Keeping Master Green at Scale

Sundaram Ananthanarayanan
Uber Technologies

Masoud Saeida Ardekani
Uber Technologies

Denis Haenikel
Uber Technologies

Balaji Varadarajan
Uber Technologies

Simon Soriano
Uber Technologies

Dhaval Patel
Uber Technologies

Ali-Reza Adl-Tabatabai
Uber Technologies

Abstract

Giant monolithic source-code repositories are one of the fundamental pillars of the back end infrastructure in large and fast-paced software companies. The sheer volume of everyday code changes demands a reliable and efficient change management system with three uncompromisable key requirements — always *green* master, high throughput, and low commit turnaround time. *Green* refers to a master branch that always successfully compiles and passes all build steps, the opposite being *red*. A broken master (red) leads to delayed feature rollouts because a faulty code commit needs to be detected and rolled backed. Additionally, a red master has a cascading effect that hampers developer productivity—developers might face local test/build failures, or might end up working on a codebase that will eventually be rolled back.

This paper presents the design and implementation of SubmitQueue. It guarantees an always green master branch at scale: all build steps (e.g., compilation, unit tests, UI tests) successfully execute for every commit point. SubmitQueue has been in production for over a year, and can scale to thousands of daily commits to giant monolithic repositories.

ACM Reference Format:
Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. 2019. Keeping Master Green at Scale. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303970

1 Introduction

Giant monolithic source-code repositories (monorepos hereafter) form one of the fundamental pillars of backend infrastructure in many large, fast-paced software companies. Recent studies [27, 38] have shown that the monolithic model of source code management results in simplified dependency management, unified versioning, ease of collaboration, improved code visibility, and development velocity.

As shown by Perry et al. [37], concurrently committing code changes by thousands of engineers to a big repository, with millions of lines of code, may lead to a master breakage (e.g., compilation error, unit test failure, integration test failure, and unsignable artifact). Tracking down and rolling back the faulty change is a tedious and error-prone task, which very often needs human intervention. Since developers may potentially develop over a faulty master (mainline hereafter), their productivity may get hampered substantially. Additionally, a breakage in the master can cause delays in rolling out new features.

For example, prior to the launch of a new version of our mobile application for riders, hundreds of changes were committed in a matter of minutes after passing tests individually. Collectively though, they resulted in substantial performance regression, which considerably delayed shipping the application. Engineers had to spend several hours bisecting the mainline in order to identify a small list of changes that caused performance regressions. New changes further aggravated the problem such that the mainline required blocking while the investigation continued.

To prevent the above issues, the monorepo mainline needs to remain green at all times. A mainline is called green if all build steps (e.g., compilation, unit tests, UI tests) can successfully execute for every commit point in the history. Keeping the mainline green allows developers to (i) instantly release new features from any commit point in the mainline, (ii) roll back to any previously committed change, and not necessarily to the last working version, and (iii) always develop against the most recent and healthy version of the monorepo.

To guarantee an always green mainline, we need to guarantee serializability among developers' requests called *changes*. Conceptually, a change comprises of a developer's code patch padded with some build steps that need to succeed before the patch can be merged into the mainline. Therefore, committing a change means applying its patch to the mainline HEAD only if all of its build steps succeed. Observe that totally ordering changes is different from totally ordering

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03.
https://doi.org/10.1145/3302424.3303970

code patches, which a conventional code management system (e.g., git server) does. While the latter can still lead to a mainline breakage, totally ordering changes guarantees a green mainline since it ensures that all build steps succeed before committing the patch to the mainline.

In addition to ensuring serializability, the system also needs to provide SLAs on how long a developer would need to wait before her changes are merged into the mainline, called turn-around time. Ideally, this has to be small enough that engineers are willing to trade speed for correctness (i.e., always green mainline).

Ensuring the above two properties at scale is challenging. The system needs to scale to thousands of commits per day while it may take hours to perform all build steps. For example, performing UI tests of an iOS application requires compiling millions of lines of code on Mac Minis, uploading artifacts to different iPhones, and running UI tests on them. Additionally, the emergence of bots that continuously generate code (e.g., Facebook's Configurator [42]), further highlights the need for a highly scalable system that can process thousands of changes per day. Thus many existing solutions try to only alleviate, rather than eliminate, the aforementioned issues by building tools [38, 47] for rapid detection and removal of faulty commits.

This paper introduces a change management system called SubmitQueue that is responsible for continuous integration of changes into the mainline at scale while always keeping the mainline green. Based on all possible outcomes of pending changes, SubmitQueue constructs, and continuously updates a speculation graph that uses a probabilistic model, powered by logistic regression. The speculation graph allows SubmitQueue to select builds that are most likely to succeed, and speculatively execute them in parallel. Our system also uses a scalable conflict analyzer that constructs a conflict graph among pending changes. The conflict graph is then used to (1) trim the speculation space to further improve the likelihood of using remaining speculations, and (2) determine independent changes that can commit in parallel. In summary, this paper makes the following contributions:

- Based on our experience, we describe the problem space and real challenges of maintaining large monorepos, and why it is crucial to keep the mainline green all the time.
- We introduce our change management system called SubmitQueue, and detail its core components. The main novelty of SubmitQueue lies in picking the right set of techniques, and applying them together for building a highly scalable change management system.
- We evaluate SubmitQueue, compare it against similar approaches, and report how SubmitQueue performs and scales in production.

The outline of this paper is as follows: we first review background work in the next section. Section 3 explains the development life cycle, and gives a high level overview of SubmitQueue. We explain our speculation model and conflict analyzer in Section 4 and Section 5, respectively. Section 6 explains how SubmitQueue performs build steps. SubmitQueue implementation and evaluation are discussed in Section 7 and Section 8. We review additional related work in Section 9. We discuss limitations and future work in Section 10, and conclude the paper in Section 11.

2 Background

2.1 Intermittently Green Mainline

The trunk-based development approach typically commits a code patch into the mainline as soon as it gets approved, and passes pre-submission tests. A change management system then launches an exhaustive set of tests to make sure that no breakage exists, and all artifacts can be generated. In case an error is detected (e.g., due to a conflict or a failing test case), the system needs to roll back the committed change. However, since several patches could have been committed in the meantime, detecting and reverting the faulty patch is tedious and error-prone. In many cases, build sheriffs and engineers need to get involved in order to cherry-pick a set of patches for roll backs. Besides, a red mainline has the following substantial drawbacks:

- Cost of delayed rollout: for many companies, the cost of delaying rollouts of new features or security patches, even for a day, can result in substantial monetary loss.
- Cost of rollbacks: in case an issue arises with a release, engineers can only roll back to the last working version, and not any arbitrary version.
- Cost of hampered productivity: a red mainline can significantly affect productivity of engineers. At the very least, engineers might face local test and build failures. Even worse, they can end up working on a codebase that will eventually be rolled back.

To alleviate the above issues, software companies try to minimize post-submit failures by building pre-submit infrastructures (e.g., Google's presubmit infrastructure [38]), which tests patches before committing them to the mainline. Pre-submit testing approach solely focuses on individual changes, and does not consider concurrent changes. Therefore, testing infrastructures still need to (1) perform tests (e.g., integration tests) on all affected dependencies upon every commit, (2) quickly identify patches introducing regressions [47], and (3) ultimately roll back the faulty patch in case of a breakage. These steps are commonly referred to as post-submit testing.

Figure 1 shows the probability of real conflicts as the number of concurrent (i.e., pending) and potentially conflicting changes increases for our Android and iOS monorepos over the course of nine months. Roughly speaking, n concurrent changes are considered potentially conflicting if they touch the same logical parts of a repository. On the other hand, n

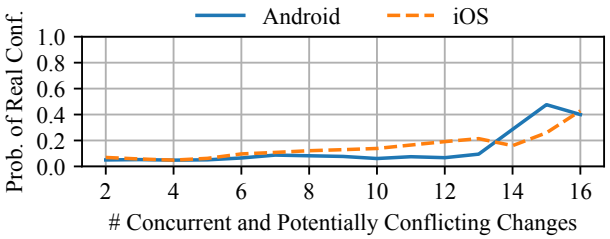


Figure 1. Probability of real conflicts as the number of concurrent and potentially conflicting changes increases.

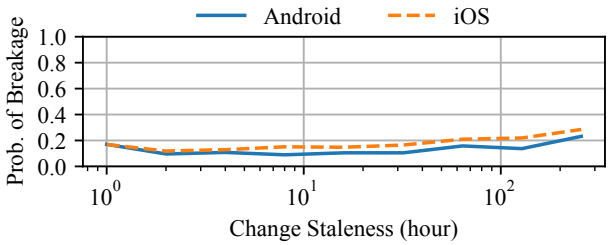


Figure 2. Probability of a mainline breakage for iOS/Android monorepos as change staleness increases.

concurrent changes have real conflicts if all the following conditions hold: (1) applying changes 1 to $n-1$ do not lead to a mainline breakage, (2) applying the n^{th} change individually does not break the mainline, and (3) applying all the changes together results in a master breakage. Consequently, the n^{th} change must have a real conflict with one or more of 1 to $n-1$ changes. Therefore, pre-submit tests cannot detect these real conflicts, and they can only be captured post-submit. Observe that with even two concurrent and potentially conflicting changes, there is a 5% chance of a real conflict. This number grows to 40% with only 16 concurrent and potentially conflicting changes. This shows that, despite all the efforts to minimize mainline breakages, it is very likely that the mainline experiences daily breakages due to the sheer volume of everyday code changes committed to a big monorepo.

