GitPrime Press

20 Patterns to Watch for in Your Engineering Team

A field guide to help you recognize achievement, spot bottlenecks, and debug your development process with data.

20 Patterns to Watch for in Your Engineering Team

A field guide to help you recognize achievement, spot bottlenecks, and debug your development process with data.

Ed 2019.03.18

Table of Contents

PART 1

Domain Champion 2
Hoarding the Code 5
Unusually High Churn 8
Bullseye Commits 11
Heroing 13
Over Helping 15
Clean As You Go 18
In the Zone 20
Bit Twiddling 22
The Busy Body 24

PART 2

Scope Creep 27
Flaky Product Ownership 29
Expanding Refactor 31
Just One More Thing 33
Rubber Stamping 35
Knowledge Silos 37
Self-Merging PRs 40
Long-Running PRs 42
A High Bus Factor 44
Sprint Retrospectives 46

Introduction

At GitPrime we believe 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 health and 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.

20 Patterns is a collection of work patterns we've observed in working with hundreds of software teams. Our hope is that you'll use this field guide to get a better feel for how the team works, and to recognize achievement, spot bottlenecks, and debug your development process with data.

PART 1

Work patterns exhibited on an individual level

PATTERN 01

Domain Champion

1. Domain Champion

The Domain Champion is an expert in a particular area of the codebase. They know nearly everything there is to know about their domain: every class, every method, every algorithm and pattern.

In truth, they probably wrote most of it, and in some cases rewrote the same sections of code multiple times.

The Domain Champion isn't just "the engineer who knows credit card processing"; it's all they ever work on. It's their whole day, every day.

Some degree of job specialization is essential and often motivating. But even within specialized roles there can be 'too much of one thing.' Managers must balance enabling a team member to unilaterally own the expertise, and encouraging breadth of experience.

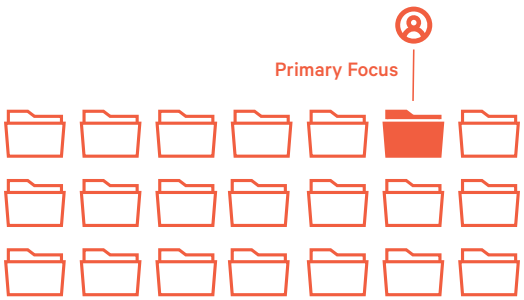
How to recognize it

Domain Champions will always work in the same area of code. They'll also rewrite their code over and over, and you'll see it in churn and legacy refactoring metrics as they perfect it.

Domain Champions are deeply familiar with one particular domain. As a result, they'll typically submit their work in small, frequent commits and will show a sustained above average *Impact*.

Because no one else knows more than the Domain Champion, there's usually very little actionable feedback that others can provide in the review process. As a result, Domain Champions will typically show low *Receptiveness* in incorporating feedback from reviews.

Domain Champions will seldom, if ever, appear blocked. Short-term, it's a highly productive pattern. But it's often not sustainable and can lead to stagnation, which of course can lead to attrition.



What to do

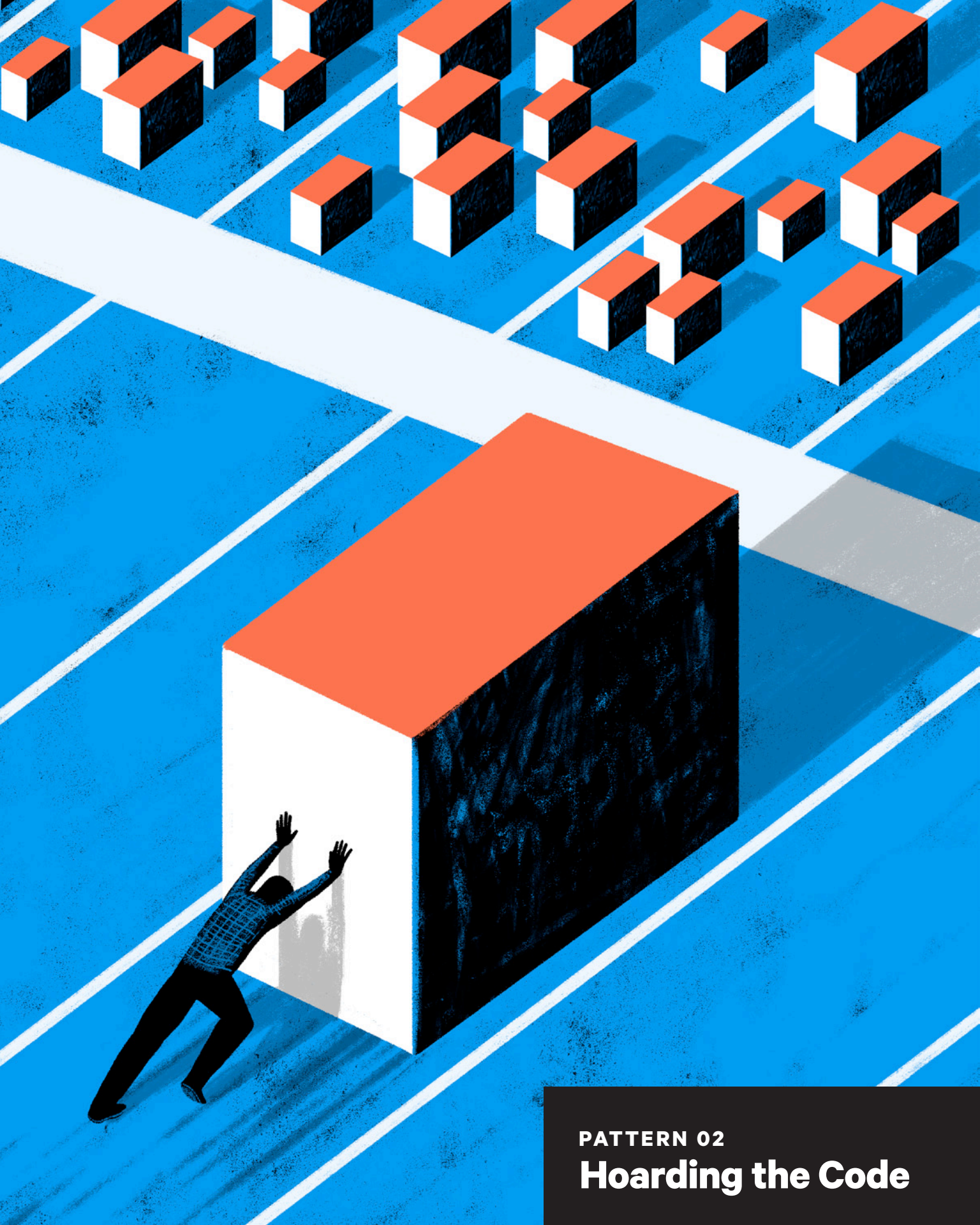
Assign tickets that focus on other areas of the codebase.

