary analysis, we uniformly normalize all profiles using $\alpha = 0.067$ which maps anomaly scores in the relevant range to much lower values, e.g., 0.95 normalizes to 0.18, 0.999 to 0.37, and 0.99999 to 0.53.

While an overall normalized and aggregated anomaly indicator may be used, among others, to prioritize or filter notifications, in many scenarios an explanation may be even more useful than a numerical score. In addition to a numerical score, every profile can offer a textual explanation, such as “*.yml files were changed – such files are rarely changed in this repository (fewer than 0.07 % of all changes)*” or “*The number of files changed is in the normal range for commits by ;author; (3 files)*.”

In our prototype, as shown in Figure 2 and 4, we highlight the explanations from the five profiles that

have the highest individual anomaly scores. In addition, we can visualize the empirical distributions in the profiles through graphs as exemplified in Figures 1 and 3.

3.5. *Implementation*

We have implemented the anomaly detection mechanism as a web-based system with a frontend that injects results into the GitHub page through a browser plugin for Chrome.

The backend clones the repository and extracts characteristics of existing commits from the Git database (including running diffs on all commits). If the repository was previously cloned, it pulls new changes and collects the commit characteristics for those changes. Subsequently, it builds the profiles as described above and computes anomaly scores on demand. The backend is implemented as a Java servlet, interacting with Git through the JGit library and uses R for statistical computations.

The frontend is implemented as a browser plugin for Chrome that rewrites the GitHub pages within the web browser. When a user browses a GitHub page of a repository, the plugin queries the backend for anomaly indicators. On the commit history page of a project, the plugin injects the anomaly indicator and its explanation into the page for every commit, as shown in Figure 4. In a similar way, the plugin could rewrite the notification page and extend the settings page to customize weights and thresholds of the anomaly detection mechanisms. Alternatively, one can change the plugin design to present a ranking of unusual changes in a period. There are many different forms of presenting this to the user, but this discussion is outside the scope of this study.

Both backend and frontend are available as open-source project at github.com/goyalr41/UnusualGitCommit and github.com/goyalr41/UnusualCommitExtension.

4. EXPERIMENTAL VALIDATION

We anticipate that changes that are statistical outliers with respect to certain commit characteristics are relevant to developers. Our hypothesis is two-fold: we can reliably and efficiently use statistical outliers to detect unusual commits and developers want to be notified about unusual commits.

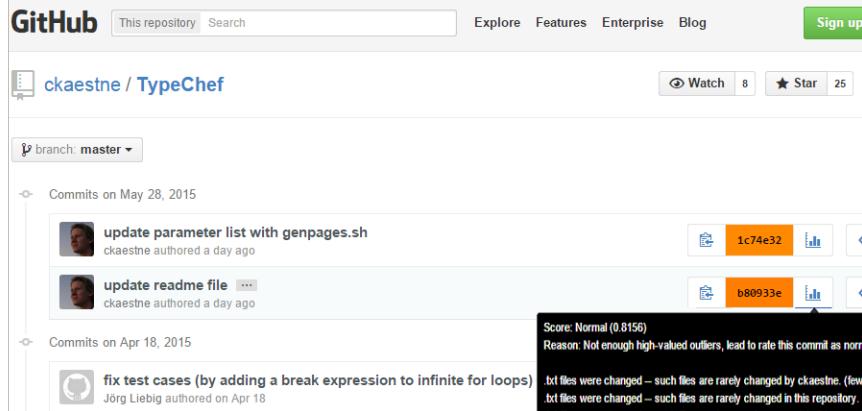


Figure 4. Anomaly scores injected into the Github history page using a browser plugin. Anomalies on commits are highlighted with background colors, tool tips show explanation, and a button links to additional information on our server.

With an interactive survey among 173 GitHub developers, we evaluate accuracy, usefulness, and practicality of our approach. Specifically, we investigate the following three research questions:

RQ1 *Does our approach identify commits that developers consider as ‘unusual’?* Our first goal is to identify whether the *developers’ notion an unusual commit* can be approximated with *our notion* of unusualness in terms of statistical models over commit characteristics. We expect that developers have a broad notion of what makes a commit unusual (see Sec. 2), but that the modeled commit characteristics play an important role.

RQ2 *Do developers want to be notified about unusual commits?* Our second goal is to identify whether high anomaly scores are a good measure to filter or prioritize notifications, that is, whether our anomaly scores highlight important commits and whether they are *actionable*. We will also investigate to what degree providing additional information about commit characteristics changes the developers’ perception and which commit characteristics they consider as important.

RQ3 *Can statistical outliers be computed efficiently?* Our final question aims to investigate practicality in everyday settings and measures the performance of our approach in terms of time required to build and evaluate our models.