Using data collected over one year, Figure 2 plots probability of a mainline breakage as change staleness increases. Change staleness is measured as how old a submitted change is with respect to the mainline HEAD at the time of submission. As expected, the probability of a mainline breakage increases as change staleness increases. However, observe that even changes with one to ten hour staleness have between 10% to 20% chance of making the mainline red. Thus, while frequent synchronization with the mainline may help avoid some conflicts, there is still a high chance of a breakage.

2.2 Always Green Mainline

The simplest solution to keep the mainline green is to enqueue every change that gets submitted to the system. A change at the head of the queue gets committed into the mainline if its build steps succeed. For instance, the rust-project [9] uses this technique to ensure that the mainline remains healthy all the time[2]. This approach does not scale as the number of changes grows. For instance, with a thousand changes per day, where each change takes 30 minutes to pass all build steps, the turnaround time of the last enqueued change will be over 20 days.

An obvious solution for scalability limitations of a single queue is to batch changes together, and commit the whole batch if all build steps for every change in the batch succeed. Otherwise, the batch needs to get divided into smaller batches, and get retried. As the batch size grows, which is the case when there are thousands of commits every day, the probability of failure (e.g., due to a conflict) increases significantly. This leads to batches getting divided and retried frequently. Therefore, SLAs cannot be guaranteed as they would vary widely based on how successful changes are as well as the incoming rate of changes. In addition, this approach also suffers from scalability issues since we still need to test and build one unit of batch at a time.

Chromium uses a variant of this approach called Commit Queue [4] to ensure an always green mainline. In Commit Queue, all pending changes go through two steps before getting merged into the mainline. The first step is called pre-commit queue where the system tries to compile a change. Changes passing the first step are picked every few hours by the second step to undergo a large suite of tests which typically take around four hours. If the build breaks at this stage, the entire batch gets rejected. Build sheriffs and their deputies spring into action at this point to attribute failures to particular faulty changes. Non-faulty changes, though, need to undergo another round of batching, and can only get accepted if no faulty change exists in a round. This approach does not scale for fast-paced developments as batches often fail, and developers have to keep resubmitting their changes. Additionally, as mentioned earlier, finding faulty changes manually in large monorepos with thousands of commits is a tedious and error-prone process, which results in long turnaround times. Finally, observe that this approach leads to shippable batches, and not shippable commits. Thus, a second change that fixes an issue of the first change must be included in the same batch in order for the first change to pass all its tests. Otherwise, the first patch will never succeed.

Optimistic execution of changes is another technique being used by production systems (e.g., Zuul [12]). Similar to optimistic concurrency control mechanisms in transactional systems, this approach assumes that every pending change in the system can succeed. Therefore, a pending change starts performing its build steps assuming that all the pending changes that were submitted before it will succeed. If a

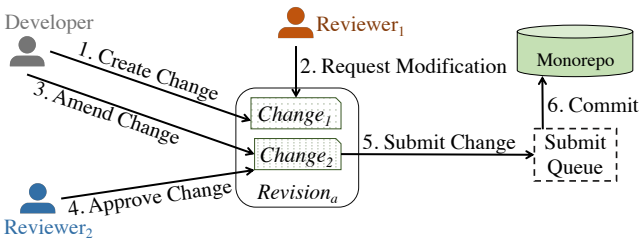


Figure 3. Development Life Cycle

change fails, then the builds that speculated on the success of the failed change needs to be aborted, and start again with new optimistic speculation. Similar to the previous solutions, this approach does not scale and results in high turnaround time since failure of a change can abort many optimistically executing builds. Moreover, abort rate increases as the probability of conflicting changes increase (Figure 1). In Section 8, we will empirically evaluate the performance of the solutions explained in this section.

3 Dev Life Cycle & System Overview

In this section, we first briefly introduce the development life cycle, and define the terms that will be used throughout the paper. We then give a high level overview of SubmitQueue, and present its architecture.

3.1 Development Life Cycle

Figure 3 presents the life cycle of our code development. A developer starts coding by creating a feature branch from the mainline's last commit point, called HEAD. She continuously commits her code locally. Once the developer finishes, she submits her code for reviews. This results in creating a revision along with a change. Conceptually, a change comprises of a code patch, required build steps along with some metadata, including author and change description. A revision is simply a container for storing multiple changes. Depending on code reviews, a developer may further modify her code, and amend her changes until a change gets approved. Once a change is approved, a developer *submits* (or lands) the change to SubmitQueue.

SubmitQueue guarantees that the change's patch gets merged into the mainline's most recent HEAD if upon applying the patch, all build steps succeed and mainline remains green. For instance, a build step can involve unit tests, integration tests, UI tests, or generating artifacts. A change is called landed (or committed), once its corresponding patch gets committed into the mainline's HEAD.

3.2 SubmitQueue Overview

SubmitQueue guarantees an always green mainline by providing the illusion of a single queue where every change gets enqueued, performs all its build steps, and ultimately

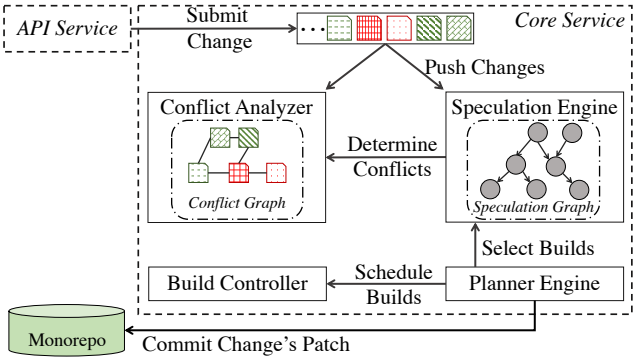


Figure 4. SubmitQueue Architecture

merged with the mainline branch if all build steps succeed. Figure 4 shows the high level architecture of SubmitQueue.

Upon submitting a change to SubmitQueue through the API service (i.e., step 5 in Figure 3), SubmitQueue enqueues the change in a distributed queue. The core service then needs to perform all necessary build steps for every enqueued (i.e., pending) change, and either lands the change, or aborts it along with a reason.

The core service uses a *planner engine* to orchestrate executions of pending changes. In order to scale to thousands of changes per day while ensuring serializability, the planner engine speculates on outcomes of pending changes using a *speculation engine*, and executes their corresponding builds in parallel by using a *build controller*.

The planner engine periodically contacts the speculation engine, and selects speculation builds that are most likely to succeed. In turn, the speculation engine uses a probabilistic model, powered by logistic regression, to compute likelihoods of speculations succeeding.

The core service also contains a scalable *conflict analyzer* that constructs a conflict graph among pending changes. The conflict graph helps the speculation engine to (1) trim the speculation space to further improve the likelihood of using remaining speculations, and (2) determine independent changes that can commit in parallel.

Based on the selected builds, the planner engine then performs the following actions: (1) schedules executions of selected builds, returned by the speculation engine, through the build controller, (2) aborts builds that do not exist in the latest set of selected builds, but were scheduled previously, and (3) commits a change's patch into the monorepo (i.e., step 6 in Figure 3) once it is safe.

4 Probabilistic Speculation

Every change submitted to SubmitQueue has two possible outcomes: (i) All build steps for the change succeed, and it gets committed (i.e., its patch gets merged into the mainline). (ii) Some build step fails, and the change is rejected. Therefore, if we can predict the outcome of a change, then we can

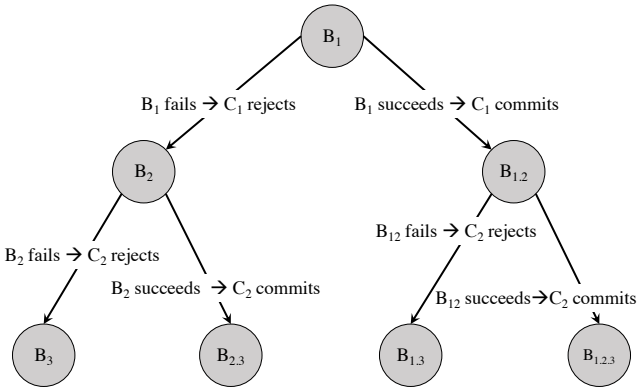


Figure 5. Speculation tree of builds and their outcomes for changes C_1, C_2, C_3

precompute the result of committing concurrent changes under this speculation.