Of course, some engineers would prefer to stay where they are. It can be very enjoyable to do a task you're good at. And, it can be uncomfortable to take on work that requires information or skills you have less practice with. But effective managers will strive to challenge their team.

Start a new conversation in your next one-on-one:

1. Acknowledge their expertise and encourage them to share that expertise with others. Ask them who, if anyone, would benefit from participating in code reviews in the domain to learn best practices.
2. Ask them what they like to work on — first generally, then specifically.
3. Ask them if they are willing to take on a small assignment outside their domain in part to help share the best practices they've developed refining the code in their domain.

Inch the engineer out of their domain using small, low-risk tickets. A little bit of diversification can go a long way toward minimizing attrition risk and maximizing best practices.



PATTERN 02

Hoarding the Code

2. Hoarding the Code

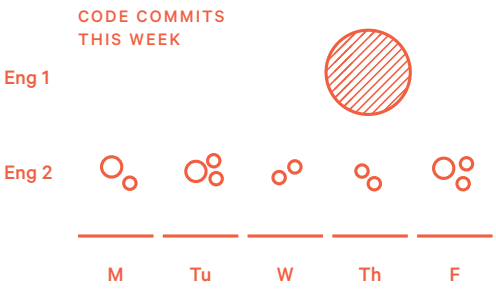
This pattern refers to the work behavior of repeatedly working privately and hoarding all work in progress to deliver one giant pull request at the end of the sprint.

It's not uncommon for programmers to wait until their work is done to share it. In creative and research-intensive fields, it can be a natural tendency to avoid sharing work when it's only just started. There are plenty of reasons why this might be: a fear of having others judge the work in progress, a fear of others taking ideas, or a desire to make the whole process seem effortless, to name a few.

Whatever the reason, the heart of the problem is this: the more an individual saves up their work, the less they collaborate with others. Working alone is inherently riskier than working with others. And Software Engineering is a team sport.

This tendency to work privately and then submit work all at once doesn't just limit and slow down the individual — it's damaging to the team's progress as a whole. Submitting work all at once means that there weren't any opportunities for collaboration along the way. Even more, once the work was submitted, someone else had to *review* all of that work. So naturally, this work behavior can also lead to lower quality code — both from the Submitter's standpoint (who didn't check in their work early to get feedback or notice potential missteps), and the Reviewer's perspective (who likely doesn't have enough time or energy to adequately review all of that code).

When you see large and infrequent commits, first consider the pattern's **duration** (have we seen this pattern for years, or has it only recently been visible?). This information can help determine whether this is the engineer's preferred way of working, or if this is caused by something outside the normal development process.



How to recognize it

Large and infrequent commits can be a sign that the engineer is working privately until their project is finished, and then submitting their work all at once.

This pattern is typically first seen in the *Work Log* report but is also identifiable in the team's *Review Workflow*. These PRs are larger and usually come later in a sprint or project. Because of this, they'll typically either take a longer time to review (relative to the team's average) or will get a lower level of review (see *Review Coverage*).

It's also common for these engineers to show lower than average *Receptiveness* in the *Submit Fundamentals*. When people work in isolation, only submitting it for review once they've decided it's the 'right' solution later in the sprint, it's generally much more difficult to take feedback on that work objectively.