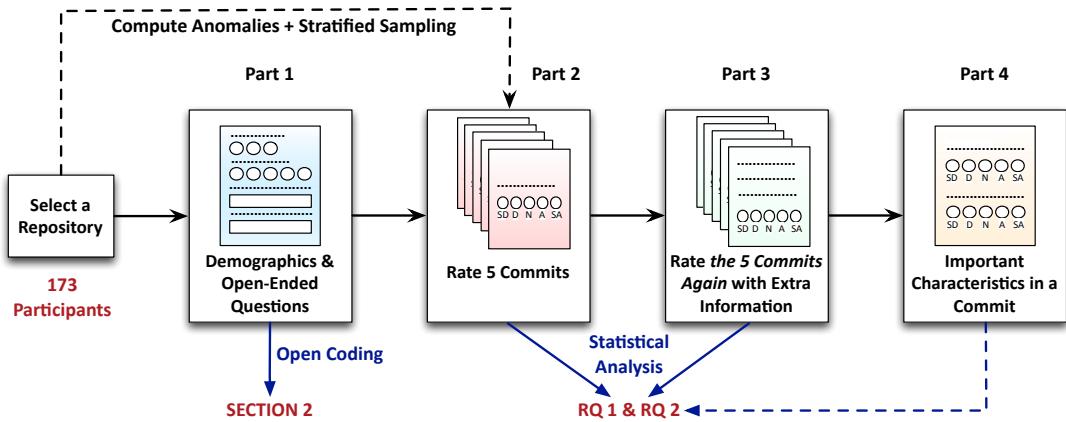


Figure 5. Experimental setup - survey steps

4.1. Experimental Setup

To answer our research questions, we conducted an experiment in form of an interactive online survey with GitHub developers as illustrated in Figure 5. In a nutshell, we ask each participant to select a GitHub repository of their choice and we select 5 commits from that repository spanning a range of anomaly scores about which we ask questions regarding unusualness and importance.

Our experiment intentionally deviates from the common setup with experimental and control groups. We assess how our participants judge the unusualness and importance of commits without revealing our judgment (no treatment). Subsequently, we ask our participants to assess the same commits again with additional anomaly information provided to identify how such information changes their view of the unusualness and importance of commits (within-subject design).

We complement that setup with a pre- and post-survey in which we ask additional questions about demographics and about which commit characteristics are relevant to them. We conducted the evaluation after a pilot run with 26 participants in which we ensured understandability and a reasonable length and narrowed down relevant questions, see below.

At the beginning of the survey, we asked our participants to choose a GitHub repository with which they are familiar, but to which they are not the main contributor. We suggest that the repository should have multiple contributors (two or more) and at least 50 commits. When they provided their GitHub username, we offered a selection of public repositories they watch. We designed these criteria

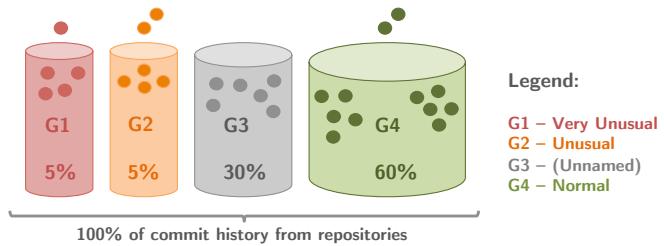


Figure 6. Stratified sampling - division of commits in groups of different degrees of unusualness

to simulate the scenario in which a developer might watch another repository during development or maintenance.

In the backend, we then clone that repository, build the profiles, and compute anomaly scores for up to 200 most recent commits. Among those recent commits, we select 5 commits with different anomaly scores (stratified sampling). We divide the commits into four groups by aggregated anomaly indicator: the 5 percent with the highest anomaly indicator, the 5 percent with the next highest anomaly indicator, the 60 percent with the lowest anomaly scores, and the remaining 30 percent. Using stratified sampling, we then selected one commit of our sample from the first group (very high anomaly indicator) as *very unusual*, two commits from the second group (high anomaly indicator) as *unusual*, and two commits from the third group (low anomaly indicator) as *normal* (see Figure 6). We decided to show a total of 5 commits, based on considerations regarding survey length and feedback from the pilot survey.

While cloning the repository and computing profiles in the backend (in part to bridge the time, typically below a minute, see RQ3 below), we show developers a presurvey with demographic questions, questions about their relationship to the repository (e.g., whether they are familiar with it, monitor it, or contribute to it) and open-ended questions regarding characteristics of commits that make commits to stand out and regarding to which kind of commits they tend to pay extra attention. We already discussed the key insights from the open-ended questions in Section 2.

In the main part of the survey, we show the sampled commits in a random order. We present each commit separately in a page that closely mirrors GitHub's view on a commit, followed by two questions about this commit's unusualness and importance on a 5-point Likert scale.

Q1a: This commit is important to me.

Q1b: I would want this commit to be brought to my attention.

Q2: It seems to be an unusual commit

Table II. Survey Questions

P1: I would want this commit to be brought to my attention.

P2: I would care about this commit.

P3: This commit is important.

P4: It seems to be an unusual commit

Table III. Survey Question in Pilot Study

Subsequently, we showed them each commit again in the same order, but this time with additional information in the form of the textual explanations (see Sec. 3.4) of the five highest anomaly scores as illustrated in Figure 2. We again ask questions about unusualness and importance on the same 5-point Likert scale to assess whether the additional information changes their opinion.

After completing the main part of the survey, we asked final questions about which commit characteristics (see Table I) they consider as useful indicators. We assess their opinion on a 5-point Likert scale.

Question Selection. We used the questions shown in Table II in our survey. When showing the commits for the first time, we asked questions Q1a and Q2 and when showing them again with additional information, we asked Q1b and Q2.

