A GUIDE TO DEVELOPER COLLABORATION WITH GITHUB:
How to Implement a Code Review Process with GitHub and Collaborator
Collaborator is a code review tool that helps development, testing and management teams work together to produce high quality code.
GitHub is a website that wraps a software version control system, “Git”, and allows software developers to host their code online for free (though there are paid models available). If you’re a software developer, you know that Github is a repository for you to store, exchange, trade, and talk about code.

Web analytics company, Alexa, ranks Github as the 83rd most popular site on the planet. That’s particularly impressive when you stop to consider GitHub’s 14 million users and 35 million repositories since its launch in 2008. No wonder it’s the world’s largest source code tool on the market today.

GitHub has proven nothing short of a revolution for a lot of small, nimble organizations, startups, and cutting edge companies. For heavily regulated, locked-down enterprises, this effect is certainly muted, but it’s subtly perceptible nonetheless.

In fact, when SmartBear interviewed more than 600 software developers, testers, and IT/Ops professionals earlier this year, we found that GitHub is now the most widely used tool for repository management.

GitHub is changing a lot of things about software development, and this includes the nature of code review.

At the end of 2015, SmartBear made significant improvements to Collaborator’s integration with Github to make it easier than ever for teams that utilize Github to do more enhanced peer code reviews. If a pull request is initiated in GitHub, Collaborator will create a code review and build a link to the review directly in the GitHub conversation. Once the review is created, teams can utilize the capabilities of Collaborator’s robust dev collaboration workflow.

In this eBook, we’ll take a closer look at how GitHub’s revolutionary platform is changing the way development teams can work together. We’ll look at the different features offered in GitHub and show how using a code review tool, like Collaborator, along with Github can help improve collaboration across your development team.
An introduction to GitHub

If you’ve been writing code for a long time, you no doubt remember the bad old days of remote work when it comes to source code version control. At the time GitHub first started attracting notice, centralized version control schemes were the standard, and when you were somewhere the source control server wasn’t, things could get painful.

Git, the version control system upon which Github is based, changed all that with distributed version control. Git was the version control of Linux — a decentralized, democratic tool that could support ad-hoc, global collaborations. Github, the website, wrapped Git up and encouraged you to work with the full safety of source control wherever you were. And, don’t worry, they reassured you, it’ll be a breeze to sync back up when you’re connected again. And it was.

Github offered remote coding to an increasingly remote workforce.

Social Coding

GitHub is often described as a social network for programmers. The term “social coding” has even appeared in some of GitHub’s marketing material. It is a platform meant, specifically, for maximum interaction.

Sure, GitHub is a vehicle for open source contributions, but that’s hardly a difference-maker for them. SourceForge was around for a long time and it would host source control for open source projects for free. There have also been other communities oriented around contributions and code sharing, such as Code Project. GitHub, however, came along and truly married social with coding, introducing feeds, followers, ubiquitous collaboration tools, and even a social network graph.

The result is an unprecedented amount of enthusiasm for global sharing of code. Ten or even twenty years ago, you probably would have hoarded source code of a side project. You wouldn’t have wanted to give away your intellectual property and you’d probably also have been embarrassed until you could tidy it up to put your best foot forward. Now the default is to throw your side work up on
GitHub and show it off for the world and your GitHub followers to see. Code is being shared like never before.

**Pull Requests**

“I accept pull requests.” This is a bit of quasi-snark that a project owner on GitHub might toss your way to mean, “If you want something to be different, roll up your sleeves and fix it.” This is a perfectly normal and encouraged workflow on GitHub.

As a GitHub user, you can browse through public projects and choose to “fork” one, creating a copy of it. You can then set about modifying it to your heart’s delight, as is the case with any open source project. Where things get interesting is that you can then create a “pull request,” which presents the original project maintainer with a “hey, what do you think of this code”-o-gram. The maintainer then reviews the request and accepts it, if they’re so inclined.

It’s not that this workflow is new, *per se*. Open source contributions have been around for as long as, well, open source. It’s that GitHub makes this such a core and easy part of the experience.

**Distributed Version Control Systems (DVCS)**

Centralized version control falls more in line with how people reason about maintenance-oriented tasks. There is a centralized source of truth and collaborators jump through various hoops to be able to touch (or sometimes even access) it. Think of a website with read-only PDFs that you can download. Anyone can take a look, but only certain people with special access can generate a new PDF and change what everyone sees.

Distributed version control systems change the game. They change the PDFs to modifiable Word documents. Once the publisher puts them up, anyone can download them, modify them, share them, email them, etc. There is no “source” of truth. Each individual copy has its own history and they all share history up to the point of their copying and one-off modification, but there isn’t an authoritative “right” one, except by convention.

