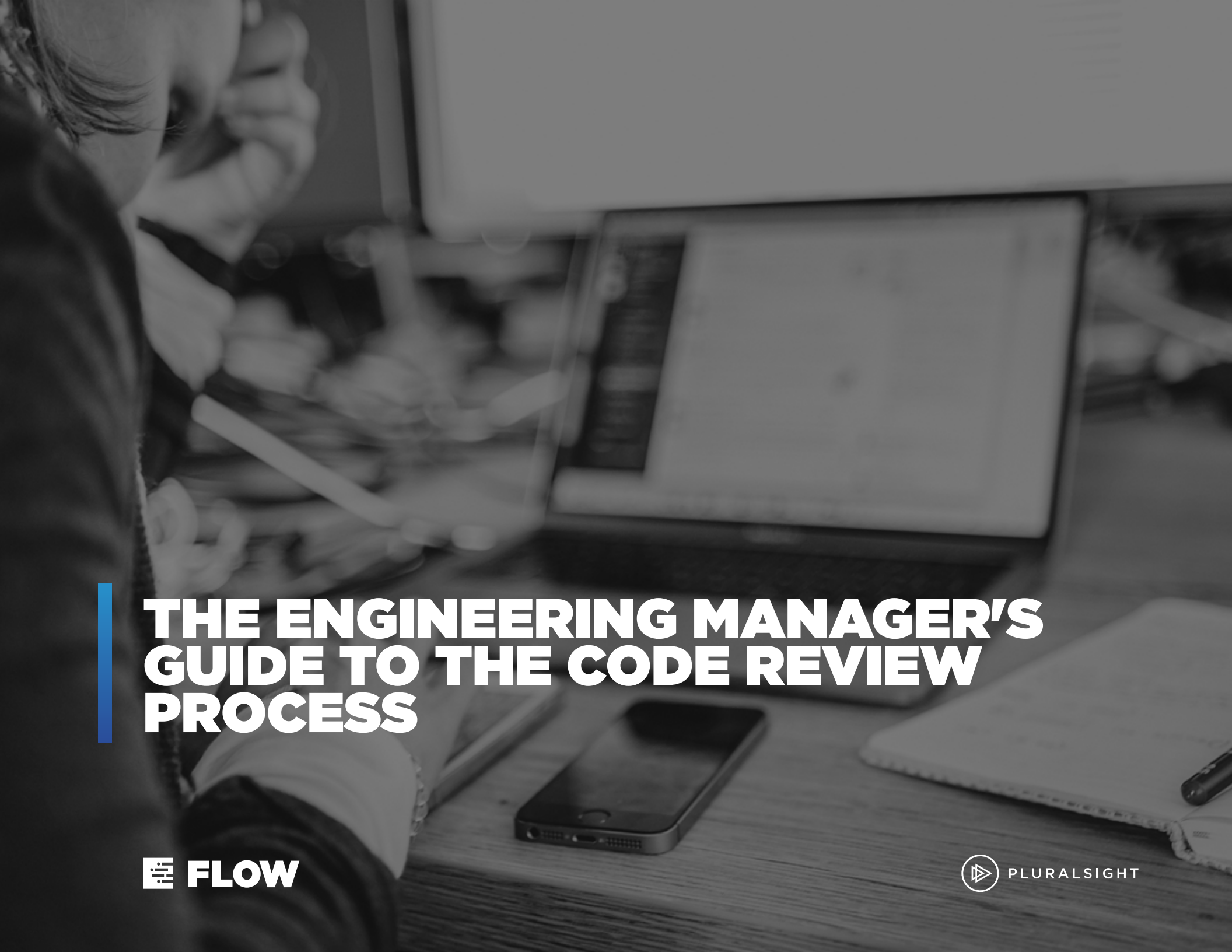


THE ENGINEERING MANAGER'S GUIDE TO THE CODE REVIEW PROCESS

FLOW

PLURALSIGHT

THE OUTCOMES OF CODE REVIEW, AND HOW MANAGERS CAN IMPROVE THE PROCESS IN A WAY THAT BENEFITS THE TEAM AS A WHOLE.