What to do

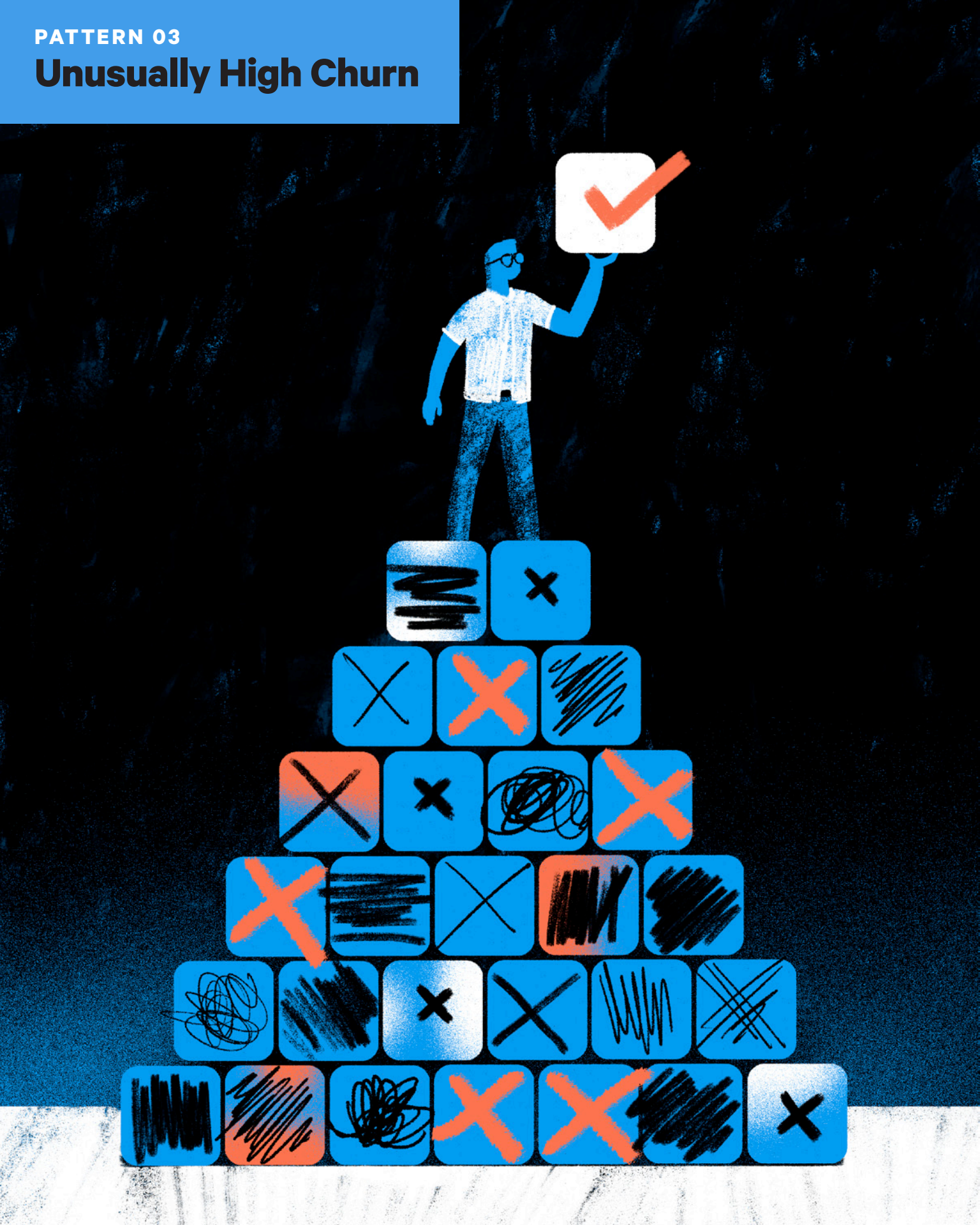
Above all else, be compassionate. Odds are, you've recognized this pattern right before or just after the end of a sprint, so these engineers are likely tired, stressed, and worn out. Make sure they get the time and space they need to recover from delivering such a big payload.

This can be great timing for an impromptu and informal 1:1. Going on a walk or getting coffee, for example, can keep the conversation casual. Get them talking about their latest project, ask what went well and what didn't, and recognize their achievement.

Along the way, bring up the topic of team collaboration, and how saving work until it's completed leaves little room for learning from others throughout the process. When teams do work together throughout a project, they can learn from each other's perspectives, reduce uncertainty and move faster, and even find improved solutions to the problem. In practice, that might look like submitting work far before the engineer thinks it's ready for a review.

PATTERN 03

Unusually High Churn



3. Unusually High Churn

Churn is a natural and healthy part of the development process and varies from project to project. However, Unusually High Churn is often an early indicator that a team or a person may be struggling with an assignment.

In benchmarking the code contribution patterns of over 85,000 software engineers, GitPrime's data science team identified that Code Churn levels frequently run between 13-30% of all code committed (i.e., 70-87% Efficiency), while a typical team can expect to operate in the neighborhood of 25% Code Churn (75% Efficiency).

Testing, reworking, and exploring various solutions is expected, and these levels will vary between people, types of projects, and stage in the software lifecycle. Given the variance, becoming familiar with your team's 'normal' levels is necessary to identify when something is off.

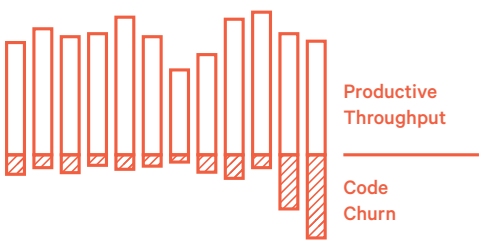
Unusually high churn levels aren't a problem in themselves. More likely, there are outside factors causing the problem.