That is the key distinguishing feature with DVCS. A “source of truth” is a matter of convention, and each copy (repository) is a stand-alone, complete entity. This enables an explosion in workflow potential. Perhaps the most killer feature of this over a traditional, centralized approach is the ability to work without a dependency on network connectivity. Instead of it being prohibitive to
work on spotty WiFi, halfway across the world from the source control machine, you can now be productive and sync up with that repo later, when you’re in a better spot.

What does it add up to?

If you went back a decade and examined the properties of code review, you’d find a more formal affair. People would assemble in a conference room and plug someone’s laptop into the projector. Someone would have compiled a list of edited files over the past weeks or months, and then the group would settle in for the long haul to review all of them. Code reviews were more formal, buttoned-up affairs with lots of official signoffs and bullet point items to make sure that the company copyright occupied the first 50 lines of every source code file. The author of the code often prepared to defend his work as if it were a PhD thesis. It was a high-touch grind.

Social coding has accustomed us to a lot more back and forth and relaxed discussion of code. It’s common for us to have our code out there on display now. Even people with jobs that would never permit the use of GitHub for source control are on there, looking at code examples in gists or looking for ideas in open source projects.

The ubiquity of the pull request has democratized contributions a good bit because the barriers to entry for contributing to your favorite tool are substantially lower. Code review thus evolves to be less “master evaluating supplicants” and more an activity of simple collaboration. This removes some of the buttoned-up formality, to be sure.

The distributed and remote nature of the work on GitHub has also helped normalize a more granular level of communication around code. With incredibly inexpensive commit operations, people commit code more frequently and can reason about changes with more granularity. Combine this with pull requests, which more or less mandate a review, and you’ve suddenly normalized small, targeted code reviews.

Is GitHub the only thing driving these changes in code review? Of course not. But it’s highly influential and visible. Whether you use it or not, GitHub is definitely having an impact on what your code reviews look like.
Are GitHub Pull Requests Enough for Code Reviews?

It’s clear that GitHub is having a significant impact on how teams are collaborating on projects, and how teams are conducting reviews.

When a developer wants to make a change to some source code, they will clone, or fork, or otherwise get what is called a working copy of the source code on their computer. They will change, and compile, and test in that working copy that they have forked from the central GitHub server. When whatever they were working on is ready to become part of the official product, they will then create the famed pull request.

The pull request will be visible in the GitHub interface, and the proposed changes will be presented. A big green button will allow you to approve, and merge the changes.

As a developer with a GitHub account, you can download a copy of any code that someone has shared. Then you can change it, enhance it, whatever, and send a pull request with your modifications. Once that pull request is accepted, your contribution becomes part of that open source project. It’s a very powerful, and a very social thing.

This can also be applied to commercial software development. Pull requests in private repositories are gates that lead into your deliverables. Pull requests are the mechanism by which the development team changes the code, and managing that change can reduce the risk of defects sneaking out the door. If you don’t carefully inspect what’s going into your software, you may not like what comes out.

Pull requests provide some control over that change associated risk. They prevent rogue developers from submitting whatever code they please, defect ridden or not. If configured correctly, you can prevent the wild, wild, west model of software development, where it’s free reign, or open season on commits. You don’t want to just allow everything in; there should be a system of checks and balances.

GitHub provides a basic feedback mechanism for pull requests in the form of comments, and @mentions. When a pull request comes in, the owner of the destination branch will need to approve or deny the change. Before that happens, you can definitely have a conversation about the code, or even specific line numbers, before approving.
At the end of the day, it’s a yes or no, keep or discard, merge or don’t merge motion. The pull request is your first and last line of defense against terrible code.

But the question remains: are GitHub pull requests enough for code reviews?

The purpose for code reviews are as diverse as the environments in which they are conducted. However, almost all code reviews have these goals in common:

- Defect-free, well-documented software.
- Software that complies with enterprise coding standards.
- Teaching and sharing knowledge between developers.

Other objectives often include: maintainability, security, consistent end-user documentation, adequate comments in code, complete unit tests, and scalability.

Code review takes place during all stages of development, except with small projects such as demos and experiments that are designed to be written quickly and probably thrown away. Even during the final stages of development when everyone is trying to meet a deadline, code reviews greatly reduce the number of regression bugs and ensure that company coding practices are not abandoned.

While there are a number of benefits of using GitHub pull requests throughout these stages, there are also certain limitations of what you can actually do, as it relates to code reviews. Pull request comments aren’t the same thing as code review.

Some of the limitations we hear from our developer audience at SmartBear, include:

- **No iterative review process:** The review process between the author and inspector of the code can involve a lot of back and forth. It can be difficult to implement a structured feedback process with GitHub alone.