The general purpose of code review is for development teams to recognize and remedy bugs before code hits production. That's the traditional view. Yet we also believe that code review can accomplish much more than just that one pragmatic outcome—especially if leaders and managers become meaningful participants in the review process.

Beyond fixing bugs, code review (we use the words PR, pull request, and code review interchangeably) results in higher quality code that is more broadly understood across a team. Studies show that this process saves money, reduces reliance on QA, and improves engineering development, knowledge sharing, and the overall culture of the team in addition to the quality of the code. It's also an opportunity for engineers to collaborate, learn from their peers, practice mentorship, and discover improved solutions to problems.

In this guide, we've assembled foundational resources for software engineering leaders to be able to communicate concisely about the outcomes of code review (both the traditional goal of higher-quality code as well as improved collaboration and problem-solving within teams). This guide also highlights the manager's role in the code review process, including eight review dynamics common to engineering teams. These dynamics include ways to recognize various working patterns in teams, and ways to leverage these insights to coach the team towards more sustainable practices and help reduce any friction in working together.

OUTCOMES OF THE REVIEW PROCESS

By its peer-review nature, code review is conducted by the same engineers who are also writing and submitting code. That means time spent reviewing code seems, on the surface, to compete with time spent creating it. Ultimately, though, PRs are a strong investment in the continuous health of a product.

Leveraging code review prior to production streamlines the development process, highlights bottlenecks and process issues, and sets the stage for improving the overall health and capacity of teams. Essentially, code review enables collaboration between engineers and a higher quality of production.

HIGHER-QUALITY CODE

Even as recently as a decade ago, the primary (and often sole) purpose of conducting code review was to ensure the quality of the code. There are different ways of doing so, of course. But none of them can compare to peer review.

In Code Complete: A Practical Handbook of Software Construction, author Steve McConnell identifies that the average defect detection rate for unit testing is a mere 25%, with function testing's rate at 35% and integration testing's at 45%. Likewise, he states that the average effectiveness of design inspections is 55%, and 60% for code inspections.