An unusually high level of churn can be indicative of one of three behaviors:

- **Perfectionism**: When an engineers' standards of "good enough" are not aligned with the company's standard of "good enough." Engineers keep going back into the code to rewrite it because they think it can and should be better but may not add much to the actual functionality of the code.
- **They're struggling with the problem at hand.** This situation manifests differently than with Hoarding the Code (Pattern #2), because in this case, the engineer initially thought they had correctly solved the problem, perhaps even sent it off for review, and then discovered it needed to be rewritten. Not just touched up. Rewritten.
- Or, most commonly, **issues concerning external stakeholders.** We see this with unclear or ambiguous specs, late arriving requirements, or mid-sprint updates to the deliverables.

How to recognize it

This pattern is characterized by **high levels of churn in the back of the sprint** or project. Watch for churn rates that climb significantly above the engineer's historical average (see the *Snapshot* and *Spot Check* reports), pairing that information with where they are in a project.



What to do

Churn is normal in lots of situations. Redesigns, prototypes, and POC's are all examples where you would expect to rewrite large chunks of code. So when you notice *unusually* high churn, take into consideration whether **this is routine** or something's off. If it's the latter:

Determine whether **an external stakeholder is driving the situation.** If so (and the engineer has verified that this is causing the higher levels of churn), then:

1. **Show the data**. Show how late arriving specs or last-minute changes are throwing the project off.
2. Pull the ticket from the sprint, or decide on an MVP and split off the additions into a refinement sprint.

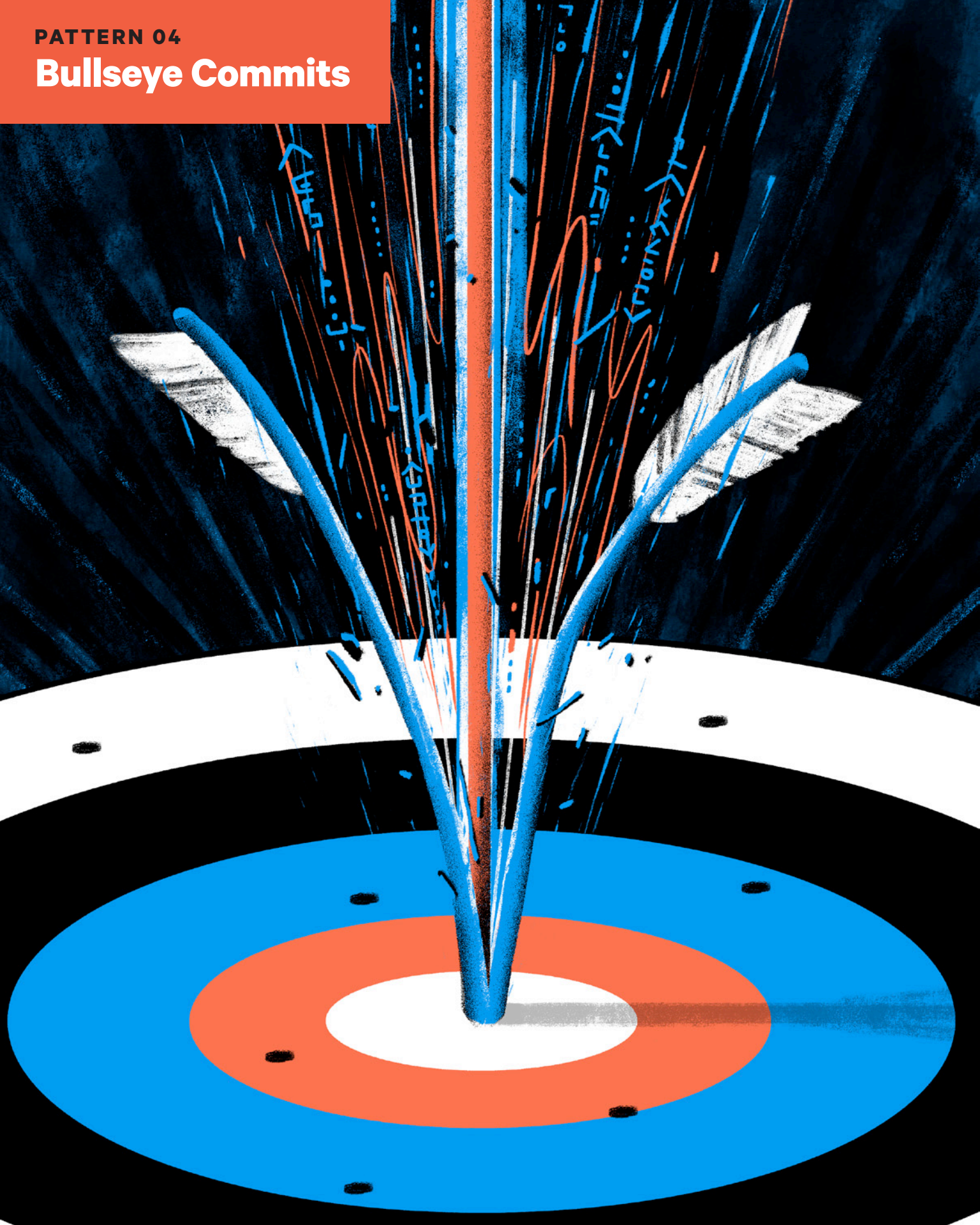
If an external stakeholder is *not* driving the Unusually High Churn, call in the cavalry!

