Code Review Patterns and Anti-Patterns

CONTENTS

WRITTEN BY JENNIFER MCGRATH
DIRECTOR OF PRODUCT AT GITPRIME

Introduction

Effective engineering managers are also effective debuggers. Effective managers view their teams as complex interdependent systems with inputs and outputs. When the outputs aren't as expected, great managers approach the problem with curiosity and are relentless in their pursuit of the root cause. They watch code reviews and visualize work patterns, spotting bottlenecks or process issues that, when cleared, increase the overall capacity of the team.

By searching for "why," they uncover organizational issues and learn how their teams work and how to resolve these problems in the future.

This Refcard is a collection of common team dynamics in the code review process based on observing several software teams.

Long-Running PRs

Long-running pull requests are PRs that have been open for a very long time (more than a week). A PR that doesn't close in a normal amount of time (within a day) can indicate uncertainty or disagreement about the code. Often, in long-running PRs, you'll notice a few back-and-forth comments, then radio silence.

Apart from the possible disagreement or confusion amongst the team, long-running PRs are also themselves a problem. A PR that is a week old can quickly become irrelevant, especially in fast-moving teams. Long-running PRs can also become bottlenecks.

HOW TO RECOGNIZE THIS PATTERN

Long-running PRs can be identified by viewing the team's pull requests over a period of time and then hovering of those that have been open for more than a day. If you see a few back-and-forth comments with signs of uncertainty or disagreement in their communication, followed by silence, it's worth checking in to see how you can move the conversation forward.

WHAT TO DO

It's usually best to first check in with the Submitter. It's their responsibility to get their work across the line, so they should be encouraged to bubble up disagreements or uncertainties as they arise. If there is a disagreement, get their read on it and offer advice to move it forward. Depending on the situation, get the Reviewer's read on it as well — ideally, when everyone is together in a room or on a call. Make a decision and ask anyone that disagrees to "disagree and commit."

GitPrime Press

20 Patterns to Watch for in Your Engineering Team

Get the book: GitPrime.com

To manage this pattern in the long-term, consider setting expectations or targets around time to first comment and time to resolve. It's also helpful to communicate best practices around timely response — when it takes engineers a day to respond to feedback, that can mean there's a lot of time spent waiting on others, and the communication isn't timely enough to be as effective as it otherwise could be.

Self-Merging PRs

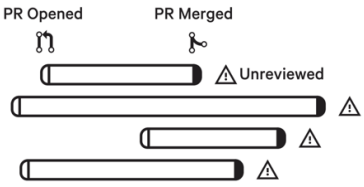
This pattern refers to when an engineer opens a pull request and then approves it themselves. This means no one else reviewed the work and it's headed to production!

As a general rule, engineers shouldn't merge their own code. In fact, most companies don't permit them to; self-merging bypasses all forms of human review and can easily introduce bugs.

If the code is worth putting on the main branch, it is worth having someone review it.

HOW TO RECOGNIZE THIS PATTERN

Self-merging is easy to see because the submitter and the reviewer are the same people and will show up as unreviewed.



WHAT TO DO

Many organizations prevent self-merging PRs by configuring their build systems to reject them. Enforced review is most common among companies that work under regulatory compliance, like fintech and biotech companies. Self-merging represents a material security risk to the company, no matter how talented an engineer is.

But even in organizations that don't enforce review, managers should be in the know when these situations do happen. Reviewing these PRs on a case-by-case basis, even though they're being reviewed after they've have been merged, will help ensure that any bugs or problems are not going to get buried.

If the commit was trivial, you might be able to give QA a heads-up to take a close look at it. If the unreviewed pull requests are non-trivial, walk those back if the circumstances allow it and require a code review.

Reducing the frequency of unreviewed and self-merged pull requests is a best practice (unreviewed PRs should be 0%, or close to it). If engineers are in the habit of self-merging without review, it may be helpful to have an informal conversation with them to ensure that they understand the "why" behind getting the review process or at least clear on expectations. If they're more senior, encourage them to follow the best practice of getting code thoroughly reviewed by others, so other engineers will model that behavior.

Heroing

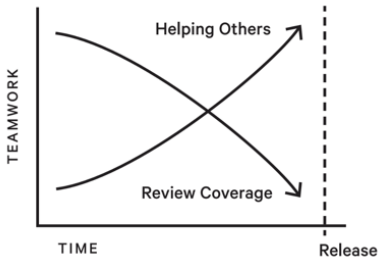
Right before a release, this engineer finds some critical defect and makes a diving catch to save the day. More formally, this pattern is the reoccurring tendency to fix other people's work at the last minute.