The selection of questions was influenced by several insights from the pilot study. Initially, we considered more questions on each commit as shown in Table III. We originally intended to capture multiple facets of unusualness, but found that pilot questions P1–P3 all strongly correlate and do not provide complementary insights. Therefore, we reduced the number of questions to two, corresponding to our two research questions. The strong correlation (P1 and P2 correlated strongly; correlation coefficient 0.67, $p < 3.1e - 19$) allowed us to ask two different questions to assess the

same information about importance of the commits, reducing a possible consistency bias in our participants (see also threats to validity).

4.2. Recruitment and Participants

As participants for our evaluation, we aimed to recruit active GitHub developers who could pick a project of their interest with multiple contributors and at least 50 commits. We recruited *active* GitHub developers on the web. We actively publicized our survey through social networks and sent emails to active GitHub developers. We identified suitable candidates from publicly available activity feeds (aggregated through the GHTorrent project [18]). We selected 39,131 developers that have filed or changed at least one pull request and that have started watching at least five repositories between January 2015 and August 2015. We sent personalized emails to 2846 of them (randomly selected). In total, 173 participants completed the survey, assessing 860 commits in 173 distinct repositories.

In Figure 7, we summarize the demographic information about our participants and their selected repositories, both presented to them as Likert scales. Over 63 percent have more than 5 years of software engineering experience and 62 percent report to use GitHub for over 3 years. In addition, 91 percent are familiar or very familiar with the selected project and 68 percent monitor commits in that project at least occasionally. The median number of commits and contributors of the analyzed repositories were 914 and 25, respectively. In addition, the repositories have median size of 414721 lines across all nonbinary files in the repository.

For the pilot study, we personally invited PhD students and researchers at a workshop on feature-oriented software development.[‡] We handed interested participants a tablet to complete our survey. In total, 26 participants completed the survey in our pilot study.

[‡]<http://www.fosd.de/meeting2015>

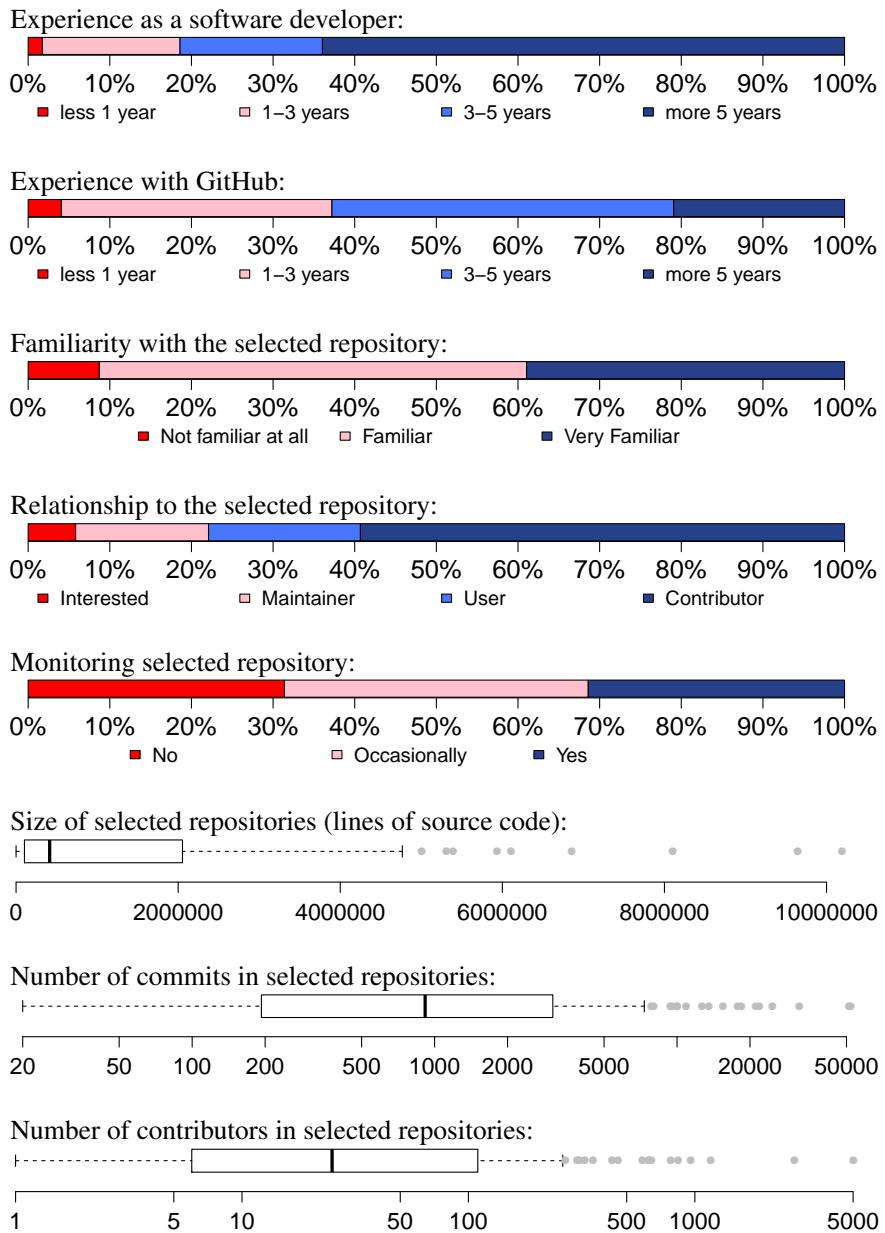


Figure 7. Demographics information about participants and repositories

4.3. Analysis Procedure

To answer our research questions, we analyzed 173 complete responses. Those do not include 21 incomplete responses or any responses from our pilot study. Since all our measures are ordinal, we use Spearman's rank correlation coefficient to assess correlations among ratings. For the open-ended questions discussed in Section 2, we followed standard open coding practices [54].