It is usually preferable to be coached by a fellow engineer or team lead instead of a manager.

1. Ask for a pre-submit code review or a rubber duck.
2. Ask to split the work. The act of dividing the work often reveals the root issue.
3. Ask a more senior engineer to assess what "good enough" is in the context of the project.
4. If the problem is difficult, or if the domain is unfamiliar, bring in another engineer to pair program.

PATTERN 04

Bullseye Commits



4. Bullseye Commits

This pattern is relatively common in most teams, but it often goes unrecognized: an engineer understands a problem, breaks down the project into smaller tasks, and submits code that has little room for improvement.

Most likely, not all of the commits that make up the project will be Bullseyes. But the ones that are, generally have a small to modest impact and were thoroughly reviewed and approved on the first try. Celebrate them!

How to recognize it

In practice, Bullseye Commits can be identified by when they were submitted in regard to the deadline, their impact, and how they were treated in the review process. Generally, the code was started and completed in advance of the deadline, with negligible churn. The commit's *Impact* was small to modest in size and was then thoroughly reviewed. It was approved on the first try (see *Review Workflow*).

It's the level of thoroughness in the review that distinguishes Bullseye Commits from Rubber Stamping (Pattern #15). In Bullseye Commits, code reviews are substantive.

What to do

Recognize a clean bullseye in a stand-up, or a simple note: "I saw that check-in, nice job!" Whether it's public or private, showing that you noticed and that you care will only reinforce this pattern.

If there's an engineer who regularly makes Bullseye Commits, it may be helpful for others to understand how they approach projects. Ask the engineer to do a lunch and learn, or consider asking them to provide feedback on another engineer's work in the review process.

PATTERN 05

Heroing

5. Heroing

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

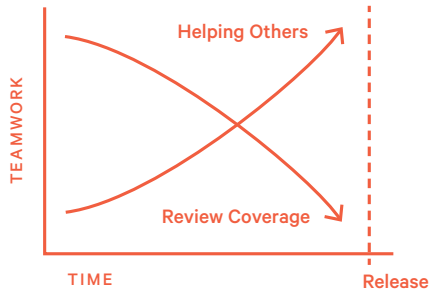
Granted, a good save is usually better than no save. But regular Heroing leads to the creation of unhealthy dynamics within the team or otherwise encourages undisciplined programming. Some team members even learn to *expect* them to jump in on every release.

Heroing can be a symptom of poor delegation or micro-management. It also points to trust issues on a number of levels. The Hero 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, the Hero 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 GitPrime'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. It'll help to get the Hero'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 Hero — to give and be receptive to feedback in reviews. Coach the Hero to turn their 'fixes' into actionable feedback for their teammates.

PATTERN 06

Over Helping

6. 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. 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 it

You'll notice this pattern in the same way you'd realize "Heroing" (Pattern #5) in GitPrime'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 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 (see *Knowledge Sharing*).
- 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 this 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.

PATTERN 07

Clean As You Go

7. Clean As You Go

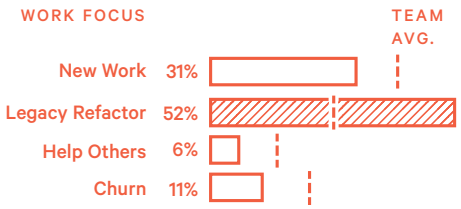
A codebase is continuously evolving by nature, but it doesn't evolve evenly across all aspects. A Clean As You Go engineer will notice and refine shortcomings even if it's not essential to the task at hand.

This pattern of continually improving the code adjacent to the code the engineer is working on, is a fantastic pattern to encourage.

"Fixing" work certainly doesn't get the attention that "feature" work does, in part because there's rarely that *ta-da* moment. While the activity of regularly fixing existing code while working on other tasks can be much less visible and recognizable than working on new code, this engineer's contribution is invaluable.

How to recognize it

"Clean As You Go" refers to when an engineer contributes new code and also mends adjacent code in the codebase. Consequently, you'll notice these engineers writing new code while also showing higher levels of legacy refactoring, that together usually exceed the expected scope of change for the assignment at hand.



