

# Scaling Collaboration at Cornell DTI

Sam Zhou (tz66)

December 10, 2019

## 1 Introduction

Cornell DTI is a student-run engineering project team that has 7 active projects and over 60 public GitHub repositories. We have weekly work sessions where we decide what to do in the following week, and developers usually complete the majority of their work individually. To manage this distributed work, we use Git as our version control system. Every day, our developer pushes code changes to our repositories to make our products better. Coordinating such frequent changes is not an easy task and poor management can easily lead to a lack of progress or even an engineering disaster.

We will first analyze some scenarios where careless management can significantly slow down development and deployment, and then introduce our solution to these problems based on automation.

## 2 Unfortunate Scenarios

### 2.1 Collaborating on an Unprotected Master Branch

Imagine we have two developers Alice and Bob who are working on the same codebase. Bob created a broken commit 6 (shown in red in Figure 1) and pushed it to master without knowing it. When Alice made a series of good commit and pushed to master. Now both Alice and Bob had a broken codebase, and they both need to go through all the commits to figure out why it is broken.

On an unprotected master branch, a broken commit can affect everyone in the team. At Cornell DTI, there are more than 4 developers in each subteam and a broken commit on the master branch can have a much greater negative impact.

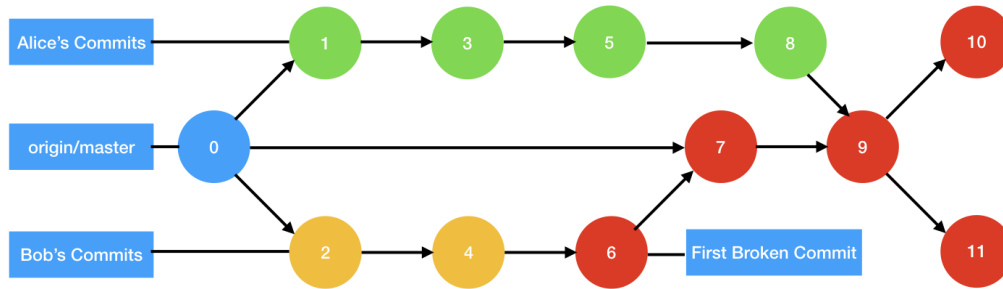


Figure 1: Both Alice and Bob can be affected by a broken commit. Commit 6, 7, 9, 10, 11 are all broken.

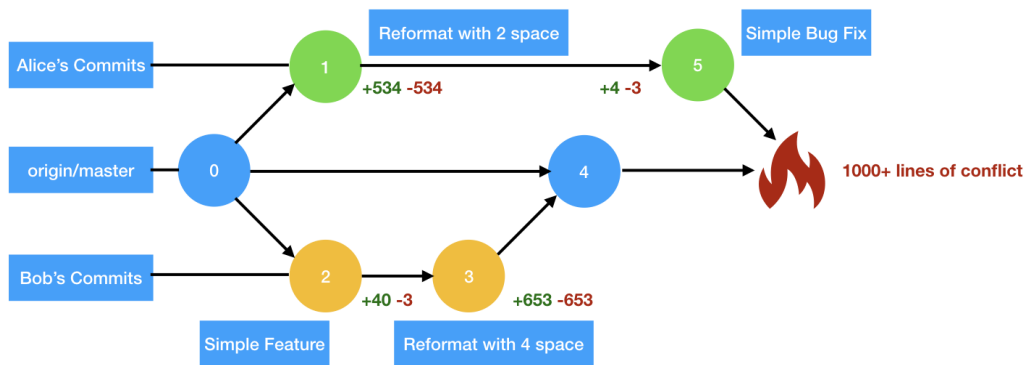


Figure 2: Different code style can result in unmergeable branches.

## 2.2 Collaborating Without a Consistent Code Style

Imagine Alice and Bob are working on the same file but on separate branches. Alice reformatted all the code with 2-space in commit 1 and made a small bug fix in commit 5. Bob implemented a simple feature in commit 2 and reformatted the same file with 4-space. Bob managed to merge his branch into master first. When Alice tried to merge her branch into master, she could be hit with over 1000 conflicting lines. The situation is visualized in Figure 2.

This is known as *merge conflict*. Git cannot distinguish between these whitespace-only changes and meaningful changes. Therefore, when the different edits are in the same place, it will force humans to resolve these differences. Resolving over 1000 lines of merge conflict is error-prone. Meaningful changes can easily get lost and we might break more functionality while merging branches.