Granted, a good save is usually better than no save. But regular saves lead to the creation of unhealthy dynamics within the team or otherwise encourages undisciplined programming. Some team members even learn to *expect* them to jump in on every release.

This pattern can be a symptom of poor delegation or micro-management. It also points to trust issues on a number of levels. This engineer will ultimately undermine growth by short-circuiting feedback loops and, over time, can foster uncertainty and self-doubt in otherwise strong engineers. At its worst, this pattern feeds a culture of laziness: everyone knows the engineer will "fix" the work anyway so why bother. Ironically, those last-minute fixes are the genesis of a lot of technical debt.

HOW TO RECOGNIZE THIS PATTERN

Heroing is characterized by an engineer who is overly engaged in helping others, particularly in the form of late arriving check-ins. They're also distinguishable in the review process, where they may be self-merging PRs (and typically right before the deadline), or they will show very low receptiveness in the review process (meaning either others aren't providing substantial feedback or the engineer isn't incorporating it).



It can be hard to disagree with their changes — especially with these changes being made so late in the sprint. This is partly why this engineer's PRs usually show a very low level of engagement in the review process.

WHAT TO DO

Rather than managing the "saves," manage the code review process.

Ideally, team members are making small and frequent commits and requesting interim reviews for larger projects. If that's not the case, consider working toward that goal first. It'll help to get this engineer's feedback early, even before the code is done.

When the team is in the habit of getting feedback early and often throughout a project, as opposed to submitting massive PRs all at once, the barrier to participating in the review process is lower. This can make it easier to promote healthier collaboration patterns and get everyone — especially the engineer providing Late-Arriving Help — to give and be receptive to feedback in reviews. Coach this engineer to turn their 'fixes' into actionable feedback for their teammates.

Over Helping

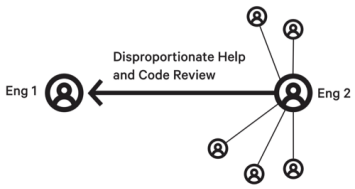
Collaboration among teammates is a natural and expected part of the development process. Over Helping is the pattern whereby one developer spends unnatural amounts of time helping another developer to get their work across the line.

Engineer One submits and Engineer Two cleans it up, over and over again. This behavior can be normal on small project-based teams. But when that 1-2-1-2 pattern doesn't taper off, it's a signal that should draw your attention.

The problem is threefold: (1) always cleaning someone else's work takes away from one's own assignments, (2) it impairs the original author's efforts toward true independent mastery, (3) it can overburden the helper and leave the original author in a continuous unnatural waiting state.

HOW TO RECOGNIZE THIS PATTERN

You'll notice this pattern in the same way you'd realize "heroing" in the review process and in viewing how these engineers are helping others. Look for reoccurring, last-minute corrections between the same two people.



In the code review process, you'll notice these two consistently review each other's work. One engineer will consistently modify the other engineer's recent work, but it's not reciprocated. The over-helping engineer will also show high levels of engagement and influence in the other person's PRs. The other engineer will not.

This behavior can be perfectly healthy and expected when in a mentorship-type situation. But beyond a certain point, rotation is in order.

WHAT TO DO

- Bring additional engineers into the code review process. A side effect of this solution is that by increasing the distribution of reviews, you're strengthening the team's overall knowledge of the codebase.
- Cross-train and assign both engineers to different areas of the codebase.
- Assign the senior engineer a very challenging project. The idea here is to give them challenging projects where they don't have the time or energy to review their friend's work.
- Lastly, the stronger of the two is showing natural leadership and coaching tendencies. Look for opportunities to feed this more broadly beyond their friend.

One note of caution: be mindful when the two engineers are friends or were colleagues at a former employer. Making light of a friendship or teasing them can be incredibly damaging and hurtful. Go the extra mile to keep it professional.

And, as always, be transparent. You're not trying to split up friendships. It's the manager's job to ensure that knowledge of the codebase is distributed evenly across the team and to ensure that people are honing their craft and growing their careers.

Just One More Thing

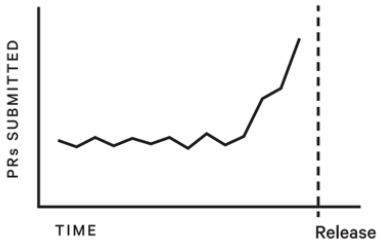
"Just one more thing" refers to the pattern of late-arriving pull requests. A team submits work but then jumps at the last minute in to make additions to that work.