Let H denote the current *HEAD*, and C_1, C_2 represent the changes that need to be committed into the mainline branch. In order to land C_1, we need to merge C_1 into H, denoted as $H \oplus C_1$, and make sure that all build steps succeed. Let B_1 denote the build steps for $H \oplus C_1$, and $B_{1.2}$ represents build steps for $H \oplus C_1 \oplus C_2$.

By assuming that B_1 has a high probability to succeed, we can proactively start executing $B_{1.2}$ in parallel with B_1. If B_1 succeeds, we can use the precomputed result of $B_{1.2}$ to determine if C_2 can commit or not. If the prediction does not hold (i.e., B_1 fails), then we need to build C_2 alone by running B_2. Therefore, the key challenge is to determine which set of builds we need to run in parallel, in order to improve turnaround time and throughput, while ensuring an always green mainline. To this end, the speculation engine builds a binary decision tree, called *speculation tree*, annotated with prediction probabilities for each edge. The speculation tree allows the planner engine to select the builds that are more likely to succeed, and speculatively execute them in parallel.

Figure 5 shows the speculation tree for changes C_1, C_2, and C_3 where C_1 is the first submitted change to SubmitQueue, and C_3 is the last submitted change. We note that in this section, and for simplicity, we assume that all pending changes conflict with each other. In the next section, we explain how we detect independent changes, and change the speculation tree to a speculation graph.

4.1 Speculate Them All

The fastest and most expensive approach is to speculate on all possible outcomes for every pending change by assuming that the probability of committing a change is equal to the probability of rejecting a change. For instance, we need to execute all the seven builds shown in Figure 5 in order to commit C_1, C_2, C_3. Therefore, we would need to perform $2^n - 1$ builds for n pending changes.

However, we do not need to speculate on all the possible builds. For example, consider the example in Figure 5. If B_1 succeeds, and C_1 can be committed, the result of B_2 (i.e., build steps for $H \oplus C_2$) becomes irrelevant since now we can either commit C_1 or $C_1 \oplus C_2$. Thus B_2 can be stopped. Similarly, if B_1 fails, and C_1 gets rejected, then $B_{1.2}$ containing both C_1 and C_2 can never be successful, and should be stopped. Thus, depending on the outcome of B_1, either B_2 or $B_{1.2}$ becomes irrelevant, and its computation is wasted. Likewise, for C_3, only one of $\{B_3, B_{1.3}, B_{2.3}, B_{1.2.3}\}$ is needed to commit C_3 when result of B_1, B_2 or $B_{1.2}$ becomes known.

To generalize, only n out of $2^n - 1$ builds will ever be needed to commit n pending changes. Considering that the system receives hundreds of changes an hour, this approach leads to substantial waste of resources since the results of most speculations will not be used.

4.2 Probabilistic Modeling

Instead of executing on all possible builds, SubmitQueue uses a probabilistic model, powered by logistic regression, to select and execute builds that are more likely to succeed.

4.2.1 Value of Build

Considering a set of changes C, the value of building these changes can be measured as

$$\mathcal{V}_{B_C} = \mathcal{B}_{B_C} \cdot \mathcal{P}_{B_C}^{needed}$$

where $\mathcal{P}_{B_C}^{needed}$ is the probability that the result of build B_C will be used to make decision to commit or reject a set of changes C. $\mathcal{B}_{B_C}$ is the benefit (e.g., monetary benefit) obtained by performing the build on the set of changes C. In this paper, and for the sake of simplicity, we assume the same benefit for all builds. In practice, we can assign different values to different builds. For instance, builds for certain projects or with certain priority (e.g., security patches) can have higher values, which in turn will be favored by SubmitQueue. Alternatively, we may assign different quotas to different teams, and let each team manages the benefits of its changes. In our current implementation, we consider the benefit of all builds to be one.

Consider the speculation tree shown in Figure 5. In this example, there are three pending changes, named C_1, C_2 and C_3. The root of the tree, B_1, is always needed as it is used to determine if C_1 can be committed safely. Thus, $\mathcal{P}_{B_1}^{needed} = 1$. $B_{1.2}$ is only needed when B_1 succeeds. If B_1 fails, $B_{1.2}$ is not needed, but B_2 is needed. Thus,

$$\begin{aligned} \mathcal{P}_{B_{1.2}}^{needed} &= \mathcal{P}_{B_1}^{succ} \\ \mathcal{P}_{B_2}^{needed} &= 1 - \mathcal{P}_{B_1}^{succ} \end{aligned} \tag{1}$$

The probability that an individual change, C_i, succeeds (denoted as $\mathcal{P}_{C_i}^{succ}$) is equal to the probability that its build independently succeeds: $\mathcal{P}_{B_1}^{succ} = \mathcal{P}_{C_1}^{succ}$. By substituting the

probability of successful builds in Equation 1, we get:

$$\mathcal{P}^{needed}_{B_{1.2}} = \mathcal{P}^{succ}_{C_1}$$
$$\mathcal{P}^{needed}_{B_2} = 1 - \mathcal{P}^{succ}_{C_1} \quad (2)$$

The above equations can be extended to builds in the next level: $B_{1.2.3}$ is only needed under the condition that both B_1 and $B_{1.2}$ succeed. This can be written as

$$\mathcal{P}^{needed}_{B_{1.2.3}} = \mathcal{P}^{succ}_{B_1} \cdot \mathcal{P}^{succ}_{B_{1.2}|B_1} \quad (3)$$

where $\mathcal{P}^{succ}_{B_{1.2}|B_1}$ denotes the probability that $B_{1.2}$ succeeds given the success of B_1. Let's consider the inverse. There are two conditions in which $B_{1.2}$ will fail given B_1 succeeds.

1. C_2 fails individually when tested against the current HEAD. For instance, it can fail because of a compilation error, a unit test failure, or a UI test failure.
2. C_2 *conflicts* with C_1, denoted as $\mathcal{P}^{conf}_{C_1,C_2}$. A conflict implies that C_2 succeeds individually against the current HEAD but fails when combined with C_1. Examples of conflicts are merge conflicts, or unit tests failures when both changes are applied.

Based on the above conditions, we can derive

$$\mathcal{P}^{succ}_{B_{1.2}|B_1} = \mathcal{P}^{succ}_{C_2} - \mathcal{P}^{conf}_{C_1,C_2} \quad (4)$$

By substituting $\mathcal{P}^{succ}_{B_{1.2}|B_1}$ in Equation 3 with Equation 4, we get:

$$\mathcal{P}^{needed}_{B_{1.2.3}} = \mathcal{P}^{succ}_{C_1} \cdot (\mathcal{P}^{succ}_{C_2} - \mathcal{P}^{conf}_{C_1,C_2}) \quad (5)$$

Consequently, by determining the probability that a change succeeds, and the probability that some changes conflict, we can compute the value of builds. This allows SubmitQueue to schedule builds with the highest value in order to commit pending changes.

We observed that statically assigning $\mathcal{P}^{succ}_{C_i}$ and $\mathcal{P}^{conf}_{C_1,C_2}$ based on heuristics results in sub-optimal selection of builds, and high turnaround time. This is because the penalty of mis-speculation, with hundreds of commits every hour and long build times, is huge. Additionally, by statically assigning probabilities, the system is not able to react to build successes or failures.

SubmitQueue uses the conventional regression model for predicting probabilities of a change success or a change failure. As we will show in Section 8, we observed that by correctly estimating $\mathcal{P}^{succ}_{C_i}$ and $\mathcal{P}^{conf}_{C_i,C_j}$, SubmitQueue's performance becomes close to the performance of a system with an oracle that can accurately foresee the success/failure of a build. Section 7 explains how the regression model is trained, and used by SubmitQueue.

4.2.2 Speculating Independent Changes

In the previous section, we considered cases where changes can potentially conflict with each other. However, in many cases, changes do not conflict with each other. For instance, let's consider C_1, C_2 where C_1 changes a *README* file in a service while C_2 introduces a feature in a mobile app. If we were to use the previous approach and speculate on C_1's success while testing C_2, then probability of success of $B_{1.2}$, given that $\mathcal{P}^{conf}_{C_1,C_2}$ =0 is:

$$\mathcal{P}^{succ}_{B_{1.2}} = \mathcal{P}^{succ}_{C_1} \cdot \mathcal{P}^{succ}_{C_2}$$

This implies that if two changes are completely independent, then building them individually and committing them in parallel has a higher probability to succeed than building them together. In the next section, we explain how the speculation engine uses the conflict analyzer to determine whether two changes conflict or not.