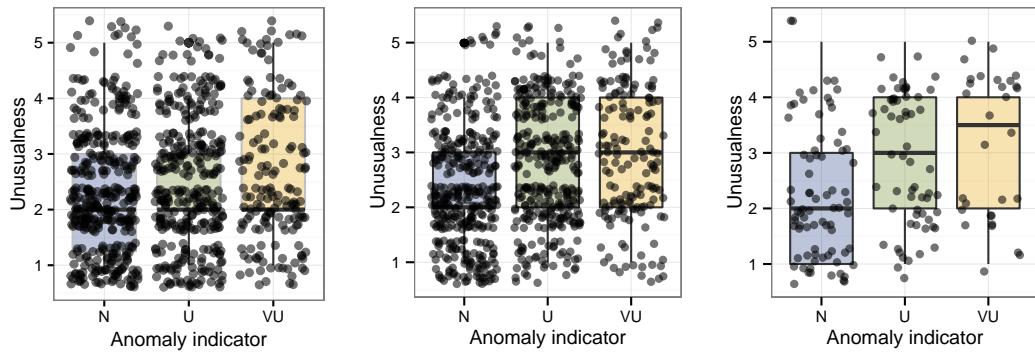


Figure 8. Comparison of agreement level between our notion and the developers' notion of unusualness. The x axis describes our anomaly judgment as normal (N; bottom 60 %), unusual (U; top 10 %), and very unusual (VU; top 5 %); the y axis describes the participants rating regarding unusualness from strongly disagree (1) to strongly agree (5) that the commit is unusual, resulting in 15 possible combinations. Data is jittered to show frequency of answers; boxplots are overlaid to highlight the distributions.

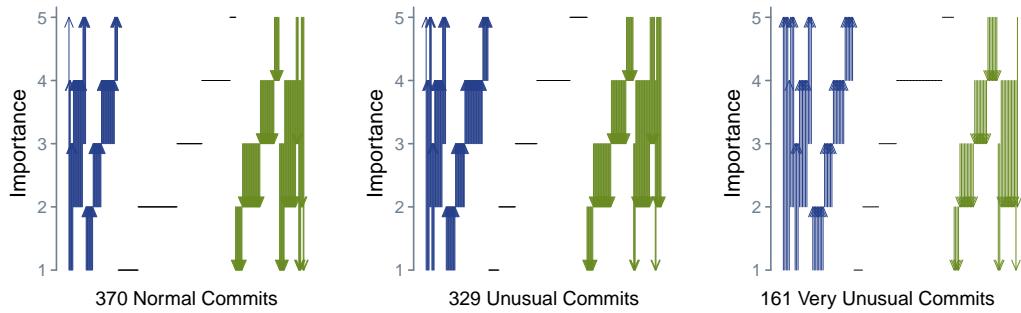


Figure 9. Changes in developer's perception of commits' importance: (a) normal commits, (b) unusual commits, (c) very unusual commits. Each plot shows the difference between two ratings (before and after revealing additional information, rated on strongly disagree (1) to strongly agree (5) that the commit is important) as arrow for all commits in a group of commits; commits are sorted by the change. Upward arrows indicate that participants judged the commit as more important with additional information; longer arrows indicate a larger change; dots indicate consistent ratings.

4.4. Results

RQ1: Predicting Unusual Commits. Regarding our first research question (*Does our approach identify commits that developers consider as 'unusual'?*), we found that our anomaly score is a very weak predictor of whether developers rate a commit as unusual (correlation coefficient 0.10, $p < 0.0033$). As visible in Figure 8a, our anomaly score correctly predicts many of the commits that

participants rate as highly unusual with high anomaly scores (high recall; 68 percent of commits that participants rate with 4 or 5, we classified as unusual or very unusual), but developers do not consistently rate commits with high anomaly scores as unusual (low precision; only 40 percent of commits we selected as highly unusual were rated as 4 or 5 by participants).

The low predictive power can be explained by the broad notion of ‘unusualness’ that reasonably includes many facets not covered by our models but that can be added from other sources, as discussed in Sec. 2. In fact, if we perform the same analysis only for the 38 participants (22 percent) that referred to some notion of size when asked about which commits stand out, we can predict unusual commits more reliably for those developers (correlation coefficient 0.27, $p < 0.00035$; Figure 8c).

Finally, we evaluated how showing developers additional information about commits influences their judgment of unusualness. We found that in general, developers tend to agree more with our notion of unusual given additional information as shown in Figure 8b, but the effect is again relatively weak (correlation coefficient 0.21, $p < 2.5e-10$).