Sometimes only one or two individual contributors will show this pattern, but that generally points to behaviors that require a different approach. But when the majority of the team is submitting PRs right before a deadline, it can mean there are larger process or even cultural issues that are causing an unpredictable workflow.

This pattern can occur for a wide range of reasons, including last minute requests, poor planning or estimates, and too much work in progress. In any case, late-arriving PRs hinder the team's ability to review the code properly.

HOW TO RECOGNIZE THIS PATTERN

"Just one more thing," when appearing across a team, is characterized by a spike in PRs being submitted near the end of a sprint *after* the main PR was approved. These engineers will also show a high level of New Work.



WHAT TO DO

Late-arriving PRs are a sign that work is being rushed and given less review. Even when the work is submitted by engineers who are very familiar with the code, the PRs should be treated as riskier than other equally sized commits that are submitted earlier in the sprint.

When you notice a spike in PRs being submitted, it can be helpful to review the work submitted and decide whether it should be given an extra day's review (pushing back the deadline).

Longer-term, consider working with the team to identify any bottlenecks or process issues (like planning) that could be eliminated or improved.

- If the team's estimates or deadlines are causing last-minute stress, consider setting internal deadlines for projects. Another framework that some teams use is to consider 'the three levers' in setting a deadline: the external deadline (if any), the scope of the project, and resources available. It's typically not realistic to change one without having to change the others, so it can help the planning process to take all three variables into account.

- If last-minute requests are coming in from outside the team, talking to the managers whose groups are regularly causing the problem can give you the opportunity to show the impact of the problem and understand what's going on from their perspective.

Rubber Stamping

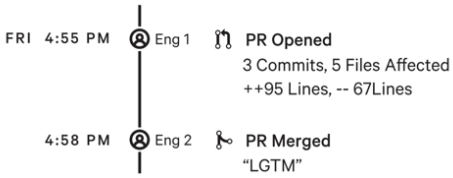
In politics, rubber stamping is the process whereby officials approve legislation without careful thought.

In software development, rubber stamping is the process by which an engineer approves their colleague's PR without giving it a substantial review. Often, the Submitter will have some level of seniority in the team, and the Reviewer trusts that the work is good enough. In other situations, someone doesn't value code review, or everyone just ran out of time and felt the need to push the PR through.

In any case, the code review process has a wide range of benefits and outcomes: teams see improved code quality, increased knowledge transfer within and across teams, more significant opportunities for collaboration and mentorship, and improved solutions to problems. So when an individual submits code for review and no review is given, we sacrifice all of these outcomes for short-term efficiency.

HOW TO RECOGNIZE THIS PATTERN

Rubber Stamping is most noticeable in the review process. Watch the team's review workflow for PRs that opened and closed in a preposterously short period of time, with a very low percentage of responsiveness (0%, or close to it). Low levels of engagement in reviews can also be seen in whether team members are incorporating feedback.



When review happens later (extending the time to resolve), their PRs will show little to no back-and-forth discussion. If there were no comments on the PR, this PR will show as unreviewed.

WHAT TO DO

While reviewing other people's work is a substantial part of what it means to be a professional software developer, it's not always recognized as such. Rubber Stamping often occurs in environments where the review process is given little attention or recognition; when leadership praises the behaviors they want to see in code reviews, we generally see that the way people work will shift to match those expectations.

On an individual level, it can be helpful to coach team members on what substantial reviews look like in practice by showing examples from others on their team. Try deconstructing the feedback together in a 1:1, so engineers leave the meeting with a framework they can use moving forward. If there are specific engineers who are frequently given less review, take into consideration how they're responding to any feedback in the process, how large their PRs are, or the time at which their PRs are submitted.

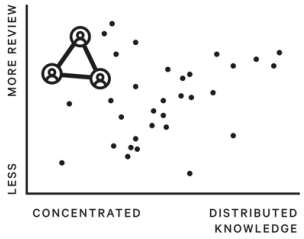
Knowledge Silos

Knowledge silos are usually experienced between departments in traditional organizational structures, but they also form within teams when information is not passing freely between individuals.