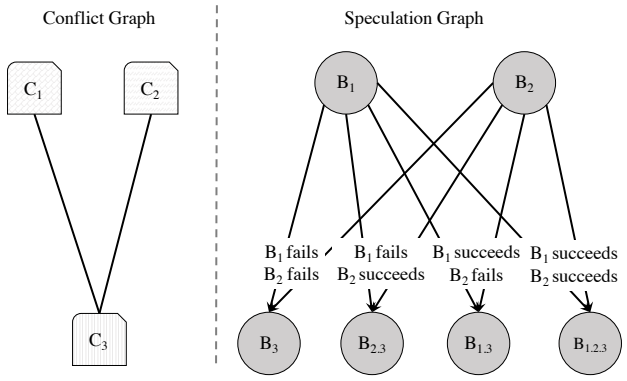


Figure 6. Conflict graph and speculation graph for independent C_1 and C_2 changes both conflicting with C_3

5 Conflict Analyzer

In the previous section, we assumed that all changes conflict with each other. Therefore, the result of our speculation model is always a tree (e.g., Figure 5) where the tree root is the first enqueued pending change.

By determining independent changes, the speculation engine can (1) change the speculation tree into a speculation graph where independent changes can get committed in parallel, and (2) trim unnecessary builds from the graph.

First consider the scenario depicted in Figure 6. In this example, C_1 and C_2 do not conflict with each other while both of them conflict with C_3. As explained in Section 4.2.2, since C_1 and C_2 are independent, B_1 and B_2 can execute in parallel to determine the outcome of the two changes. On the other hand, since C_3 potentially conflicts with both C_1 and C_2, C_3 needs to speculate on both C_1 and C_2 which results in four builds as shown in Figure 6. Observe that this is equal to the number of speculation builds for C_3 depicted in Figure 5. Therefore, taking the conflict graph into account in this example solely helps C_2 as it only requires one build, and can commit independently.

Figure 7 depicts a second example where C_1 conflicts with both C_2 and C_3 while C_2 and C_3 are independent. As a result, unlike the example shown in Figure 5, there are only two

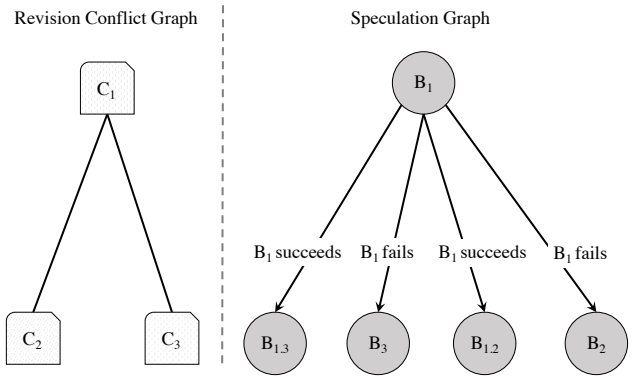


Figure 7. Conflict graph and speculation graph for C_1 conflicting with both C_2 and C_3

builds to consider for C_3 and the choice is solely based on C_1. Therefore, the total number of possible builds decreases from seven to five.

5.1 Build Systems & Targets

In order to build a conflict graph among pending changes, the conflict analyzer relies on on the build system. A build system (e.g., Bazel [1], Buck [3]) partitions the code into smaller entities called *targets*. Roughly speaking, a target is a list of source files, their dependencies along with a specification on how these entities can be translated into one or more output files. The directed acyclic graph of targets are then executed in a topological order such that the executed build actions are hermetic.

5.2 Conflict Detection

Every change modifies a set of build targets, which is a transitive closure of targets whose source files are modified by the change along with the set of targets that depend on them. Roughly speaking, two changes conflict if they both affect a common set of build targets.

To this end, we associate every build target with a unique target hash that represents its current state. We then use target hashes to detect which build targets are affected by a change. Algorithm 1 demonstrates how a target hash is computed for a given target. First, we find all the dependencies of a target, recursively compute their target hashes, and update a message digest with these hashes (lines 3 to 5). Second, we find all the source files that the target depends on, and update the message digest with their contents (lines 6 to 8). Third, we convert the message digest to a target hash - a fixed length hash value.

Formally, let $\delta_{H \oplus C_i}$ denote a set of targets affected by C_i against HEAD. An affected target is defined as a 2-tuple (*name*, *hash*) where *name* represents the target's name, and *hash* represents the target's hash value after applying C_i. Also, let $\cap_{name}$ denote intersection on target names. Thus,

Algorithm 1 Calculating Target Hash Algorithm

```
1:  function THASH(target)
2:      md ← ∅
3:      deps ← dependentTargetsOf(target)
4:      for dep ∈ deps do
5:          md ← md ⊕ THASH(dep)
6:      srcs ← sourceFilesOf(target)
7:      for src ∈ srcs do
8:          md = md ⊕ contentsOf(src))
9:      return hash(md)
10:
```

$\delta_{H \oplus C_i} \cap_{name} \delta_{H \oplus C_j} \neq \varnothing$ implies that C_i and C_j conflict since there are some common targets affected by both changes.

Two changes might still conflict even if they do not affect a common set of targets. For example, consider the case shown in Figure 8. The original build graph for the HEAD contains three targets where target Y depends on target X while target Z is independent. Numbers inside circles denote target hash values. Applying C_1 to the HEAD leads to targets X and Y being affected. Thus, they have different target hashes compared to target X and Y in the original build graph. More precisely, $\delta_{H \oplus C_1} = \{(X, 4), (Y, 5)\}$. After applying C_2 to the HEAD, target Z along with its dependencies change: $\delta_{H \oplus C_2} = \{(Z, 6)\}$. Observe that while these two changes conflict with each other, simply computing the intersection of affected targets does not yield any conflict.

Consequently, to accurately determine two independent changes, we also need to make sure that every hash of an affected target after applying both changes to the HEAD is either observed after applying the first change to the HEAD, or observed after applying the second change to the HEAD. If neither holds, it implies that the composition of both changes results in a new affected target, and two changes conflict. Formally, C_i conflicts with C_j if the following equation holds.

$$\delta_{H \oplus C_i} \cup \delta_{H \oplus C_j} \neq \delta_{H \oplus C_i \oplus C_j} \tag{6}$$

For instance, in Figure 8, $\delta_{H \oplus C_1} \cup \delta_{H \oplus C_2} = \{(X, 4), (Y, 5), (Z, 6)\}$ while $\delta_{H \oplus C_1 \oplus C_2} = \{(X, 4), (Y, 5), (Z, 7)\}$.

Determining whether two changes conflict based on the above approach requires building the target graphs four times using H, $H \oplus C_i$, $H \oplus C_j$, and $H \oplus C_i \oplus C_j$. Therefore, for committing n changes, we need to compute the build graphs approximately n^2 times. Since it may take several minutes to compute the build graph for a change on a big monorepo with millions of lines of code, the above solution cannot scale to thousands of commits per day.

Observe that Equation 6 is only needed if a build graph changes as a result of applying a change. We observed that only 7.9% (resp. 1.6%) of changes actually cause a change to the build graph for iOS (resp. Backend) monorepos over a span of five months. Therefore, if the build graph remains unchanged, we just need to use the intersection of changed

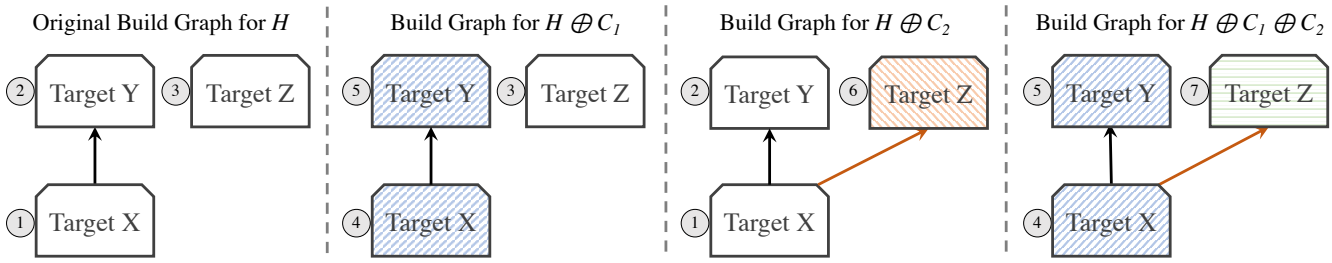


Figure 8. Changes in target graph with different changes. Circle denotes a target hash value. Arrow from target X to target Y implies output of target X is required for building target Y. Shaded target node means the target is affected by the corresponding change.

targets to decide whether the two changes conflict. This leads to substantial elimination of needed build graphs.

Alternatively, we can also determine if C_i and C_j conflict by solely computing a union of build graphs for H, $H \oplus C_i$, and $H \oplus C_j$ instead of also computing a build graph for $H \oplus C_i \oplus C_j$. This approach requires n build graphs for detecting conflicts among n changes.