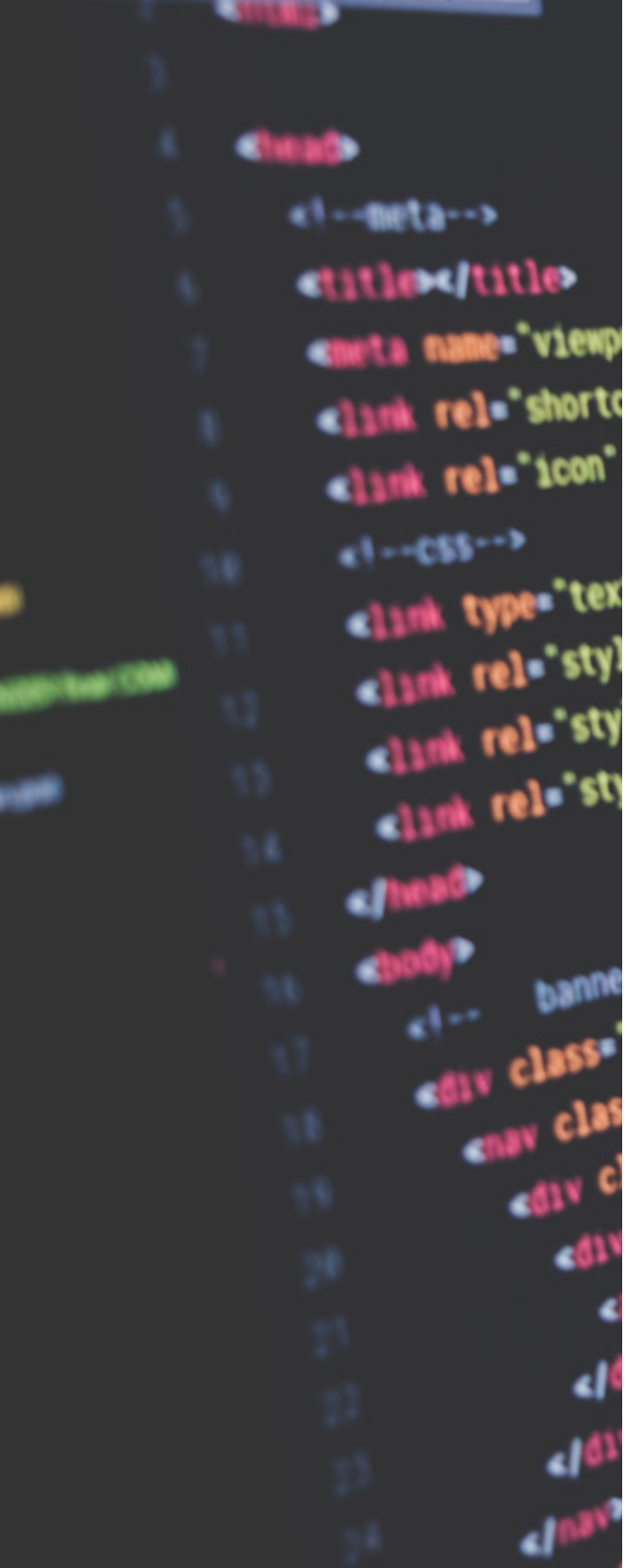
Code review has been shown to decrease errors by a staggering 80%. For example, McConnell points to a study that examined a group of 11 programs developed by the same team—five of which underwent no review at all. Those five programs contained, on average, 4.5 errors per 100 lines of code. On the flip side, the six programs that underwent code review averaged only .82 errors per 100 lines.

Some notable companies have shared research that indicates the use of review as a vital component of saving money and reducing errors:

- IBM's 500,000-line Orbit project used 11 levels of inspection and had only about 1% of the errors that would be expected without review—all while achieving early delivery.
- A study of a 200-person organization within AT&T introduced code reviews and subsequently experienced a 14% increase in productivity and a 90% decrease in defects.
- Jet Propulsion Laboratories estimates savings of approximately $25,000, simply by identifying (and repairing) defects in review at earlier stages.

A large-scale study of coder review coverage and participation at the University of California-Berkeley shows that code review reduces the number of bugs and security bugs in production. The research also suggests that implementing code review policies may have a positive effect on the quality and security of software.

TL;DR? Yes, code review is time spent not writing code, yet the process actually generates higher integrity within the code and does so more productively. It ultimately improves efficiency while maximizing your engineers' time and talents, as well as other resources.



COLLABORATION AND IMPROVED SOLUTIONS TO PROBLEMS

As software development methodology has evolved, so has our understanding of code review. It's no longer solely about ensuring the quality of the code—it's also become a realm (and perhaps THE realm) for engineers to relay information, learn from each other, and develop as creative professionals.

In so many ways, engineering is a loner's activity. Earbuds in, head down, no distractions, and it's amazing what an engineer can create in a day. But engineering teams are teams for a reason—collaboration, such as what PRs enable, contributes to monster levels of knowledge-sharing and therefore more creative, powerful solutions. As they say, software development is a team sport.

Researchers conducted a survey with 645 top contributors to active OSS projects. The results suggested that engineers have a strong interest in maintaining awareness of project status to get inspiration and avoid duplicating work—but they don't tend to proactively propagate information. In other words, there's often a delta between how engineers want to leverage the review process and how it's actually being leveraged (in the study, it said that many times the communication was limited to low-level concerns). The study pointed out that the challenges that are causing these barriers that limit the outcomes of code review are mostly social in nature. This presents a clear opportunity for managers to facilitate discussions in retrospectives or standups about the "state of the code review process" within that team, and how it might be improved.