In software engineering, Knowledge Silos can be identified in the code review process. Knowledge silos form when a group of engineers only review each other's work. Imagine two or three engineers who review all of each other's PRs, and don't review anyone else's PRs on their team. These engineers learn about each other's work and techniques, and the areas of the code that they're working in. Other engineers on the team who aren't part of the silo don't have that same level of information.

There are plenty of reasons why engineers will get into a cycle of only reviewing each other's work — figuring out the reasons why, through discussions with the team and by reviewing the communication patterns in the review process, can sometimes point you toward the broader team dynamics at play. For example, If these engineers only want to work together because everyone else is slow to review their code, consider setting expectations around Time to First Comment, and Reaction Time.

When knowledge silos exist for an extended period of time, they can often begin to show signs of Rubber Stamping. Reviewing a select group of engineer's work for a long time can lead to less substantial reviews simply because the engineers trust that each other's work is good enough. When that happens, these situations can turn into bug factories. Work is being approved and pushed forward without adequate evaluation.



HOW TO RECOGNIZE THIS PATTERN

When team members are co-located, a basic understanding of where people sit in an office along with an awareness of any other social bonds can be helpful indicators as to where silos may form.

You can also measure the team's distribution of knowledge by visualizing who reviews whose PRs to identify knowledge silos. High-level metrics on these dynamics will provide insight into trends. Are there silos forming or are we trending away from silos and toward more equal distribution of reviews?

You can then drill down into specific team dynamics. When there are Silos, there will be a small group of engineers that review only each other's work across multiple sprints.

WHAT TO DO

Bring in the outsiders! One of the most natural ways to manage this pattern is to look for outliers and stranded engineers and get those individuals involved in the review process. Consider also anyone who

could be cross-trained or onboarded on a specific area of the code than an engineer within the silo is working on.

Assign other engineers to review the work of the individuals that make up the silo and have the individuals within that tight-knit group review the work of others outside their group.

A High Bus Factor

"Bus factor" is a thought experiment that asks what the consequence would be if an individual team member were hit by a bus. More specifically:

Bus factor (noun): The number of team members that need to get hit by a bus before your project is doomed to fail.

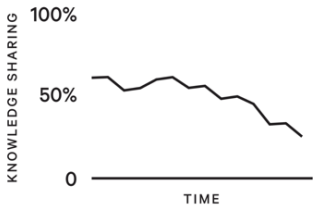
Having a low bus factor is risky. A high bus factor means that there is a greater distribution of knowledge and know-how about the code across the team.

When more than one engineer knows about each area of the team's code, there's more optionality for managers to assign tasks and more people that can provide substantial reviews, reducing the possibility of bottlenecks to a release.

For example, if three engineers know how to work in the billing system, a manager can assign a task in that domain to any of those three engineers. Contrarily, if there are knowledge silos, or if only one engineer has experience working in the billing system, the manager will have difficulty assigning those tasks to any other engineer.

HOW TO RECOGNIZE THIS PATTERN

A team's distribution of knowledge can be visualized with the knowledge sharing index, or an index that looks at who reviews whose PRs. It's best to use this report within teams that you would expect to review each other's work. A low index means that there is a lower distribution of knowledge across a team, representing a higher bus factor risk. A low index means there may be silos forming; a high Index represents a greater distribution of knowledge across the team.



Furthermore, it helps to start with the Index to get a high-level understanding and then drill down into specific team dynamics. If the Index is trending downward, check to see if team members are getting into a cycle of only reviewing each other's work. Otherwise, visualize how the team is sharing the responsibility of review and how everyone is participating, and recognize that behavior publicly.

WHAT TO DO

A high bus factor can be achieved when team members are making small and frequent commits, and there's a healthy level of collaboration and debate in reviews from everyone on the team. It can be helpful to keep this in mind when providing feedback in 1:1s and when onboarding new hires to the team.

When you see a low sharing index (e.g. a low bus factor, higher risk), see who usually reviews whose PRs on the team for opportunities to get team members more involved in the review process. When you see the behavior that you want to see in the review process, consider recognizing that in a team-wide meeting.

Written by **Jennifer McGrath,** *Director of Product at GitPrime*

Jennifer is a product leader with 20 years of experience in engineering and product leadership. She is Director of Product at GitPrime and was previously Director of Product Management at Vantiv. When she's not wrangling product ideas, she's outside rafting, hiking, skiing, or watching her kids do sports!

DZone
A DEVADA MEDIA PROPERTY

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects, and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code, and more. "DZone is a developer's dream," says PC Magazine.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300