Let $\mathcal{G}_H$ denote a build graph at HEAD where a node is a target, and an edge is a dependency between two nodes. The following algorithm determines if two changes conflict:

- **Step 1:** we first build a *union graph* among $\mathcal{G}_H$, $\mathcal{G}_{H \oplus C_i}$, and $\mathcal{G}_{H \oplus C_j}$ as follows: for every node in any of these build graphs, we create a corresponding node in the union graph. Every node in the union graph contains all the target hashes of its corresponding nodes in the individual build graphs. Precisely, a node in the union graph is a 4-tuple of target name and all of its corresponding hashes in $\mathcal{G}_H$, $\mathcal{G}_{H \oplus C_i}$, and $\mathcal{G}_{H \oplus C_j}$. For example, considering Figure 8, we will have the following nodes: $(A, 2, 5, 2)$, $(B, 1, 4, 1)$, and $(C, 3, 3, 6)$. Lastly, we will add an edge between two nodes in the union graph if there exists an edge between its corresponding two nodes in any of the build graphs.
- **Step 2:** a node is tagged as *affected* by C_i (resp. C_j) if its target hash in $\mathcal{G}_H$ is different from its target hash in $\mathcal{G}_{H \oplus C_i}$ (resp. $\mathcal{G}_{H \oplus C_j}$). Considering the example shown in Figure 8, targets X and Y are tagged affected by C_1, and target Z is tagged affected by C_2 in the union graph.
- **Step 3:** we then traverse the union graph in a topological order, and mark a visiting node as affected by C_i (resp. C_j) if any of its dependent nodes is affected by C_i (resp. C_j). Therefore, in our example, target Z is tagged affected by both C_1 and C_2.
- **Step 4:** C_i and C_j conflict if there exists a node that is affected by both C_i and C_j.

6 Planner Engine & Build Controller

Based on the number of available resources, the planner engine contacts the speculation engine on every epoch, and receives a prioritized list of builds needed. The planner engine then terminates builds that are currently running, but are not part of the received list. It then schedules new builds according to their priorities in the list using the build controller. In turn, the build controller employs the following techniques to efficiently schedule builds.

Minimal Set of Build Steps. Instead of performing all build steps, the build controller eliminates build steps that are being executed by prior builds. For instance, when scheduling $B_{1.2.3}$ (build steps for $H \oplus C_1 \oplus C_2 \oplus C_3$), the build controller can eliminate the build steps that are already performed by prior builds (i.e., B_1 or $B_{1.2}$). Therefore, the build controller only needs to perform build steps for targets affected by C_3 (i.e., $\delta_{H \oplus C_1 \oplus C_2 \oplus C_3} - \delta_{H \oplus C_1 \oplus C_2}$).

Load Balancing. Once the list of affected targets is determined, the build controller uniformly distributes the work among the available worker nodes. Accordingly, it maintains the history of build steps that were performed, along with their average build durations. Based on this data, the build controller assigns build steps to workers such that every worker has an even amount of work.

Caching Artifacts. In order to further improve the turnaround times, the build controller also leverages caching mechanisms that exist in build systems to reuse generated artifacts, instead of building them from scratch.

7 Implementation

7.1 API & Core Services

SubmitQueue's API Service is a stateless backend service that provides the following functions: landing a change, and getting the state of a change. The service is implemented in 5k lines of Java using Dropwizard [6] - a popular Java framework for developing RESTful web services. It also features a web UI using cycle.js [5] to help developers track the state of their changes.

The core service is implemented with 40k lines of Java code. It uses Buck [3] as the underlying build system for constructing conflict graphs, MySQL as the backend storage system, Apache Helix for sharding queues across machines, and RxJava for communicating events within a process.

One challenge in dealing with a speculation graph is the number of builds that need to be considered to find the most valuable builds. More precisely, for n pending changes in the system, the speculation engine needs to consider and sort 2^n builds in the worst case. In order to scale to hundreds of concurrent pending changes, SubmitQueue uses a greedy best-first algorithm that visits nodes having the highest probability first. Because edges in the speculation graph represent the probability a path is needed, the value of a node always goes down as the path length increases. Hence, greedy best first helps SubmitQueue to navigate the speculation graph optimally in memory while scaling to hundreds of pending changes per hour. Observe that the required space complexity of this approach is $O(n)$.

7.2 Model Training

We trained our success prediction models *predictSuccess*(C_i) and *predictConflict*(C_i,C_j) that estimate $\mathcal{P}_{C_i}^{succ}$ and $\mathcal{P}_{C_i,C_j}^{conf}$ in a supervised manner using logistic regression. We selected historical changes that went through SubmitQueue along with their final results for this purpose. We then extracted around 100 handpicked features. Our feature set is almost identical for training both of the above models, and can be categorized as follows:

Change. Clearly changes play the most important role in predicting the likelihoods of a change succeeding, or conflicting with another change. Thus, we considered several features of a change, including (i) number of affected targets, (ii) number of git commits inside a change, (iii) number of binaries added/removed, (iv) number of files changed along with number of lines/hunks added or removed, and (v) status of initial tests/checks run before submitting a change.

Revision. As explained in Section 3, a revision is a container for storing changes. We observed that revisions with multiple submitted changes tend to succeed more often than newer revisions. Additionally, we noticed that the nature of a revision has an impact on its success. Therefore, we included features such as (i) number of times changes are submitted to a revision, and (ii) revert and test plan suggested as part of the revision.

Developer. Developers also play a major role in determining whether a change may succeed or not. For instance, experienced developers do due diligence before landing their changes while inexperienced developers tend to land buggy changes more often. Some developers may also work on more fragile code-paths (e.g., core libraries). Thus their initial land attempts fail more often. Additionally, we observed that developer data helped us tremendously in detecting chances of conflicts among changes. This is due to the fact that developers working on the same set of features (or code path) conflict with each other more often. Consequently, we selected several developer features such as their name, employment length, and levels.

Speculation. The above identified features, associated with revisions/changes, are static: they do not change over time for a given land request. Therefore, while they can initially help to determine likelihoods of a change succeeding or conflicting with another change, they cannot help as SubmitQueue performs different speculations. This leads to high mis-speculation penalties when initial speculations are not accurate. To side step this issue, the number of speculations that succeeded or failed were also included for training.

The dataset was then divided into two sets: 70% for training the model and 30% for validating the model. The model was trained using scikit [10] in Python, and has an accuracy of 97%. To avoid overfitting the models and keeping the computation during actual prediction fast, we also ran our model against recursive feature elimination (RFE) [25]. This helped us reduce the set of features to just the bare minimum.

In our best performing model, the following features had the highest positive correlation scores: (1) number of succeeded speculations, (2) revision revert and test plans, and (3) number of initial tests that succeeded before submitting a change. In contrast, number of failed speculations, and number of times that changes are submitted to a revision had the most negative weights. We also note that while developer features such as the developer name had high predictive power, the correlation varied based on different developers.

8 Evaluation

We evaluated SubmitQueue by comparing it against the following approaches:

- *Speculate-all:* as discussed in Section 4.1, this approach tries all possible combinations that exist in the speculation graph. Therefore, it assumes that the probability of a build to succeed is 50%.
- *Single-Queue:* where all non-independent changes are enqueued, and processed one by one, à la Bors [2]. Independent changes, on the other hand, are processed in parallel.
- *Optimistic speculation:* similar to Zuul [12], this approach selects a path in the speculation graph by assuming that all concurrent changes will succeed. Therefore, a pending change starts performing its build steps assuming that all the pending changes that were submitted before it will succeed. If a change fails, all the builds that speculated on the success of the failed change need to get aborted, and a new path in the speculation graph is selected.

To compare these approaches in a more meaningful way, we also implemented an Oracle, and normalized turnaround time and throughput of the above approaches against it. Our Oracle implementation can perfectly predict the outcome of a change. Since Oracle always makes the right decision in

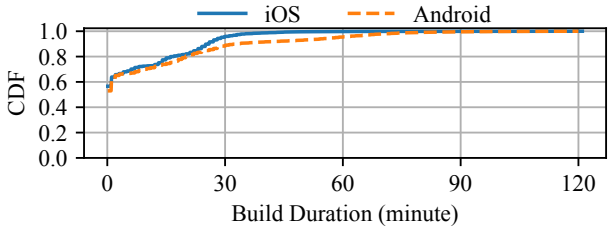


Figure 9. CDF of build duration for iOS/Android monorepos

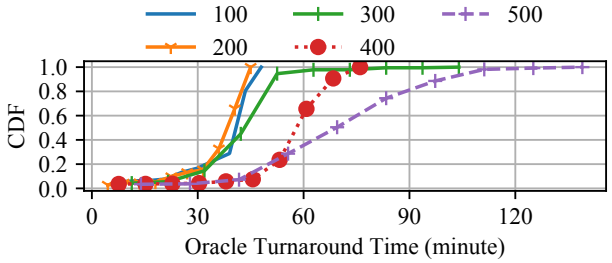


Figure 10. CDF of Oracle turnaround time for 100, 200, 300, 400, and 500 changes per hour

the speculation graph, it has the shortest turnaround time, lowest overhead, and the best throughput.