- **No distinction between comments & issues:** When you have a work item and participant in a code review, the issues that are found during that review are not necessarily meant to be new work items. A code review tools enables you to make comments, which the author can review and respond to. Or issues, which will need to be addressed before the review can be closed.
• **No workflow options:** The pull request workflow works well for some projects but some teams, especially those within enterprise organizations, will need to customize the workflow to fit in with their existing processes. One of the benefits of using a code review tool, along with GitHub, is that you can assign roles to those participating in the review and fit the tool into your existing workflow.

• **Not effective for large changesets:** Large changesets can get complicated fast, without a tool to track changes that have taken place. A code review tool improves traceability of your changes so that everyone involved in the review understands what is being done.

• **Minimal metrics that actually matter:** When it comes to code reviews, metrics matter. Two of the most important metrics – defect density and inspection rate – are only available with a robust code collaboration tool. These metrics help improve the code review process already in place.

GitHub is playing a critical role in how development teams collaborate on projects. But when it comes to implementing a code review process, teams need a tool that will enable them to scale their process. In the next section, we’ll take a look at how using a code review tool along with GitHub can benefit your organization.
Beyond GitHub: Using a Code Review Tool for Code Review

GitHub pull requests are the wave of the future in terms of managing changes to your source code, but what happens if you need more? What happens if you’re regulated, or audited, or you feel the need for enhanced reporting? Pull request comments aren’t the same thing as code review. Here’s how we enhance the process with SmartBear’s developer collaboration tool, Collaborator.

Collaborator integrates with GitHub to provide:

- An iterative review process
- Enhanced defect tracking
- Stellar metrics and reporting, including how much time was actually spent reviewing code
- Support for multiple workflows
- Custom fields on the header of the review, as well as per participant, and per defect.

Here’s a closer look at how it works:

A developer creates a new feature branch, or forks a repository, and begins to make some changes. Once they’ve come to a point where they are ready to get some feedback or deliver that code, they commit their code and create a pull request.

That’s where Collaborator steps in, automatically detecting the pull request, and creating a new review within the code review platform. The developer can then invite his or her colleagues to participate in the review, providing feedback in the form of comments and defects.

This is where the advantages of Collaborator start to become evident.

Defects and comments are not the same thing. Comments are just that, a conversation, whereas defects must be corrected by the author in order to complete the review. This provides a clear path for the code author; they know what must be corrected. Along the way, the reviewers can also classify the defect by severity, type, or you can create your own classifications by way of defect custom fields.
When the reviewers have finished inspecting the code, they will transition the state of the review back to the code author to correct the items that have been found. The reviewers can then look at the comments, and find all defects clearly printed out in Collaborator’s defect log. It’s now the author’s turn to work to fix those defects. Once they’ve finished that, they simply push a new changelist, or maybe even a few changelists up to the GitHub pull request, and Collaborator will attach them to the review.

So, to recap

The author was working on a feature or bug fix, and proposed a code change. They created a pull request, and got some feedback from colleagues. Said colleagues found some issues, and asked the author to fix them, and he or she did. After correcting the issues, the author pushed the new patch set to GitHub, and will now transition the review back to the reviewers for verification.

Did the code author successfully fix the defects? If so, the review may complete, and the code will be approved. Go ahead and merge in that pull request, it’s good to go. Chances are good that not every discovered defect is going to be fixed in a single round of rework. The reviewers can file more defects, add more comments, and send the review right back to rework if the developer is struggling to fix the existing defects, or if new ones are discovered along the way. The developer can then work to fix the items, and simply push a new changelist to the pull request when they need more feedback, or think they’ve achieved success. This process can repeat as many times as is necessary to get it right.

Implementing a code review process

With the right tools, your team will be able to implement a code review process to catch bugs, improve quality, and improve collaboration on your development team.

But what does an effective code review process look like? What is the best approach to code review? And how can you transform your organization to adopt a code review culture?

We’ll take a closer look at these questions in the next section.
Implementing a Code Review Process: 10 Steps to Follow

Development teams large and small have implemented successful code review strategies with Collaborator.

1. Review fewer than 400 lines of code at a time

A SmartBear study of a Cisco Systems programming team revealed that developers should review no more than 200 to 400 lines of code (LOC) at a time. The brain can only effectively process so much information at a time; beyond 400 LOC, the ability to find defects diminishes.

In practice, a review of 200-400 LOC over 60 to 90 minutes should yield 70-90% defect discovery. So, if 10 defects existed in the code, a properly conducted review would find between seven and nine of them.