Similar scenarios did occur in DTI's codebase. Once we had a commit that changed



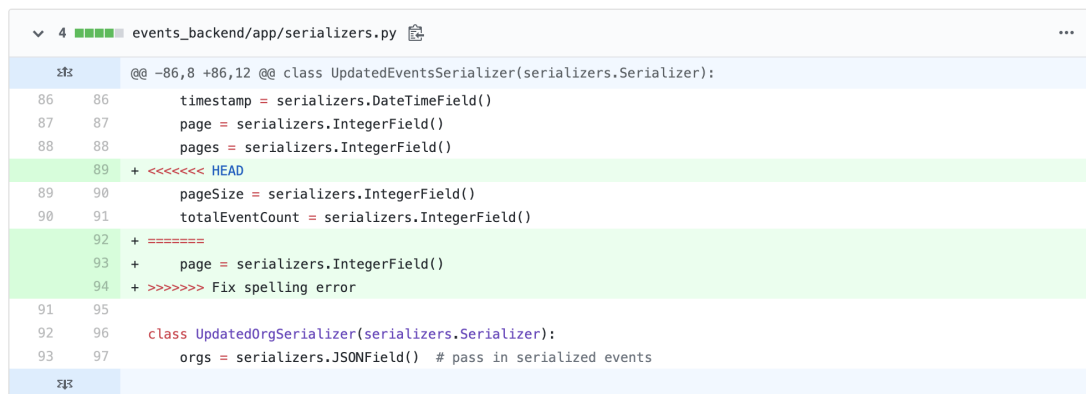
Showing 65 changed files with 17,706 additions and 17,696 deletions. Unified Split

```

  6 .babelrc
  @@ -1,4 +1,4 @@
  1 - {
  2 -   "presets": ["es2015", "react", "stage-2"],
  3 -   "plugins": ["transform-es2015-destructuring", "transform-object-rest-spread"]
  4 - }
  1 + {
  2 +   "presets": ["es2015", "react", "stage-2"],
  3 +   "plugins": ["transform-es2015-destructuring", "transform-object-rest-spread"]
  4 + }

```

Figure 3: A commit that changes all line-endings.



```

  4 events_backend/app/serializers.py
  @@ -86,8 +86,12 @@ class UpdatedEventsSerializer(serializers.Serializer):
  86 timestamp = serializers.DateTimeField()
  87 page = serializers.IntegerField()
  88 pages = serializers.IntegerField()
  89 + <<<<<<< HEAD
  89 pageSize = serializers.IntegerField()
  90 totalEventCount = serializers.IntegerField()
  92 + =====
  93 + page = serializers.IntegerField()
  94 + >>>>>> Fix spelling error
  91
  92 class UpdatedOrgSerializer(serializers.Serializer):
  93 orgs = serializers.JSONField() # pass in serialized events

```

Figure 4: A leftover merge conflict mark that is accidentally pushed.

all line-endings from `\n` to `\r\n` and created a huge diff, as shown in Figure 3. The second author tried very hard to find what is really changed, but in the end, we still had a leftover merge conflict mark that has been accidentally pushed, as shown in Figure 4.

### 2.3 Code Review by Manual Inspection

After we finished dealing with merge conflicts caused by bad collaboration, we will enter the code review stage. Imagine a code review workflow where the author of the code has to self-report whether his or her code follows code style of the project and is thoroughly tested. The reviewer either needs to risk pushing broken code to master to believe that the author is honest about the self-report or tries to validate the claim by himself or herself again.

The validation process can be painful. The reviewer needs to first stop his or her own work, switch to a different branch, and clear all caches to have a clean install. A clean install can take up to 2 minutes. The overhead of rigid code review is so big that

developers only do it when they see the live demo of others' work during work session. As a result, reviewers usually only care about whether things work rather than the code quality.

## 2.4 Deployment with a Codebase with Uncertain Quality

Imagine product manager Eve decided that it is time to deploy a new version. Eve knows that the code is not carefully reviewed and never fully tested. The entire team had to playtest the application for an entire week to ensure nothing goes wrong.

Suppose someone found and reported a bug in the middle of the week. The bug might only get fixed by the end of the week since our developers are also students and they are not always available. Feeling that the quality might be worrying, Eve decided to start another week of internal testing. The same story could happen again to further delay the launch for another week.