In summary, our model of unusual commits is only a weak predictor of how developers rate ‘unusualness.’ It explains to a large degree size-related criteria and is a better predictor for the many developers who equate unusualness with size. When presented with additional information about commits, developers tend to agree more with our notion of unusualness.

RQ2: Unusualness to Filter Notification. Whereas RQ1 only investigated whether our notion of unusual matches the participants’ notion, RQ2 now investigates to what degree the results are actionable.

The initial result is negative again: Developers do not consider unusual commits as important. This holds both for our notion of unusualness in terms of our anomaly score (correlation 0.17; $p < 0.00000056$) and even for the developers’ own rating of unusualness (correlation 0.068; $p < 0.046$).

However, we found that information that explains unusual aspects of a commit (see Fig. 2) significantly influences developers opinions about a commit. That is, while being considered as unusual is not actionable, considering information about why a commit is a statistical outlier is

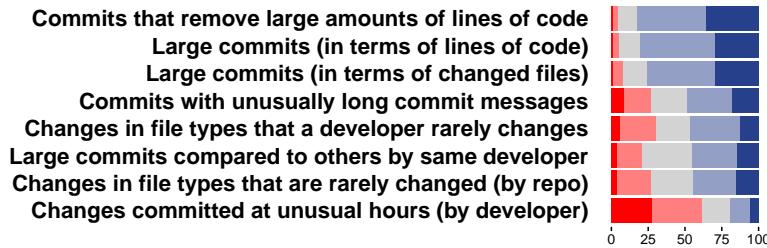


Figure 10. Developer's opinions about the usefulness of specific indicators

actionable. As shown in Fig. 9, with additional information, developers more frequently changed their judgment of importance to a higher value for commits we rated as very unusual, than they downgraded their rating. The number of changes are also significantly higher than what we observed in the pilot when we asked both questions for the same commit without additional information (correlation dropped as expected from 0.67 in the pilot to 0.48 in our study). Similarly, the developers unusualness ratings correlate stronger with their rating of importance given the additional information (correlation 0.37 instead of 0.068).

Finally, we asked developers which kinds of statistical outliers or indicators would be useful to them. Fig. 10 summarizes their opinion expressed using a 5-point Likert agreement scale, showing that the participants think that most of the profiles are useful; in fact only the time of day profile received controversial opinions.

In summary, developers consider some commits as unusual, but those are not necessarily commits that they consider to be important and want to be notified about. However, providing explanations for why commits are statistical outliers in the context of a repository or for a developer provides actionable insight in that developers often want to be notified about such outliers. Overall, this shows that the anomaly score is indeed just one among many indicators that signify whether a commit is important to developers (cf. Section 2). Nonetheless, the results also show that the anomaly score is actionable and can be a useful contributor to filtering and prioritizing notifications (in concert with other mechanisms), but that explanations are important and appreciated.

RQ3: Efficiency. Learning and evaluating profiles is relatively cheap. When learning a profile, we need to clone the repository and gather commit characteristics for past commits, which is the

Time required to compute profiles (in seconds):

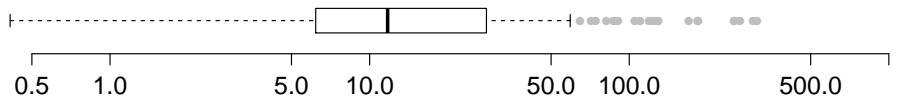


Figure 11. Time required to compute profiles for repositories in seconds (few outliers omitted)

dominant cost in the entire process. Building the profiles subsequently is fast; it involves computing histograms and linear regression and data sets of moderate size (20–7974 commits for 90 percent of all projects), whereas looking up the anomaly value for a commit requires merely evaluating simple formulas and is essentially instantaneous. Incrementally refining profiles over time is similarly fast, as we only need to pull new changes, collect commit characteristics for new commits, and update existing profiles (incrementally or recomputing them). To this day, we have created profiles for 252 GitHub projects. For almost all repositories, profiles can be built within one minute as shown in Figure 11. Only very few outliers for very large projects require longer initial computation time of up to 20 minutes for one 800MB repository of a game engine.

4.5. Exploratory Analysis

To understand more about how our profiles relate to the developers' notion of unusualness, we performed an additional ex-post exploratory analysis on our data. Specifically, we explore which profiles had the strongest predictive power in our data set, which may provide insights for further research. To that end, we use an automated model selection approach, specifically stepwise regression using the *gelmulti* R package.[§] The algorithm exhaustively computes logistic regression models while trying to minimize the set of profiles that maximize the explained variance of the outcome variable (developers' notion of unusualness). We use the profiles standardized scores from the past commit data and the developers judgment about each commit unusualness as input to the algorithm. We compute best models for their answers before and after we show them information about the unusualness of commits.

The best model for unusualness before showing additional information explains 7.06 percent of the variance ($p < 1.86e-11$) and is composed by six profiles (with decreasing predictiveness): number

[§]<https://cran.r-project.org/web/packages/gelmulti/>

of lines of code changed by developer, number of files changed (both at the repository level and by developer), number of files removed by developer, time of commit, and the combination of files changed. The strongest predictors, identified by the standardized regression coefficients, are: number of lines of code changed by developer and number of files changed.