Furthermore, by making code review (and therefore code development) a true team sport, organizations make themselves more resilient and adept by managing and reducing knowledge silos. Atlassian's guide to agile development contends that agile teams realize such benefits when code review decentralizes work across the engineering organization. Specific knowledge about the code base is not exclusive only to one team or even one engineer.

Code review, therefore, can serve as an antidote to (largely unintentional) knowledge hoarding. The process inherently encourages knowledge sharing, engineer participation, and collaboration. In many ways, this process has even replaced much more traditional training—which used to be an engineering manager's highest impact activity.

No more. Now, an engineering manager can realize the greatest gains by developing a healthy code review dynamic within the team. Traditional reviews were motivated by the drive to reduce errors. Today's reviews offer managers opportunities to guide their teams toward practices that promote healthier, more engaging, and more constructive conversations within the team.

THE ENGINEERING MANAGER'S ROLE IN THE CODE REVIEW PROCESS

Ultimately, the job of engineering leaders is not to code—it is, instead, to remove obstacles so their teams are able to spend more time working on valuable solutions and so their work output has the reach, impact, and visibility it deserves. So it would be easy enough to designate code review as the engineers' domain, where managers need not tread.

But that makes for a massive missed opportunity.

Without guidance and a healthy review culture, the PR process can disintegrate into unproductive, though understandable, behaviors. Some developers see code review as pulling them away from their true work, and it's impossible to deny that review is a somewhat subjective process that, unshepherded, can lead to disagreements, stalled commits, and even outright hostility.

Many individuals also have not experienced the art of accepting and giving criticism, and therefore haven't learned it. Unintentional communication breakdowns can lead to both social and technical frustration, which of course gums up the works too.

These sorts of PR environments are not sustainable or healthy. That's why managerial support is critical, if reviews are to become opportunities for teams to learn from each other and work toward more effective, more creative solutions.

This holds true with engineers of all levels. Engineers new to the organization (or new to the industry) glean the team's culture, pace of work, style, and implicit coding standards through involvement in the PR process. Senior developers are able to coach more junior ones in their domain expertise, and for engineers of all levels, code review is a chance to identify strengths and weaknesses (both individually and organizationally). And for co-located and distributed teams alike, code review is perhaps the richest opportunity for work-centric socialization and team-building.

So where do managers take part in this process?

Where training was once at the heart of management, code review is now the prime way of improving an engineering team's output. By participating in and observing code review, managers are able to track the health and productivity of the team, which provides insight into where to intervene and where to encourage progress.

In other words—the team is responsible for creating code and for peer reviewing that code. Managers are responsible for the behavioral trends exhibited in their teams' code reviews.

In our work, we've learned to recognize common patterns exhibited by software engineering teams—both successful patterns that can be nourished, and problematic ones that an aware manager can remedy. Here, we've assembled eight of those dynamics that demonstrate the behaviors common to developers and to engineering teams, how to recognize them using Flow metrics, and what managers can do to bolster their teams' health and productivity.

COMMON DYNAMICS TO IDENTIFY IN THE CODE REVIEW PROCESS

LONG-RUNNING PRS

Long-running pull requests have been open for an unusually long period of time. Organizations ship at different rates, so a PR that stays open for 5 hours could be long-running for one organization, where 24 hours will long-running for another. Sometimes, PRs will stay open for several days.

There are a number of reasons why a PR might stay open for an extended period of time:

- There's uncertainty or disagreement about the code (which can reveal itself with a few back-and-forth comments earlier in the PR followed by silence)
- There are large spaces of time between comments and responses in the review
- The PR is massive (think: multiple days' or even weeks' worth of work) and team members are avoiding having to review all of that code
- The PR was submitted at 5pm on a Friday, so the review didn't start until the following Monday at best

Apart from being symptomatic of possible disagreement or confusion within 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. In short, long-running PRs are bottlenecks to a release.