What to do

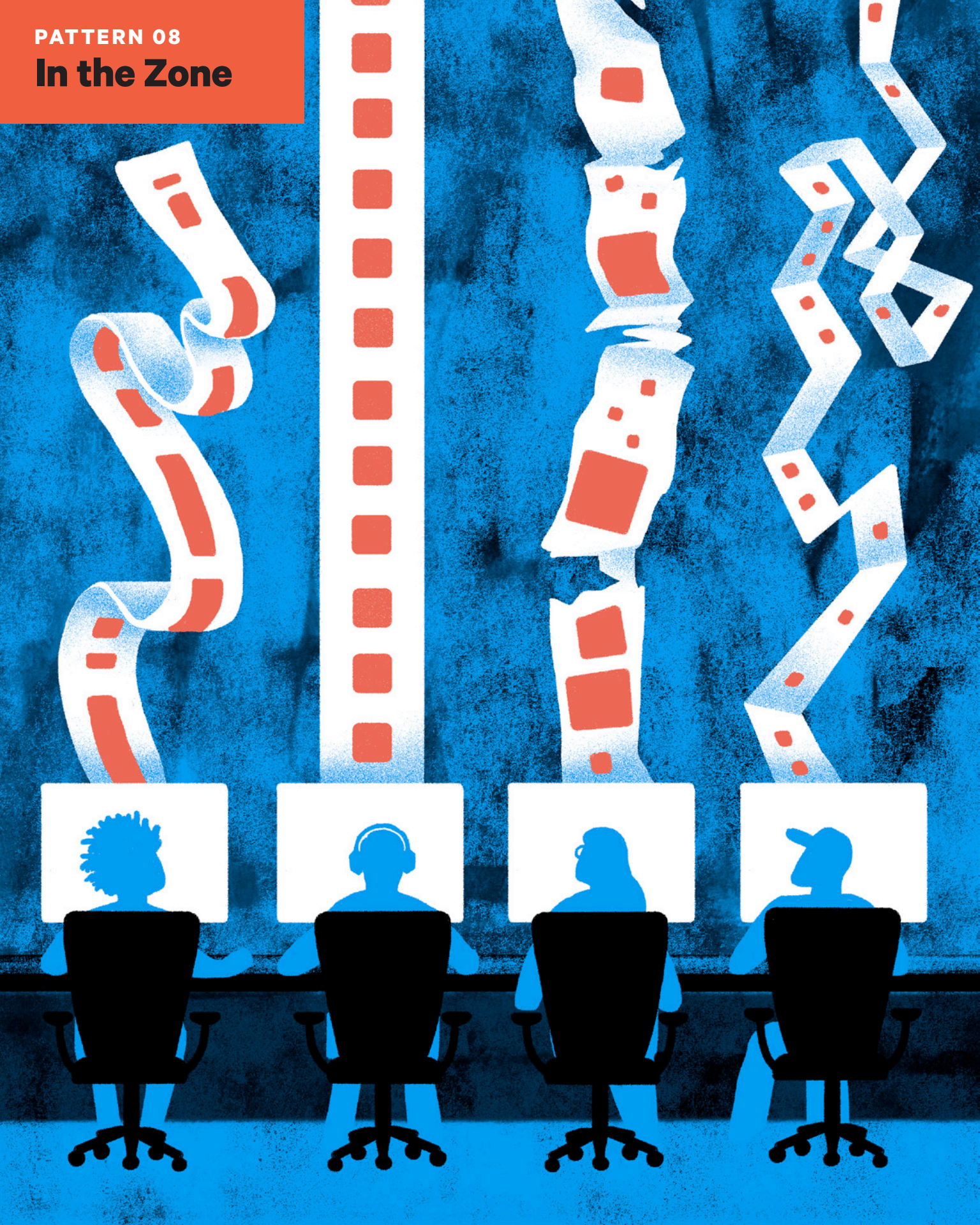
Recognize this engineer's work publicly and use it as a model for other team members to work towards. Regularly acknowledge it in sprint retrospectives and standups, even after you first observe the pattern. Consistent acknowledgment lets everyone know you value this effort.

Encourage this engineer to formalize their work pattern with documented coding standards (e.g., naming conventions, getter/setters, preferred patterns for ORM work, etc.)

A little bit of encouragement can go a long way to minimizing tech debt, and the engineers who Clean As They Go can help demonstrate how these best practices look.

PATTERN 08

In the Zone



8. In the Zone

This pattern is exhibited by engineers whose work is, in a word, consistent. They have a knack for getting in the zone and shipping high-quality work week in and week out. Their work is reliable and predictable in nearly every way.

Professional software development is an endurance sport. To create lasting enterprise value, you must show up every day and produce quality work. Real value creation can take years.

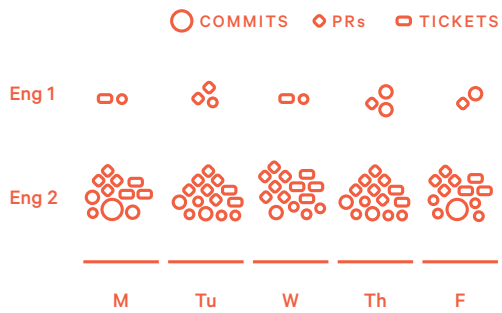
It's tempting to think of this as a person and not a pattern. However it's useful to depersonalize this engineer's work as a pattern — it's easier for others to model discrete behaviors than it is to model a person.

How to recognize it

An engineer in the zone organizes their day to eliminate distraction and focus on delivering business value. Their *Active Days* are consistently above average. Their *Impact* is high and consistent. Their PRs are timely, evenly paced, and nicely sized. They consistently participate in reviews, so their *Involvement* is high and consistent. Their churn is usually lower than average.