## 3 Avoiding Unfortunate Scenarios by Automation

Problems listed in the above sections are a result of non-strict or even non-existence of code review process. A stricter code review process is necessary to slightly slow everyone down to enable a much solid foundation, which can enable us to move faster in the long term.

Enforcing strict code review can be tedious. When I started my term as developer lead, I planned to speed up the code review process through automation.

### 3.1 Continuous Integration

Continuous integration (CI) is the practice of building and testing each commit in the repository. It is done automatically by machines and can be integrated with GitHub to display the status of each commit.

Continuous integration allows us to run all of the following checks automatically.

Linters can run during CI to show a list of all instances of bad code. Linter errors can even be annotated on GitHub's web interface to help developers find and fix them. Figure 5 shows an example of potential problems automatically caught and annotated by linters.

For projects that use statically typed programming languages, we can use relevant compilers during building. For dynamically typed languages, we can also at least check

```
34 +   const arrowKeyHandler = (event: KeyboardEvent<HTMLInputElement>): void => {
35 +     if (event.key == 'Down') {
```

⚠ Check warning on line 35 in frontend/src/components/Util/SearchBox/index.tsx

GitHub Actions / build

frontend/src/components/Util/SearchBox/index.tsx#L35

Expected '===' and instead saw '=='

```
36 +     //TODO, possible solution can be mapping Down key to tab since tab functionality works
```

✖ Check failure on line 36 in frontend/src/components/Util/SearchBox/index.tsx

GitHub Actions / build

frontend/src/components/Util/SearchBox/index.tsx#L36

Expected exception block, space or tab after '/' in comment

Figure 5: Automatic Linter Annotation on the Code

whether the program has syntax errors. As a result, no obviously broken code can slip through reviews.

We also build the entire project and run some unit tests and integration tests on them. Although our test suite is not very good at this point, it serves as a good smoke test to tell us whether basic features are working. Therefore, reviewers don't need to waste time to review code that fails those basic tests.

Continuous integration helps developers to fix problems at diff time, and study has shown that with more powerful tests and static analysis tools, we can catch more sophisticated problems that were once believed to be detectable by human reviewers, and fix them before they go into production.

### 3.2 Branch Protection and Pull Requests

Continuous integration is great, but itself alone cannot solve the problems. Developers can still choose to ignore the result and push to master directly. Therefore, we also need to set up branch protection to prevent that.

Branch protection is a GitHub feature that can fail a code push to a branch unless certain conditions are met. We can configure it in a way that will fail the push unless all continuous integration checks are passing. When this type of branch protection is enabled, pushing to master directly is disallowed. Developers must first create a new branch, push code to the new branch, wait until all checks passing and then merge. Figure 6 shows a typical Git workflow with required CI checks. We can use this approach to ensure our master branch is always working.