How to recognize them: Long-running PRs can quickly be identified in the team's *Review Workflow* report, filtered by "PR Status: Open" and sorted by "oldest PRs." Select the number of PRs you'd like to see in one view, then hover over 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 the communication, followed by silence, it's worth checking in to see how you can move the conversation forward.

What to do:

- **If there are signs of disagreement or confusion in the discussion:** 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 who disagrees to "disagree and commit" for the sake of the team's progress.
- **To manage this pattern in the long-term, or if there are large spaces between comments and responses in the review:** Set expectations or targets around *Time to First Comment* and *Time to Resolve*. (Both metrics can be found in Flow's *PR Resolution* report.) It can also be helpful to communicate best practices around responding to colleagues in a timely manner. When it takes someone a day to respond to a comment, 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 could be.

HEROING

Heroing is the reoccurring tendency to fix other people's work at the last minute. Right before each release, the Hero finds some critical defect and makes a diving catch to save the day.

Of course, attention to detail is essential and a good save is usually better than no save. But regular Heroing leads to the creation of unhealthy dynamics within the team or otherwise encourages undisciplined programming. Some team members even learn to expect Heroes to jump in on every release.

Heroing can be a symptom of micro-management or poor delegation. It also points to trust issues on a number of levels. Heroing 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, Heroing feeds a culture of laziness: everyone knows the Hero 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 it: The Hero typically dominates Flow's *Help Others* metric, 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 Hero 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 the Hero's PRs usually show a very low level of engagement in the review process (see the *Review and Collaboration* metrics).

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. Getting the Hero's feedback early, even before the code is done, will help improve the problematic tendencies.
- 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 Hero—to give and be receptive to feedback in reviews. Coach the Hero to turn their "fixes" into actionable feedback for their teammates to implement with time to spare.

OVER-HELPING

Collaboration among teammates is to be expected, as it is a natural part of the development process. However, "**Over-helping**" can occur if one developer spends an unnatural amount of time helping another developer get their work across the line.

Engineer One submits. 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 can 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 it: You'll notice this common dynamic in the same way you'd realize Heroing in Flow's *Review and Collaboration* reports and the *Help Others* metric. Look for reoccurring, last-minute corrections between the same two people.

In the *Review and Collaboration* and *Operational* reports, you'll notice these two engineers consistently review each other's work. One engineer will have a high *Help Others*, but it's not reciprocated. The load-bearing engineer will also show high levels of *Influence* and *Review Coverage*. The other engineer will not. One engineer will have a high *Impact*; the other won't.

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 colleague's work.
- Lastly, the stronger of the two is showing natural leadership and coaching tendencies. Look for opportunities to feed these skills more broadly to the whole team.
- 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.

OVER-ENGAGEMENT

It may seem counterintuitive that over-communicating in code review can be a problem. After all, we want our teams to be excited about—and engaged in—their work. However, since over-commenting comes at the expense of an engineer's own deliverables, such over-engagement can affect team morale because some engineers are not achieving what is, ultimately, their responsibility to accomplish.

The problem arises when over-engaged engineers spend a disproportionate amount of their time providing feedback (helpful or not) on other people's work and too little time working on their own projects. The healthy mix of reviewing and contributing is thrown out of whack.

Furthermore, while gregarious teammates often ignite interesting and creative conversations, this problem emerges when their PR contributions tip from beneficial commentary to comment spam. Their contributions may make them appear busy, but the data shows this behavior is not constructive.

How to recognize it: As a manager, you may miss that this is happening, because most of what you see appears to be sound engagement. But the teammates know it's going on. They're on the receiving end of that over-engagement, and they're acutely aware of it—they just might be unable to talk to you about it.

You may first spot this dynamic in the *Review Workflow* when you notice a team member is frequently engaging in the PR process as a Reviewer. Perhaps they're commenting, providing feedback, or otherwise showing a strong level of engagement with other people's work. This behavior is only something worth noticing when the feedback that's being provided is going far beyond what's "good enough" for that specific project. If you notice a highly engaged Reviewer, you can go to their *Player Card* to see their level of *Involvement* and *Influence*—and evaluate that in context with their *Code Fundamentals* ("Is this team member spending more time in review than on their own work, and is that expected?"), and with the team's averages for that time period.