In this section, we first study how the above approaches impact turnaround time (Section 8.2) and throughput (Section 8.3) as number of changes per hour and number of workers increase. We then take a more microscopic look at SubmitQueue, and investigate benefits of conflict analyzer, and how it helps different approaches (Section 8.4). We present the state of our monorepo and its breakages before SubmitQueue was launched in Section 8.5. We conclude our evaluation section by reporting the results of our survey on SubmitQueue in Section 8.6.

8.1 Setup

We ran all the iOS build steps in our datacenter with Mac Minis, each equipped with 2 cores (Intel Core i7 processors, 3.3 GHz), 16 GB memory and 512 GB of SSD storage. Our API and core servies run on Linux machines equipped with 32 cores (Intel Core i7 processors, 3 Ghz), with 128 GB memory and 2 TB of SSD storage.

Figure 9 plots the cumulative distribution function (CDF) of build durations for changes submitted to our iOS and Android monorepos in the first 9 months of 2018. Since they have similar CDFs, we only report our evaluation results for the iOS monorepo.

To evaluate the performance of SubmitQueue in a controlled way, we selected the above changes, and ingested them into our system at different rates (i.e., 100, 200, 300, 400 and 500 changes per hour). Thus, the only difference with the real data is the inter-arrival time between two changes in order to maintain a fixed incoming rate. Figure 10 shows the CDF of turnaround time for different rates with Oracle running under no contention (with 2000 workers). Observe that the difference between Figure 9 and Figure 10 is in fact the cost of serializing conflicting changes.

8.2 Turnaround Time

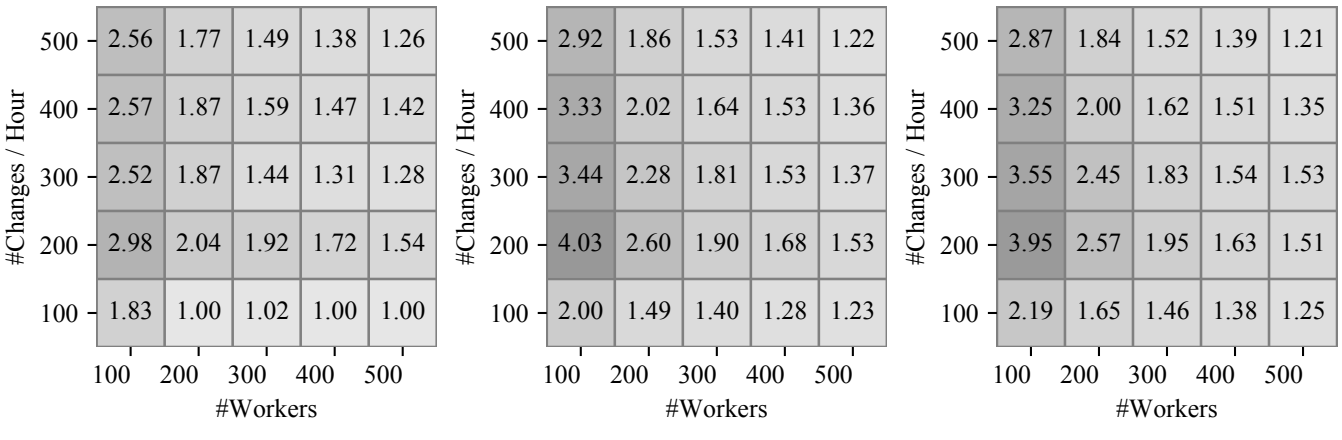
Figure 11 plots P50, P95 and P99 turnaround times normalized against Oracle. As illustrated by Figure 11a, SubmitQueue's logistic regression model for probabilistic speculation along with conflict analyzer leads to maximum of 2.98x slower 50-percentile turnaround time, compared to the optimal Oracle solution. Similarly, SubmitQueue's 95-percentile turnaround latency increases by 4x under high contention in which not enough resources are available for handling incoming changes (see Figure 11b). Yet, by properly provisioning the number of worker nodes, the turnaround time can substantially be reduced. For instance, if the maximum number of incoming changes per hour is 500, and with 500 workers, 50-percentile, 95-percentile and 99-percentile turnaround time gets reduced to 1.2x of the Oracle's.

On the other hand, as shown in Figure 11, the P50 turnaround time of the Speculate-all approach suffers up to 15x, while its P99 turnaround time increases up to 24x. Interestingly though, the Optimistic approach results in better turnaround times compared to Speculate-all. This is due to the fact that a significant proportion of changes are expected to succeed even under high load. Thus, speculating deep instead of going wide helps. We observed that Single-Queue yields the worst turnaround times. With 500 changes per hour, P50, P95, P99 turnaround times grow 80x, 129x and 132x respectively compared to the turnaround times of the Oracle. Due to its poor performance, we omitted the results of Single-Queue.

Observe that as expected, turnaround time decreases as we increase the number of workers in all the cases. However, normalized (P50, P95 and P99) turnaround times for 200 changes per hour is higher than normalized turnaround times for 300/400/500 changes per hour with the same number of machines. Unlike normalized turnaround time, we observed that actual turnaround time consistently increases as the number of changes per hour increases. Yet, depending on production data, in certain hours, Oracle can perform substantially better than other approaches which results in larger gaps.

8.3 Throughput

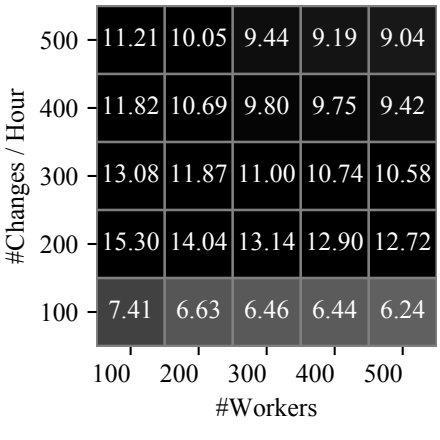
Figure 12 illustrates average throughput of different approaches normalized against Oracle. SubmitQueue's approach has the least throughput slowdown compared with other approaches. While under high contention, SubmitQueue experiences up to 60% slowdown compared to Oracle, the slowdown reduces to around 20% with 500 workers. We observed similar