The best model for unusualness after showing information explains 11.9 percent of the variance ($p < 2.2e-16$) and is composed from the following seven profiles: number of lines of code changed by developer, number of files removed by developer, size of commits messages, and the combination of files changed (the last two computed both at the repository level and by developer).

The exploratory results confirm that our profiles can explain some aspect of unusualness, and that especially size metrics are the strongest predictors. They also confirm that our profiles should not be used in isolation, but in concert with other mechanisms. This aligns with the results from our presurvey and also with prior research observations [33].

4.6. Threats to Validity

Construct Validity: Our current implementation of the profiles have limitations that threaten construct validity. First, we build developer profiles by grouping commits by email address; we currently do not take additional steps to identify when two email addresses belong to the same developer, which makes developer profiles potentially unreliable when the developer uses multiple email addresses [57]. Second, we currently consider renaming of a file as addition of one file and deletion of other, which can lead to size-based outliers for large renaming operations. Third, we ignore merge commits at this point. Fourth, in our evaluation, we used a simpler implementation of the time-based profiles in which we set a fixed high anomaly score for commits at times that on average contain fewer than 5 percent of all commits. Improving profiles, as well as additional profiles, will improve the anomaly reporting, but we expect little impact on the overall results of our evaluation.

Internal Validity: There are several biases that may influence the results. First, there is a selection bias in that developers who face problems with notification clutter may be more inclined to respond

to our survey. This most likely inflates the number of monitored repositories in our evaluation (see Figure 7), but we do not expect this to significantly impact ratings of unusualness and importance.

Second, as we ask developers to rate the same commit again with additional information there is both a potential consistency bias (participants tend to stick to their answers) and an opposite potential subject-expectancy bias (participants agree with suggestions). To reduce the consistency bias, we pose two separate but correlated questions Q1a and Q1b regarding importance. To reduce subject-expectancy bias, we always provide five explanations independent of our anomaly score. Using separate questions Q1a and Q1b introduces the additional noise in our analysis, despite the strong and statistically highly significant correlation between the two found in the pilot study. Although we addressed these biases in our setup, we cannot entirely exclude their influence.

Third, our results might be biased by the aggregation method used to combine multiple indicators. We kept it constant to keep the experimental design simple. It could be interesting to explore whether other aggregation methods provide better predictive results.

Fourth, the visual representation of commits with all lines of changed code may bias developers to focus overproportionally on size in their judgment. We did not quantify this effect, but rather decided on a single visual representation that aligns closely with GitHub's view of a commit.

Finally, the use of stratified sampling, with the majority of shown commits being unusual or very unusual by our measure (three out of five commits shown), may affect developers' perception of unusualness and affect their judgments about the commits.

External Validity: We studied only commits in publicly available GitHub projects as assessed by developers with a GitHub account. Although we expect generally similar characteristics, we cannot generalize our results to other version control systems and especially not to other development contexts with different cultures and expectations toward commits (e.g., end-user programming). In addition, to prevent overfitting of data our approach is limited to repositories with a given minimum size (50 commits in our evaluation) and developers with a given minimum number of commits (only developers with at least 20 commits were considered for own profiles).

5. RELATED WORK

Change in software systems has been studied, measured, and modeled intensively for many decades [8, 13, 14, 32, 37, 58, 59]. Change is inevitable; for example, Lehman postulated that software “undergoes continual changes or becomes progressively less useful” [32]. Instead of assuming stability or backward compatibility, transparent environments and social-coding platforms promote change awareness by making information about changes transparent [11, 12]. In practice and more traditional settings, developers often broadcast change announcements to others by email or through code reviews to achieve transparency [1, 13]. Examples of achieving transparency through notifications reach back at least to Brook’s descriptions of the OS/360 development [6, Ch. 7] Similarly, awareness mechanisms have been successful to notify developers about potential conflicts during concurrent development to seek collaborative solutions [2, 7, 51–53].

Relying on notifications in transparent environments can quickly lead to information overload though [3, 4, 6, 11]. Due to information overload important information may get lost in a sea of noise. Several researchers and practitioners have attempted to identify those notifications that are relevant to a specific developer. As described in the introduction, this includes detecting backward-incompatible changes in used APIs [24], prioritizing notifications based on code ownership [42], and highlighting commits that fix bugs or vulnerabilities [22, 55]. At a technical level, there are many strategies that can identify the impact (and thus importance) of changes, typically referred to as *change impact analysis* [5, 36, 46, 49]. Similarly, there are many techniques that can detect potentially defective changes, from static analysis [26, 50] to machine learning [30]. Reiss furthermore explored a technique to identify problematic code by detecting ‘unusual’ patterns in the AST when comparing the code to a large corpus of known code [45]. Our work addresses the same problem but from a fundamentally different perspective. We do not envision our anomaly detection tool as a standalone filtering mechanism, but as complementary building block to scale transparent environments. In addition, anomaly scores and explanations could easily be integrated into a code review process, similar to Google’s integration of static analysis tools [50].