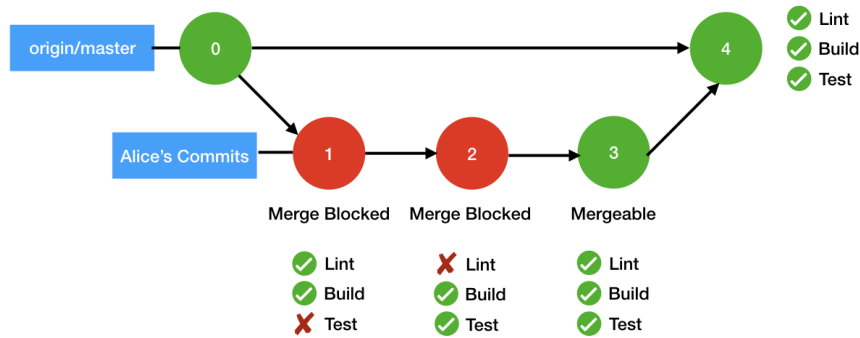


Figure 6: Git Workflow with CI

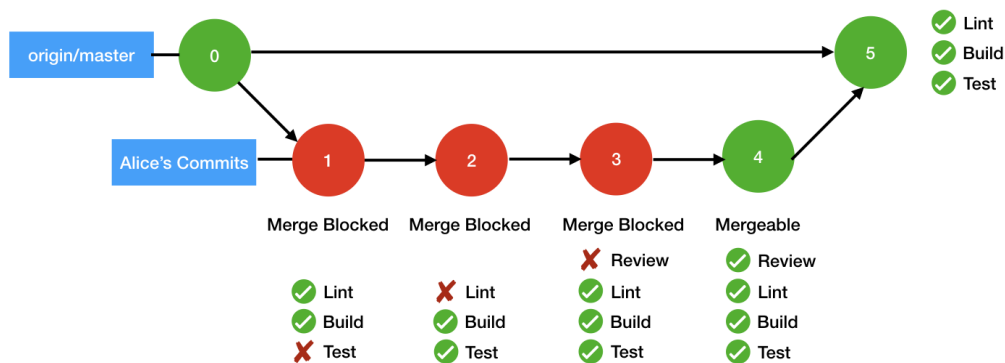


Figure 7: Git Workflow with CI and Code Review

Continuous integration is powerful, but it cannot catch all problems. Sometimes developers implemented something in an ugly way or with security vulnerabilities. These problems should be caught as early as possible before no one has any context of how to fix them.

Pull request is another GitHub feature that helps with this workflow. Once developers think that they have implemented what's required, they can open a pull request. Inside a pull request, reviewers can approve it or request changes. Branch protection can also be configured so that a branch cannot be merged into the master branch unless it passes the code review. Figure 7 shows a typical Git workflow with required CI checks and code review. We can use this approach to additionally ensure that all code pushed to master has been examined.

Since all the linter and compiler issues can now be checked automatically in CI, reviewers can spend most of their time looking for higher-level issues like clean code and good design, which in theory can result in much better code quality and faster review.

### 3.3 Continuous Deployment

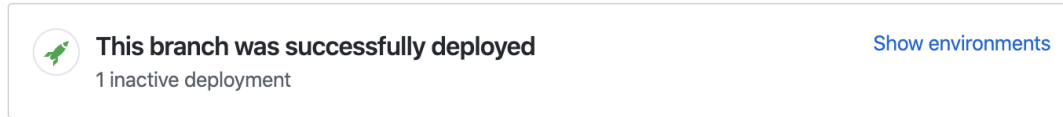


Figure 8: Deployment badge for each pull request

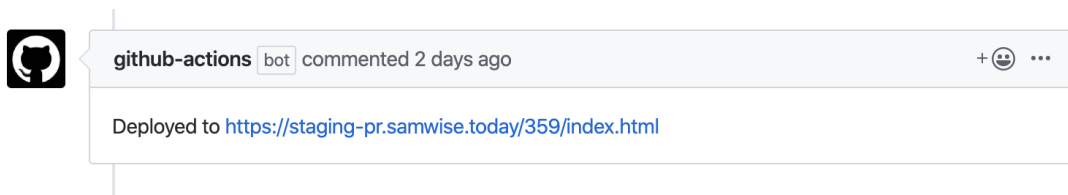


Figure 9: Bot comment on each pull request after build passes

Reviewers still need to switch to another branch and then running the code locally to ensure everything works. This is still tedious and reviewers may just give up and hit the approve button without actually looking at anything.

To combat the lazy reviewer problem, we decide not to blame the reviewers but to ensure that reviewers can easily see how changes affect real systems.

Continuous deployment is a practice that deploys each good commit (after passing) CI to some servers.

For Heroku apps, we use the review app feature to create a separate instance of server for reviewers to playtest the changes. Figure 8 shows a successful deployment.

For Firebase apps, I created a continuous deployment workflow to automatically deploy it, and let a bot comment on the pull request when it's ready. Figure 9 shows an example of a bot comment on pull request thread.

When the changes are eventually merged into master, the master code will then be automatically deployed to staging for everyone inside DTI to playtest it.

Continuous deployment not only ensures that we never push broken apps to production, but also its configuration file serves as a piece of self-documenting code that teaches new developers how to set up the local environment. It will never go out-of-date since otherwise the continuous deployment will noticeably break.

## 4 Gradual Rollout

The new workflow looks good in theory, but it has to be rolled out to prove itself. However, the rollout is not as simple as enabling branch protection and writing CI and CD configuration for every project.

Imagine that we have a codebase that has never enforced any kind of code quality and we now want to enforce the strictest possible code quality standard. Transitioning from a poorly written codebase to a clean one implies a lot of work, and it can take a lot of developers' time that could be spent on creating new features. We need a way to minimize the impact of code cleanup while ensuring that the cleanup will not be delayed forever.