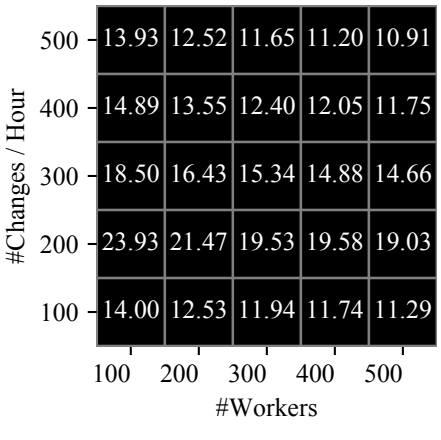
(a) SubmitQueue P50 Turnaround Time

(b) SubmitQueue P95 Turnaround Time

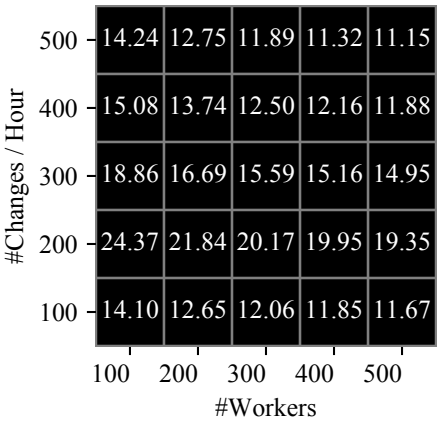
(c) SubmitQueue P99 Turnaround Time



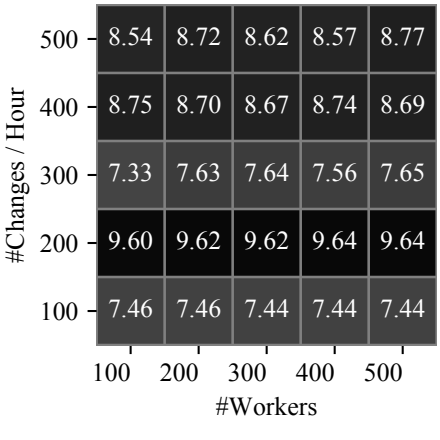
(d) Speculate-all P50 Turnaround Time



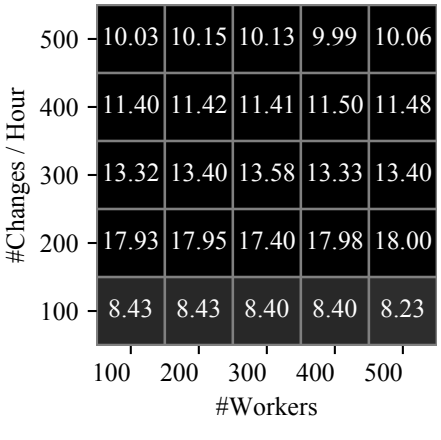
(e) Speculate-all P95 Turnaround Time



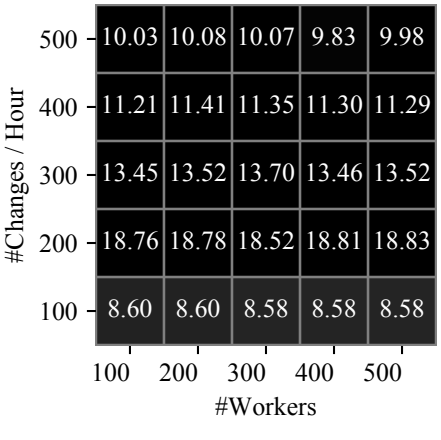
(f) Speculate-all P99 Turnaround Time



(g) Optimistic P50 Turnaround Time



(h) Optimistic P95 Turnaround Time

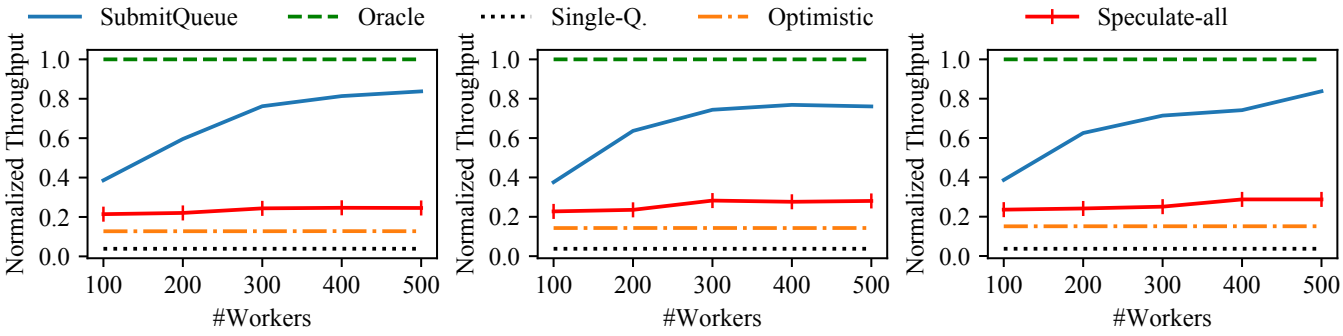


(i) Optimistic P99 Turnaround Time

Figure 11. Turnaround Time Normalized Against Oracle

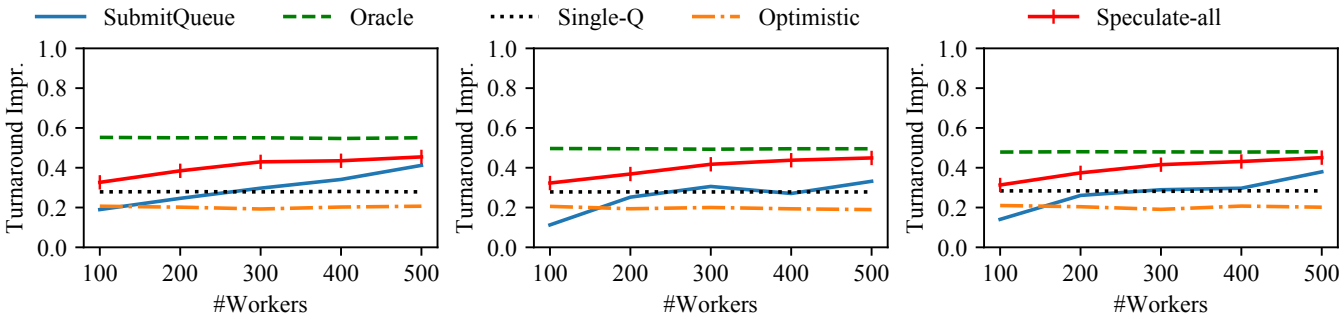
behavior with 200 changes per hour. Additionally, for 100 changes per hour, we noticed that SubmitQueue's throughput matches Oracle's. Throughput results for 100 and 200 changes per hour are not included.

Among the studied solutions, Single-Queue has the worst throughput (i.e., 95% slowdown). Surprisingly, we also observed that not only does the Optimistic approach do worse than Speculate-all, it remains unchanged as we increase the number of workers. This is due to the fact that with the Optimistic approach, the system assumes that all changes will



(a) 300 Changes / Hour **(b)** 400 Changes / Hour **(c)** 500 Changes / Hour

Figure 12. Average throughput normalized against Oracle



(a) 300 Changes / Hour **(b)** 400 Changes / Hour **(c)** 500 Changes / Hour

Figure 13. P95 turnaround time improvement when using conflict analyzer

succeed. Therefore, its throughput is limited by the number of contiguous changes that succeed. However, since this is always less than 100 in our workload, the throughput remains the same as we increase the number of machines.

Finally, observe that adding more workers does not have any effect on the Speculate-all approach. Since our speculation graph is deep, adding a few hundred workers does not have any effect on throughput. However, for wide speculation graphs (i.e., more independent changes), we expect much better performance.

8.4 Benefits of Conflict Analyzer

Figure 13 shows how the conflict analyzer improves 95-percentile turnaround time of different approaches. The turnaround time of Oracle improves by up to 60% with the conflict analyzer. Likewise, both SubmitQueue and Speculate-all also substantially benefit from the conflict analyzer as it helps them to perform more parallel builds.

Surprisingly though, the effect of the conflict analyzer on the Optimistic approach is only 20%. Moreover, observe that for both the Optimistic and Single-Queue approaches, the turnaround time improvement remains constant as we increase the number of workers. This is due to the fact that the build graph on the iOS monorepo is very deep (i.e., only a handful of leaf-level nodes) resulting in a large number of conflicts among changes. Consequently, the speculation graph has few independent changes that can execute and commit in parallel. Therefore, we expect substantially better improvements when using the conflict analyzer for repositories that have a wider build graph.

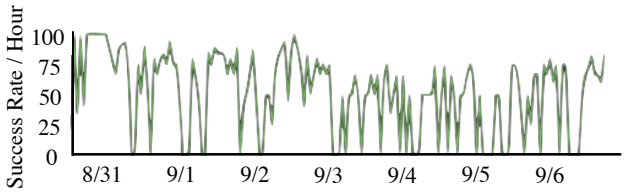


Figure 14. State of the mainline for iOS monorepo prior to SubmitQueue launch

8.5 Mainline State Prior to SubmitQueue

Figure 14 shows the state of iOS mainline prior to SubmitQueue. Over a span of one week, the mainline was green

only 52% of the time affecting both development and rollouts. This clearly exhibits the need for a change management system that guarantees an always green mainline on fast-moving projects involving thousands of developers. Since its launch, our mainlines have remained green at all times.

8.6 Survey on Benefits of SubmitQueue

We conducted an internal survey, and asked for feedback from developers, release engineers, and release managers. Specifically, we sought answers to the following two questions: (1) what is the perceived impact of an always green master on personal productivity and getting code to production, and (2) how good is the performance of SubmitQueue in production. Out of 40 people who answered the survey, close to 70% had some previous experience committing to monorepos, and 60% of responders had experience with an always green mainline.

92% of survey takers believed that enforcing an always green mainline positively impacts their personal productivity while 8% saw no impact. Interestingly though, none of the responders claimed that an always green master negatively impacts their productivity. Similarly, 88% of responders believed that an always green master positively impacts deploying code to production, while only one responder thought it imposes a negative impact.

As for the performance of SubmitQueue on the scale of 1 to 5 (where 5 is the best performance), 94% of responders felt that it was either on par or significantly better than prior experiences.

9 Related Work

In Section 2, we reviewed various solutions for preventing master breakages, and different techniques for recovering from master breakages. In this section, we focus on other research works that are relevant to SubmitQueue, but are not directly comparable to it.

Proactive Conflict Detection. An alternative approach to reduce the number of regressions in a monorepo is a proactive detection of potential conflicts among developers working on the same project. These solutions provide awareness to developers, and notify them that parallel changes are taking place. The granularity of these solutions can be file-level [15], method-level using static-analysis [21, 24, 41], and repository-level [16]. Yet, these solutions do not prevent mainline breakages, and solely notify developers of potential breakages. It is up to developers to further investigate, and prevent a breakage. Unlike SubmitQueue, these solutions do not scale because the number of commits that need to be considered is extremely high in large monorepos. Additionally, these approaches encourage developers to rush committing their changes in order to avoid merging with conflicting changes [20].