What to do:

- It helps to remain aware that over-engaged engineers may have come to believe through learned behavior that commenting in this way allows them to contribute most helpfully to the team. Helping others is helpful, right? So as a leader, you need to ensure that you're setting proper expectations and guiding these engineers to correct course.
- It can be challenging to tell these engineers not to comment so much, because they often feel they are contributing positively. A more constructive approach is to add projects to their bucket of work over the next couple sprints. As they feel the pressure of hitting their deliverables, they'll naturally tend to scale back on their commenting activity. That's when you give them an out.
- By doing this, you can let them know, "You clearly have a lot of work on your plate. If that needs to come at the expense of some of the code review you've been doing, that's totally okay." Just giving them that little out is usually all you need to restore balance on the team. That engineer is hitting objectives for the sprint, and the team feels overall happier and more harmonious.

JUST ONE MORE THING

Just One More Thing refers to a pattern of late-arriving pull requests. A team submits work, but then—right before the deadline—they jump in and make additions to that work.

Sometimes only one or two individual contributors will show this pattern, and that generally points to behaviors that require an individual. **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.

How to recognize it: 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 PRs 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.
- Longer-term, consider working with the team to identify any bottlenecks or process issues that could be eliminated or improved.
- If the team's **estimates or deadlines** are causing last-minute stress, consider setting different 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 the 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 stakeholders or 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.

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. They form when a group of engineers review only each others' work.

Imagine two or three engineers who review all of each others' 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, while 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 reviewing only each other's work — figuring out the reasons why, through discussions with the team and by reviewing the Team Collaboration metrics, can sometimes point you toward the broader team dynamics at play. For example, if these engineers want to work together because everyone else on the team is slow to review their code, you can consider setting expectations around Time to First Comment and *Reaction Time*.

Whatever the cause, reviewing a select group of engineers' work for a long time can lead to less substantial reviews simply because the engineers trust that each others' 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 it: 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 use the *Knowledge Sharing* report to visualize how knowledge is being distributed across a team in the review process and to identify knowledge silos. If there are two or three people who review only each others' code, the team's *Knowledge Sharing Index* will trend toward 0. If the majority of the team reviews each others' code, the Index will trend toward 1.

You can then drill down into specific team dynamics with the *Review Radar*. When there are silos, there will be a small group of engineers who review only each others' work across multiple sprints.

What to do:

- Bring in the outsiders! Look for outliers and stranded engineers and get those individuals involved in the review process. You can also see whether there's anyone who could be cross-trained or onboarded on a specific area of the code that an engineering 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.

SELF-MERGING PRS

Self-merging pull requests refers to when engineers open a pull request and then approve it themselves. This means no one else reviewed the work and it's headed to production.

As a general rule, engineers should never merge their own code. In fact, most companies don't permit them to: self-merging bypasses any form of check on the code, as well as skipping the opportunity for improvement and learning.

If the code is worth putting into the main code branch, it is worth having somebody review it. Self-merging represents a material security risk to the company, no matter how talented an engineer is. Yet as a practical matter, unreviewed pull requests happen a lot, for any number of reasons.

How to recognize it: Self-merging is easy to see because the submitter and the reviewer are the same people. In Flow, these instances will show up in the team's Unreviewed PRs metric as well as in the *Review Workflow.*

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 or Biotech companies.
- 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 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 the review process or that they are 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.

RAMPING UP

We cannot overstate how helpful the PR process can be in encouraging the team to consistently learning from each other. Code review is always a chance for cross-pollinating information and expertise between team members. So using PRs in the process of onboarding and ramping up engineers new to the organization is a great way to build connections between developers who will be working together.