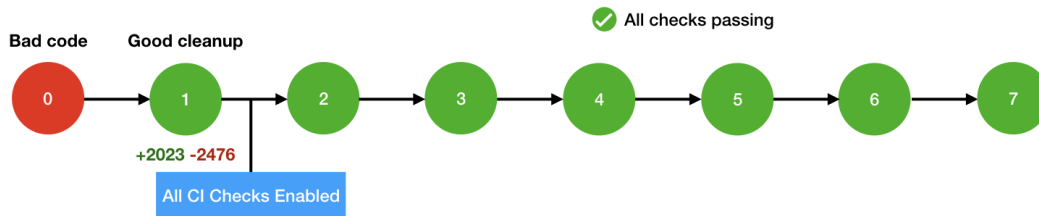


Figure 10: An idealistic massive codebase cleanup. Once cleanup is done, every commit is good.



Figure 11: A realistic massive codebase cleanup. Cleanup becomes messy and is reverted.

One possible approach is to let a developer who really cares about code quality to do a massive cleanup that fixes every problem in one commit. After this massive commit,



we can strictly enforce all CI checks and ensure all future commits are good. Figure 10 shows a possible commit-graph when everything works as expected.

However, this approach is based upon some overly optimistic assumptions:

- The developer can fix all problems.
- The developer will not accidentally break some features during cleanup.
- The developer can do the cleanup fast enough to avoid disrupting normal development.

If these assumptions are not true, then either the developer gives up under the pressure of feature delays or the massive cleanup commit gets reverted because it breaks too many features. While the developer is trying again, new features might already be developed, which makes the cleanup more error-prone. After several failed attempts of cleanup, the team may completely give up on code quality because improving it seems too difficult. Figure 11 shows how this approach can become very chaotic at the end.

It is also impossible to ask a team to stop developing new features until all code fixes are done, because we expect continuous progress in product development. Therefore, the only viable approach is to gradually strictify automatic code quality checks.

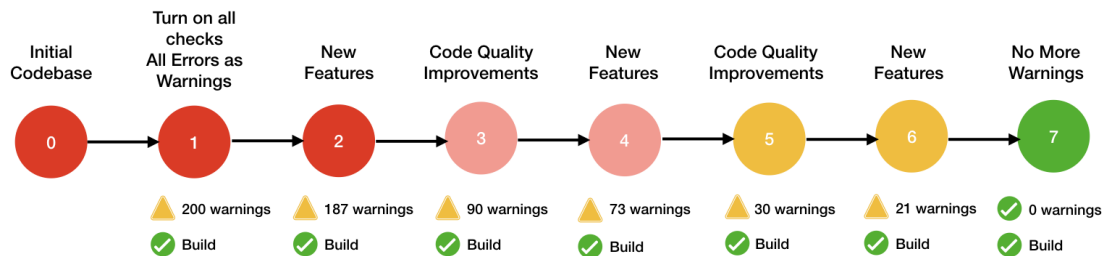


Figure 12: Gradually cleanup codebase. The number of warnings keep dropping.

```
Tue, 08 Oct 2019 02:38:55 GMT * 156 problems (0 errors, 156 warnings)
Tue, 05 Nov 2019 03:12:19 GMT * 117 problems (0 errors, 117 warnings)
Wed, 13 Nov 2019 18:51:52 GMT * 63 problems (0 errors, 63 warnings)
Mon, 18 Nov 2019 17:12:00 GMT * 29 problems (0 errors, 29 warnings)
```

Figure 13: Decreasing number of linter warnings

The gradual rollout process is illustrated in Figure 12. We first need to turn on all CI checks to ensure that the problems are highly visible to every developer in the team. However, instead of starting from the strictest standard, we start from loose ones. We convert all errors to warnings so that developers do not have to fix all problems at once. Instead, they can develop new features while fixing code quality issues.

In this process, we use CI to track our code quality improvement progress, and everyone can monitor the process without running the check manually every day. Figure 13 shows the CI logs on linter warnings of the same codebase at different times. We use this CI generated data to inform ourselves about the current state of code quality and prioritize either feature development or code quality improvements accordingly.

In the end, when all the warnings are addressed, we can turn them back into errors so that bad code will be impossible to merge in the future.

## 5 Results and Next Steps

At the time when this article is written, the number of obviously broken commit pushed to master dropped to zero for all projects with CI and CD enabled. For projects with CI and CD, we observed a significant reduction of review time for developers. As a result, developers created more small and self-contained pull requests which make review even easier.

With a stricter and more automated system in place, we plan to significantly increase testing coverage in future semesters to improve the quality of our apps and ensure developers can move fast without accidentally breaking things.