When you’re ready to implement a code review process, there are a few important steps you’ll want to follow. Here are 10 to help get your team started.
2. Take your time. Inspection rates should under 500 LOC per hour

It can be tempting to tear through a review, assuming that someone else will catch the errors that you don’t find. However, research shows a significant drop in defect density at rates faster than 500 LOC per hour. Code reviews in reasonable quantity, at a slower pace for a limited amount of time results in the most effective code review.

3. Do not review for more than 60 minutes at a time

Just as you shouldn’t review code too quickly, you also should not review for too long in one sitting. When people engage in any activity requiring concentrated effort over a period of time, performance starts dropping off after about 60 minutes. Studies show that taking breaks from a task over a period of time can greatly improve quality of work.

4. Set goals and capture metrics

Before implementing a process, your team should decide how you will measure the effectiveness of peer review and name a few tangible goals. Start with external metrics. For example, “reduce support calls by 15%,” or “cut the percentage of defects injected by development in half.” This information should give you a quantifiable picture of how your code is improving. “Fix more bugs” is not an effective goal.

It’s also useful to watch internal process metrics, including:

- Inspection rate: the speed with which a review is performed
- Defect rate: the number of bugs found per hour of review
- Defect density: the average number of bugs found per line of code

Realistically, only automated or strictly controlled processes can provide repeatable metrics. A metrics-driven code review tool gathers data automatically so that your information is accurate and without human bias.
5. Authors should annotate source code before the review
Authors should annotate code before the review occurs because annotations guide the reviewer through the changes, showing which files to look at first and defending the reason behind each code modification. Annotations should be directed at other reviewers to ease the process and provide more depth in context. As an added benefit, the author will often find additional errors before the peer review even begins. More bugs found prior to peer review will yield in lower defect density because fewer bugs exist overall.

6. Use checklists
It’s very likely that each person on your team makes the same 10 mistakes over and over. Omissions in particular are the hardest defects to find because it’s difficult to review something that isn’t there. Checklists are the most effective way to eliminate frequently made errors and to combat the challenges of omission finding.

7. Establish a process for fixing defects found
Even after optimizing code review processes by time-boxing reviews, limiting LOC reviewed per hour and naming key metrics for your team, there’s still a key review step missing. How will the bugs be fixed? It seems obvious, but many teams do not have a systematic method for fixing the bugs they’ve worked so hard to find.

The best way to ensure that defects are fixed is to use a collaborative code review tool that allows reviewers to log bugs, discuss them with the author, and approve changes in the code. Without an automated tool, bugs found in review likely aren’t logged in the team’s usual defect tracking system because they are found before code is released to QA.

8. Foster a positive code review culture
While it’s easy to see defects as purely negative, each bug is actually an opportunity for the team to improve code quality. Peer review also allows junior team members to learn from senior leaders and for even the most experienced programmers to break bad habits.

Defects found in peer review are not an acceptable rubric by which to evaluate team members. Reports pulled from peer code reviews should never be used in performance reports. If personal metrics become a basis for compensation or promotion, developers will become hostile toward the process and naturally focus on improving personal metrics rather than writing better overall code.
9. Embrace the subconscious implications of peer review

The knowledge that others will be examining their work naturally drives people to produce a better product. This “Ego Effect” naturally incentivizes developers to write cleaner code because their peers will certainly see it. The SmartBear study of Cisco Systems found that “spot checking” 20% to 33% of the code resulted in lower defect density with minimal time expenditure. If your code has a 1-in-3 chance of being called out for review, that’s enough of an incentive to double-check your work.

10. Practice lightweight code reviews

Between email, over-the-shoulder, Microsoft Word, tool-assisted and hybrids of all types there are countless ways to collaboratively review code. However, to fully optimize your team’s time and to effectively measure its results, a lightweight, tool-assisted process is recommended.

The SmartBear study of Cisco Systems found that lightweight code review takes less than 20% the time of formal reviews and finds just as many bugs! Formal, or heavyweight, inspection averages nine hours per 200 LOC. While often effective, this rigid process requires up to six participants and hours of meetings paging through detailed code printouts.
Code Review that Helps Build Better Software and Teams

With support for all major SCMs and IDEs, Collaborator allows you to start reviewing directly in GitHub or any of your other favorite tools.

Collaborator extends the GitHub pull request capability by providing companies with a robust peer review workflow that can be configured to meet the entire organization’s needs. Collaborator’s iterative review process ensures that files have been reviewed and modified before they are delivered. Even if you’re not using GitHub, Collaborator’s iterative workflow can be used. In fact, Collaborator integrates with 11 different SCMs including Git, Subversion, Mercurial, TFS, Perforce and more.

Get started today. Download your free trial of Collaborator at SmartBear.com/Collaborator.
With the right tools and best practices, your team can peer review all of your code.