Transactional Systems. Transactional systems, including distributed data stores, software transactional memories and databases, use concurrency control schemes to ensure that committing a transaction does not break their consistency model (e.g., Serializability). To guarantee consistency, they employ different concurrency control mechanisms such as pessimistic 2PL mechanism (e.g., Spanner [19] and Calvin [43]), an optimistic concurrency control mechanism (e.g., Percolator [36] and CLOCC [13]), or conflict reordering mechanism (e.g., Rococo [34]). These schemes are tailored for relatively short running transactions. SubmitQueue on the other hand aims at ensuring serializability for transactions (i.e., changes) that may need hours to finish. Therefore, the penalty of both optimistic and pessimistic execution is extremely high as shown in Section 8. Additionally, conventional transactional systems can easily validate a transaction to make sure that it does not conflict with another transaction before committing it. SubmitQueue, on the other hand, needs to ensure the integrity of the mainline by performing several build steps before being able to commit a change.

Bug & Regression Detections. Bug & regression detections have been studied extensively by PL, software engineering, and systems communities. These works can be classified in the following categories: (i) techniques [28, 45] to identify bugs prior to submitting changes to repositories, (ii) tools [7, 29, 30] to find bugs after a change is submitted, and (iii) solutions [14, 18, 31, 44] that help online engineers and developers to detect and localize bugs post deployment.

Kamei et al. [28] and Yang et al. [45] introduce different ML techniques to detect if a change is risky and might cause regressions before it is submitted to a repository. To this end, they use information such as reviews, authors, and modified files. Similarly, SubmitQueue uses a ML model to determine the likelihood of a change's build succeeding or not. However, unlike these techniques, SubmitQueue also considers concurrent pending changes. Additionally, instead of rejecting a change, SubmitQueue uses the outcome of its model to determine builds that are more likely to succeed.

Developers can easily leverage many of the above tools to detect bugs prior to deploying their code by introducing new build steps, and let SubmitQueue runs those build steps before committing the code into a repository. For instance, one can use different tools [8, 11, 22, 26, 35] to eliminate `NullPointerException`, or detect data races by simply adding extra build steps that need to run upon submitting new changes.

Test Selection. As a number of changes along with a number of tests grow in many giant repositories, it may not be feasible to perform all tests during builds. To address this issue, several solutions have been proposed based on static analysis [17, 33, 40, 46], dynamic analysis [23, 39], and data-driven [32] using a ML model. Therefore, upon committing

a change, and instead of conducting all test cases, only a subset of test cases are required to run.

Our build controller currently does not leverage any of these techniques, and performs all required build steps (including test cases) for every build. Applying the above techniques is an interesting future work that can lead to substantial performance improvements.

Speculative Execution. Speculative execution has been used in almost every field in computer science, including processors, operating systems, programming languages, and distributed systems to name a few. SubmitQueue applies a similar technique in order to keep the mainline green at scale when thousands of developers commit their changes to a mono repository every day.

10 Discussion

While SubmitQueue addresses many challenges of a change management system, it still has some limitations. In this section, we discuss its limitations, and future works.

Change Reordering. The current version of SubmitQueue respects the order in which changes are submitted to the system. Therefore, small changes that are submitted to the system after a large change with long turnaround time (e.g., refactoring of a library API) need to wait for the large change to commit/abort before committing or aborting. As future work, we plan to reorder non-independent changes in order to improve throughput, and provide a better balance between starvation and fairness.

Other ML Techniques. Our current logistic regression model has an accuracy of 97%, and performs well in production as we showed in Section 8. Yet, exploring other ML techniques such as Gradient Boosting for our prediction model remains an interesting future work.

Build Preemption. As explained in Section 6, the planner engine aborts ongoing builds if the likelihood of their success drops, and the speculation engine returns a new set of builds that are more likely to succeed. However, if a build is near its completion, it might be beneficial to continue running its build steps, instead of preemptively aborting the build. As future work, we plan to further investigate this issue, and only abort builds that are very unlikely to be needed.

Batching Independent Changes. SubmitQueue performs all build steps of independent changes separately. A better approach is to batch independent changes expected to succeed together before running their build steps. While this approach can lead to better hardware utilization and lower cost, false prediction can result in higher turnaround time. Finding a balance between cost and turnaround time still needs to be explored.

11 Conclusion

This paper introduces a change management system called SubmitQueue that is responsible for continuous integration of changes into the mainline at scale while keeping the mainline green by ensuring serializability among changes. Based on possible outcomes of pending changes, SubmitQueue constructs, and continuously updates a speculation graph that uses a probabilistic model, powered by logistic regression. The speculation graph allows SubmitQueue to select builds that are most likely to succeed, and speculatively execute them in parallel. It also uses a conflict analyzer that constructs a conflict graph among pending changes in order to (1) trim the speculation space to further improve the likelihood of using remaining speculations, and (2) determine independent changes that can commit in parallel. SubmitQueue has been in production for over a year, and can scale to thousands of daily commits to giant monolithic repositories.

Acknowledgments

We thank Raj Barik, Milind Chabbi, Kamal Chaya, Prateek Gupta, Manu Sridharan, and Matthew Williams for their valuable suggestions on early versions of this paper. We also would like to thank our Shepherd, Steven Hand, and our anonymous reviewers for their insightful feedback and suggestions.

References

[1] 2018. Bazel. https://bazel.build/.
[2] 2018. Bors. https://github.com/graydon/bors.
[3] 2018. Buck. https://buckbuild.com/.
[4] 2018. Commit Queue. https://dev.chromium.org/developers/tree-sheriffs/sheriff-details-chromium-os/commit-queue-overview.
[5] 2018. Cycle.js. https://cycle.js.org/.
[6] 2018. Dropwizard. https://www.dropwizard.io.
[7] 2018. Git-bisect. https://git-scm.com/docs/git-bisect.
[8] 2018. NullAway. https://github.com/uber/NullAway.
[9] 2018. Rust-lang. https://www.rust-lang.org.
[10] 2018. Scikit. http://scikit-learn.org/stable/.
[11] 2018. ThreadSanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html.
[12] 2018. Zuul. https://zuul-ci.org/.
[13] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *International Conference on the Management of Data (SIGMOD)*. 23–34.
[14] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-Scale Services. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 493–509.
[15] Jacob T. Biehl, Mary Czerwinski, Mary Czerwinski, Greg Smith, and George G. Robertson. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In *Conference on Human Factors in Computing Systems (CHI)*. 1313–1322.
[16] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Symposium on the Foundations of Software Engineering (FSE) and European Software Engineering Conference (ESEC)*. 168–178.

[17] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 809–820.

[18] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *International Conference on Software Engineering (ICSE)*. 34–44.

[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 251–264.

[20] Cleidson R. B. de Souza, David F. Redmiles, and Paul Dourish. 2003. "Breaking the code", moving between private and public work in collaborative software development. In *International Conference on Supporting Group Work (GROUP)*. 105–114.

[21] Prasun Dewan and Rajesh Hegde. 2007. European Conference on Computer Supported Cooperative Work (ECSCW). 159–178.

[22] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Symposium on Operating Systems Principles (SOSP)*. 237–252.

[23] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis (ISSTA)*. 211–222.

[24] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *International Conference on Software Engineering (ICSE)*. 342–352.

[25] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. 2002. Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning* 46, 1 (01 Jan 2002), 389–422.

[26] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Conference on Programming Languages Design and Implementation (PLDI)*. 337–348.

[27] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K. Smith, Collin Winter, and Emerson Murphy-Hill. 2018. Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google. In *International Conference on Software Engineering (ICSE)*. 225–234.

[28] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773.

[29] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[30] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *International Conference on Automated Software Engineering (ASE)*. 81–90.

[31] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Conference on Programming Languages Design and Implementation (PLDI)*. 15–26.

[32] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2018. Predictive Test Selection. *Computing Research Repository (CoRR)* abs/1810.05286 (2018). arXiv:1810.05286 http://arxiv.org/abs/1810.05286

[33] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242.

[34] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 479–494.

[35] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Conference on Programming Languages Design and Implementation (PLDI)*. 308–319.

[36] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 251–264.

[37] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. 2001. Parallel Changes in Large-scale Software Development: An Observational Case Study. *ACM Transactions on Software Engineering and Methodology* 10, 3 (July 2001), 308–337.

[38] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59 (2016), 78–87.

[39] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (April 1997), 173–210.

[40] Barbara G. Ryder and Frank Tip. 2001. Change Impact Analysis for Object-oriented Programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 46–53.

[41] Anita Sarma, Gerald Bortis, and Andre van der Hoek. 2007. Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces. In *International Conference on Automated Software Engineering (ASE)*. 94–103.

[42] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Symposium on Operating Systems Principles (SOSP)*. 328–343.

[43] Alexander Thomson, Thaddeus Diamond, Philip Shao, and Daniel J. Abadi. 2012. Calvin : Fast Distributed Transactions for Partitioned Database Systems. In *International Conference on the Management of Data (SIGMOD)*. 1–12.

[44] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. In *Symposium on Operating Systems Principles (SOSP)*. 131–144.

[45] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *International Conference on Software Quality, Reliability and Security (QRS)*. 17–26.

[46] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *International Conference on Software Engineering (ICSE)*. 199–209.

[47] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 113–122.