A key component of successful engineer onboarding is ensuring that new arrivals are learning both the code, as well as how to engage meaningfully with their new teammates.

If you ask your engineers—and particularly recent hires—how things are going, you're bound to get the initial "Great" answer. Things may be great, and they often are. But sometimes they may be going off track like Han Solo in a botched princess rescue, bluffing their way by saying, "Everything's fine, we're all fine here now, thank you…How are you?"

So you can rely on the data to show you how things are truly going.

What to do:

- Use the data to understand the dynamics of a new hire's review engagement. By looking at PRs Submitted and PRs Reviewed, you can identify where new hires are interacting with their teammates. If you've paired engineers together, you can see where those interactions are going well, as well as where new hires have jumped in on other engineers' work too. After all, commenting on multiple people's work will build rapport within the team over time.

- The data can also help you understand the quality of their comments in the review process by examining the responsiveness and review coverage of both the new hires and the experienced developers. This will demonstrate the traction the new hires are getting within the team, and where engineers may be dragging their feet, you can use this opportunity to remind them about the importance of welcoming new team members into the fold.

Remember that PRs are about so much more than finding bugs. While ramping up, as well as throughout engineers' time with the team, code review is a fantastic way to reinforce cultural values. By encouraging engineers to participate more in the PR process, not only are you encouraging best practices, you're also empowering them to speak their minds and use their voices. "You were brought onto the team for good reasons," you're saying. "Your team wants to hear what you have to contribute."

Participation as a newcomer also reinforces the notion for the established engineers that they need to be open to receiving and accepting feedback. "This is an important part of who you are," you're saying. "And this is an important value for this organization."

In short, PRs are one of the best ways not to just talk about your values but to actually live them, every day.

And that's true not just in the onboarding process, but in your day-to-day, too. Developing software is no longer about individual engineers cranking out code (if it ever was). It's a team effort. By focusing on these dynamics of the code review process and utilizing the data available to you, you can improve the value your team gleans from the code review process—and start guiding better engineers and better products as a result.

CONCLUSION

Ultimately, code review is about so much more than just catching errors. It's a place where the team can work together to create even better solutions for customers. The review process is where team members can share knowledge, provide feedback, learn from one another, and build a culture that supports healthy collaboration patterns.

And managers can provide the most high-value contributions not by participating in those reviews, but by looking at the process as a whole and noticing quick wins and areas where the team could work better together. A leader's role is to remove obstacles that block developers from doing their best work, and to coach team members toward healthy work and collaborate patterns. They work on the process, rather than in the process.

We hope this guide helps deepen your understanding of the practical benefits of code review, as well as the manager's capacity to support the team in reaching their potential.

For further reading, we've expanded on these patterns to identify in our recently published book: 20 Patterns to Watch For in Your Engineering Team, which you can download for free.

FLOW

BUILT BY ENGINEERS, FOR ENGINEERS

Pluralsight gives you the confidence you need to accelerate velocity by providing visibility into your software engineering process.

ENGINEERING LEADERS HAVE BEEN OPERATING IN THE DARK

For many organizations, software engineering is one of the most expensive and mission-critical departments. Companies invest millions of dollars in software engineering without a feedback loop to understand how well they're doing or where to focus on improvement.

FLOW TURNS THE LIGHTS ON WITH OBJECTIVE DATA

Flow generates actionable metrics to optimize release processes, improve collaboration workflows and remove bottlenecks while creating unprecedented visibility for all levels of management.

GET DEEP VISIBILITY INTO YOUR DEVELOPMENT PROCESS

Flow instruments the tools in your development workflow—from commit data, pull requests, tickets and more—to provide actionable insights into individual and team workflows.

TURN WORKFLOW DATA INTO OPERATIONAL IMPROVEMENT

Flow gives software leaders a fact-based view of effectiveness and performance with prescriptive metrics to drive process improvement. The end result is improved quality, more time spent coding, healthier distribution of knowledge, and faster time to market.

PLURALSIGHT