What to do

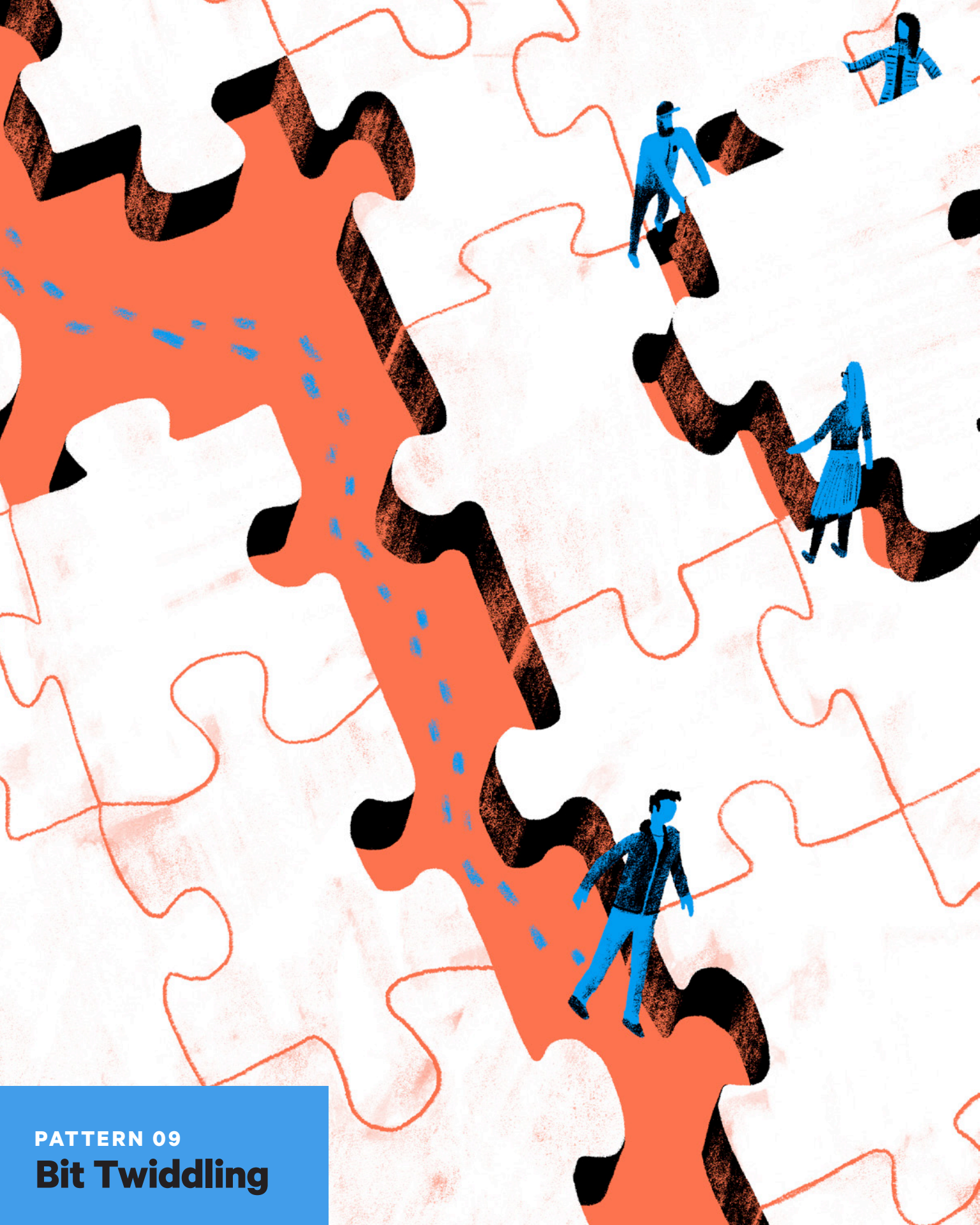
Similar to engineers who exhibit the "Clean As You Go" pattern, it helps to acknowledge this pattern either publicly, privately, or both. Emphasize their consistency and how great code is built *not* in a single sprint or pulling all-nighters. The *Work Log* and *Review and Collaboration report*s will show this pattern over time, and they can be used to support the story (e.g., "six weeks of committing code every day is something to admire").



If increasing overall team velocity is important to you, helping everyone on your team find their zone is a foundational place to start.

An essay from Paul Graham, titled "Maker's Schedule, Manager's Schedule" offers context and strategies for blocking meetings and creating space to get in the zone.

Small changes in scheduling and reduction of interruptions can amount to significant increase in capacity. Furthermore, consistently getting in the zone allows your team to ship at a sustainable pace without suffering from the burnout of heroic sprints.



PATTERN 09

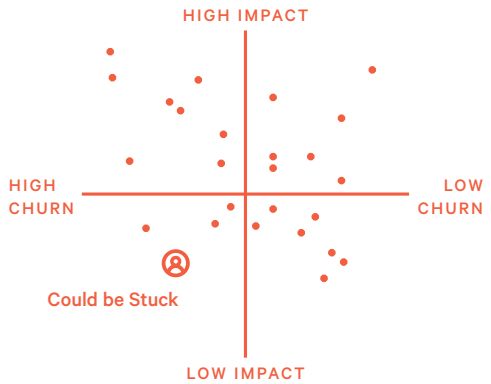
Bit Twiddling

9. Bit Twiddling

Bit Twiddling is like working on jigsaw puzzle to the point where everything looks the same and you're not making progress anymore. You might pick up the same piece, try it in a few places, rotate it, put down, only to pick it up a few minutes later.

Bit Twiddling reveals itself when an engineer is unwaveringly focused on a single area of the codebase for a very long time, making only slight changes here and there. This often happens because the engineer doesn't fully understand the problem or the context for making the change.

They may be losing steam and motivation, or are at high risk for doing so.



How to recognize it

Look for high rates of churn in the same area of the code. The key is to couple repetition and refactoring with ambivalence or indifference in code review over an extended period.