The two previous approaches that directly target reducing notification overhead [24, 42] (based on binary compatibility and code ownership, see above) evaluate their approach only on proxy metrics without consulting developers, for example, approximating relevance by whether a developer has subsequently modified a file. In contrast, we designed an experimental setup that allows to assess to what degree automated judgments on commits actually match the developers' perception of importance for notifications by asking developers about commits relevant to them.

We use anomaly detection for a novel purpose. Chandola et al. provide a comprehensive overview of the anomaly detection field [9], which focuses on applications as diverse as intrusion detection, fraud detection, industrial damage detection, image processing, and traffic monitoring, and which uses techniques as diverse as machine learning for classification, nearest neighbor, and clustering as well as various statistical and information-theoretic approaches. Given the lack of labeled data (i.e., known anomalies), we use an unsupervised learning strategy based on both parametric and non-parametric standard statistical techniques. As we do not focus on anomalies among dimensions (e.g., commits with many changed lines but only few changed files), we detect anomalies separately for each dimension and aggregate them subsequently. If developers were willing to tag commits as unusual or important, there is a wide range of supervised machine-learning techniques that can be used to detect anomalies [9, 16].

Trude et al. interviewed GitHub developers about what kind of information should be summarized in a development activity feed. The developers consistently mentioned *unusual events* as being important [56]. In addition, they propose a dashboard to detect unusual events in commit histories and perform a preliminary evaluation with six interviews within a Brazilian company [33]. Their interviewees mention similar reasons for what makes commits unusual and their examples of unusual events align with the commits characteristics that we investigate in this paper. Our work complements theirs with a substantially different model for detecting anomalies (e.g., not assuming normal distributions, covering additional characteristics such as time of day) and a large-scale evaluation with 173 developers. At a technical level, we integrate our results in GitHub's interface with a browser plugin, instead of building a separate SVN-based dashboard.

In parallel to our research, Li et al. [34] investigate influential software changes and propose categories to identify them in an automated fashion using machine learning classification techniques. Although they propose similar metrics that could also help developers to reduce the amount of changes they need to inspect, their goal, study design, and technical approach are different from ours. First, their work aim at predicting influential changes, which they imply could affect many aspects of a system later. They claim that detecting these changes at commit time could support developers in making better-informed decisions about their potential effects before they happen. In our work, we focus on unusual commits and aim at reducing the amount of commits that developers have to review when changes occur. Second, both studies propose metrics based on intuition, but evaluate them differently. Their work uses a survey to evaluate categories of influential commits provided by the authors, while we survey developers to judge the unusualness of commits from repositories they are familiar with. Along with finding unusual commits, we also try to find which of those unusual commits are actionable. Finally, their machine learning model based on classifiers and cross-validation and aggregation methods are distinct from ours. In order to create their classifier, they tried out two standard techniques for classification tasks: Naïve Bayes and Random Forest. In contrast, our analysis based on established anomaly detection techniques [9, 15, 31, 44] aims at creating repository-specific profiles based on outliers from past commits and at using these profiles to calculate anomaly scores for commits in the repository. Our approach also provides human-understandable profiles and thresholds that are useful for explaining why commits are marked as unusual. In contrast, many machine learning techniques cannot provide the same explanatory evidence.

Rosen et al. [48] propose a tool named “Commit Guru” that provides developers and managers with risk information of their commits. Similar to our prototype, their tool rely on change measures to identify defect-inducing changes. Their measures have been validated against 11 large and long-lived projects in a previous work [28] and capture properties such as size, diffusion, purpose, history of changes, but also incorporates information about developers’ experience. Our work complements their study with a large-scale evaluation with 173 developers and reinforce the importance of some their studied measures in characterizing important changes.

6. CONCLUSION

Transparent environments and social-coding platforms support developers in staying abreast of changes in projects of interest. However, the amount of information produced can quickly overwhelm developers, making it harder for them to distinguish relevant changes from typical ones.

In this paper, we described an anomaly-detection mechanism designed to identify unusual commits in repositories. We evaluated our mechanism using a survey-based strategy to measure to what degree our model can predict changes developers judge as unusual and to what degree we can identify commits that developers want to be notified about.

We learned that developers have distinct motivations when judging the importance commits and statistical outliers have only little predictive power. However, once we present the reasons why these commits are unusual, developers often revisit their position and consider these commits as relevant for notification. Also, we found that some characteristics that developers pay attention to (e.g, commits with lengthy discussions or made by not regular team members), can be profiled in terms of statistical outliers and integrated to existing prioritization and filtering approaches to identify relevant changes in maintenance tasks. Our anomaly-detection mechanism is a building block in scaling transparent environments.

REFERENCES

1. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 712–721. IEEE Computer Society, 2013.
2. J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: A visual dashboard for fostering awareness in software teams. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 1313–1322. ACM Press, 2007.
3. C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proc. ASE Workshop Software Support for Collaborative and Global Software Engineering*, 2015.
4. C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2016.

5. S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.
6. F. P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, Boston, MA, anniversary edition, 1995.
7. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE '11, pages 168–178. ACM Press, 2011.
8. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kriesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, Sept. 2005.
9. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15:1–15:58, July 2009.
10. B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, page 55. ACM Press, 2012.
11. L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 1277–1286. ACM Press, 2012.
12. L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Leveraging transparency. *IEEE Software*, 2013.
13. C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 241–250. ACM Press, 2008.
14. S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng. (TSE)*, 27(1):1–12, Jan 2001.
15. E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proc. Int'l Conf. Machine Learning (ICML)*, pages 255–262. Morgan Kaufmann Publishers Inc., 2000.
16. P. Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.
17. A. Gawer. The organization of technological platforms. *Research in the Sociology of Organizations*, 29:287–296, 2010.
18. G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *Proc. Working Conf. Mining Software Repositories (MSR)*, pages 12–21, 2012.
19. M. Hammad, M. L. Collard, and J. I. Maletic. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Journal*, 19(1):35–64, 2011.
20. J. Herbsleb, C. Müller-Birn, and W. B. Towne. The VistA ecosystem: Current status and future directions. Technical Report CMU-ISR-10-124, Institute for Software Research, Carnegie Mellon University, Oct. 2010.
21. I. Herraiz, D. German, and A. E. Hassan. On the distribution of source code file sizes. In *International Conference on Software and Data Technologies*, 2011.

22. A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classification of large changes into maintenance categories. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 30–39. IEEE Computer Society, 2009.
23. A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proc. Working Conf. Mining Software Repositories (MSR)*, pages 99–108. ACM Press, 2008.
24. R. Holmes and R. J. Walker. Customized awareness: Recommending relevant external change events. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 465–474. ACM Press, 2010.
25. D. Hou and X. Yao. Exploring the intent behind API evolution: A case study. In *Proc. Working Conf. Reverse Engineering (WCRAE)*, pages 131–140. IEEE Computer Society, 2011.
26. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *ACM SIGPLAN Notices*, pages 132–136. ACM Press, 2004.
27. M. Iansiti and R. Levien. *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability*. Harvard Business Press, Boston, MA, 2004.
28. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Softw. Eng. (TSE)*, 39(6):757–773, 2013.
29. P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 726–738. ACM Press, 2010.
30. S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng. (TSE)*, 34(2):181–196, 2008.
31. Y. Kou, C.-T. Lu, S. Sirwongwattana, and Y.-P. Huang. Survey of fraud detection techniques. In *Proc. Int'l Conf. Networking, Sensing and Control*, volume 2, pages 749–754 Vol.2, 2004.
32. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980.
33. L. Leite, C. Treude, and F. F. Filho. Uedashboard: Awareness of unusual events in commit histories. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 978–981. ACM Press, 2015.
34. D. Li, L. Li, D. Kim, T. F. Bissyandé, D. Lo, and Y. L. Traon. Watch out for This Commit! A Study of Influential Software Changes. *Computing Research Repository (CoRR)*, abs/1606.03266, 2016.
35. M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 477–487. ACM Press, 2013.
36. J. Loyall and S. Mathisen. Using dependence analysis to support the software maintenance process. In *Proc. Conf. Software Maintenance (CSM)*, pages 282–291. IEEE Computer Society, 1993.
37. N. H. Madhavji. Environment evolution: The Prism model of changes. *IEEE Trans. Softw. Eng. (TSE)*, 18(5):380–392, May 1992.
38. M. Mattsson and J. Bosch. Stability assessment of evolving industrial object-oriented frameworks. *Journal of Software Maintenance: Research and Practice*, 12(2):79–102, 2000.
39. T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE Computer Society, 2013.

40. Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proc. Workshops Principles of Software Evolution (IWPSE) and Software Evolution (Evol)*, IWPSE-Evol '09, pages 57–62. ACM Press, 2009.
41. L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, 2006.
42. R. Padhye, S. Mani, and V. S. Sinha. Needfeed: Taming change notifications by modeling code relevance. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, 2014.
43. S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 378–387. IEEE Computer Society, 2012.
44. O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 302–312. ACM Press, 2002.
45. S. P. Reiss. Finding unusual code. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 34–43. IEEE Computer Society, 2007.
46. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 432–448. ACM Press, 2004.
47. R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 56:1–56:11. ACM Press, 2012.
48. C. Rosen, B. Grawi, and E. Shihab. Commit Guru: Analytics and Risk Prediction of Software Commits. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 966–969, 2015.
49. G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 6(2):173–210, Apr. 1997.
50. C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2015.
51. A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 94–103. ACM Press, 2007.
52. A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 444–454. IEEE Computer Society, 2003.
53. A. Sarma, D. F. Redmiles, and A. van der Hoek. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Trans. Softw. Eng. (TSE)*, 38(4):889–908, 2012.
54. M. Schreier. *Qualitative Content Analysis in Practice*. SAGE Publications, 2012.
55. Y. Tian, J. Lawall, and D. Lo. Identifying Linux bug fixing patches. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 386–396. ACM Press, 2012.

56. C. Treude, F. F. Filho, and Uirá Kulesza. Summarizing and measuring development activity. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 625–636. ACM Press, 2015.
57. B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and Tenure Diversity in GitHub Teams. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 3789–3798. ACM Press, 2015.
58. D. M. Weiss and V. R. Basili. Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Trans. Softw. Eng. (TSE)*, 11(2):157–168, Feb. 1985.
59. S. S. Yau and J. S. Collofello. Some stability measures for software maintenance. *IEEE Trans. Softw. Eng. (TSE)*, 6(6):545–552, Nov. 1980.