For example, watch for a standard library call, or otherwise stable code, get refactored into customized code for some difficult to articulate optimization. Or, watch for code that gets refined and refactored multiple times with disinterest — light code review and PRs with generic submitter comments like "refactoring," "reorganizing," or "touch up," followed by "LGTM".

What to do

Look for ways to reenergize the engineer with a new project. Find a ticket, even a small one, that will lead into new and interesting areas of the code — even if it comes at the expense of the team's productivity in the short-term.

Creative workers thrive when tackling new and challenging problems, even if they at first balk at working outside their area of expertise. New experience typically leads to learning something new, a process most engineers enjoy.

PATTERN 10

The Busy Body

10. The Busy Body

The Busy Body is an engineer who skips all over the codebase: they'll fix a front-end problem here, jump to some refactoring, then fiddle with the database over there.

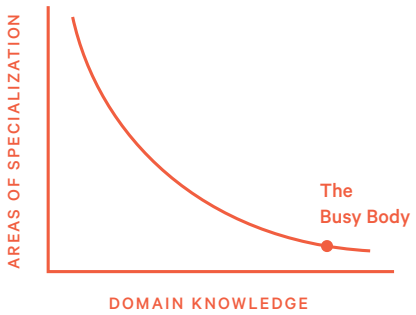
Their work is always lightweight and shies away from heavier problems. This behavior can be perfectly normal over short periods or in isolated instances. And, in fact, some shifting around is healthy.

But the Busy Body is problematic over a long period because these engineers end up without a strong sense of ownership. There's nothing for them to point at and say, "I made that." Even if they can solve a wide range of problems, lacking something that they own can lead to attrition.

How to recognize it

Engineers exhibiting this pattern will show high levels of *Impact* and lots of small pull requests without any identifiable home base in the code. They'll show a high level of *Involvement* in the review process. And because they typically spend their time building and spend less time bug fixing their own work, they'll show high levels of new work and relatively low churn.

These dynamics are often first identified in the *Player Card* report or in the team's *Submit and Review Fundamentals*.



What to do

Give these engineers something to own top to bottom. Whether it's a module, a new feature, or a large project, ask them to do more than just 'get it done'. Ask them to become an expert in that particular area or on that specific project.

Then, double down on their strengths in that area: assign them the 1.1 version, the bug fixes, the unit tests, and the documentation, then give them the 1.2 and 1.3 versions as well. Allow them the opportunity to get to know their domain, to work with it, to teach others about it, and to develop a mastery. Ask them to give a presentation on the project to highlight lessons learned and best practices. The key is to nurture a true sense of ownership.

PART 2

Work patterns exhibited on a team-wide level

PATTERN 11

Scope Creep

11. Scope Creep

Intuitively, we all know what Scope Creep is — along with its associated risks. Still, there are plenty of different definitions for the issue so here's what we're focusing on:

Scope Creep (noun): a pattern whereby the originally agreed upon scope increases or changes **after implementation has begun**. Often, though not always, Scope Creep happens incrementally and thus invisibly.

Using data to make Scope Creep visible to all parties can help mitigate the risks of unexpected work, and can also be used to combat this pattern moving forward.

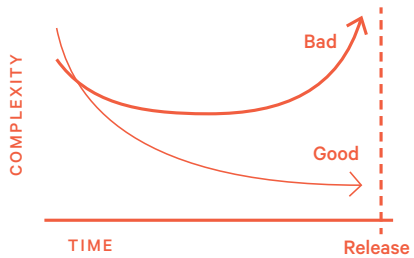
Even in the most well-defined projects, out-of-scope tasks arise. As a manager, you need to watch for runaway situations where engineers are being asked to shoulder an unreasonable increase in scope.

How to recognize it

Scope Creep is characterized by **a sharp uptick in progress toward the back of a sprint that wasn't driven by code review**.

Generally, the problem a team is solving should be getting smaller over time as features are completed. So a sudden spike in activity, particularly in the later stages of a project, tends to be a signal that something new came in.

When you see this occurrence become a pattern sprint over sprint with the same team, look to the external stakeholders that interface with that team to see what might be causing the issues.



What to do

Use GitPrime data to show the additional work caused by the scope creep. Scope creep is caused by poor planning and insufficient attention during design. It's not the engineer's responsibility to shoulder the work resulting from bad specs. Call it out! Let the people who are responsible for pushing a poorly designed project into implementation know that it's simply not ok.

Then show them how much additional work their carelessness caused. Show them the data. This more than anything will make the true consequences of scope creep visible and thus actionable.

PATTERN 12

Flaky Product Ownership

12. Flaky Product Ownership

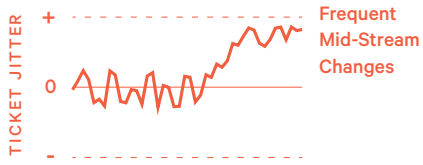
Miscommunications between Product and Engineering can easily lead to Scope Creep. Flaky Product Ownership, however, can show up slightly different in the data and also generally requires a different approach.

There are two important behaviors that fall under this category:

- A Product Owner submits incomplete requirements, leading to extra engineering time spent toward filling in the gaps or resulting in 'miscommunications' later on.
- A Product Owner changes their requests after implementation began, leading to missed deadlines.

How to recognize it

This pattern tends to reveal itself in **recurring scope creep driven by the same product owner**. You may notice a significant expansion of code that wasn't driven by code review in the back of the sprint.



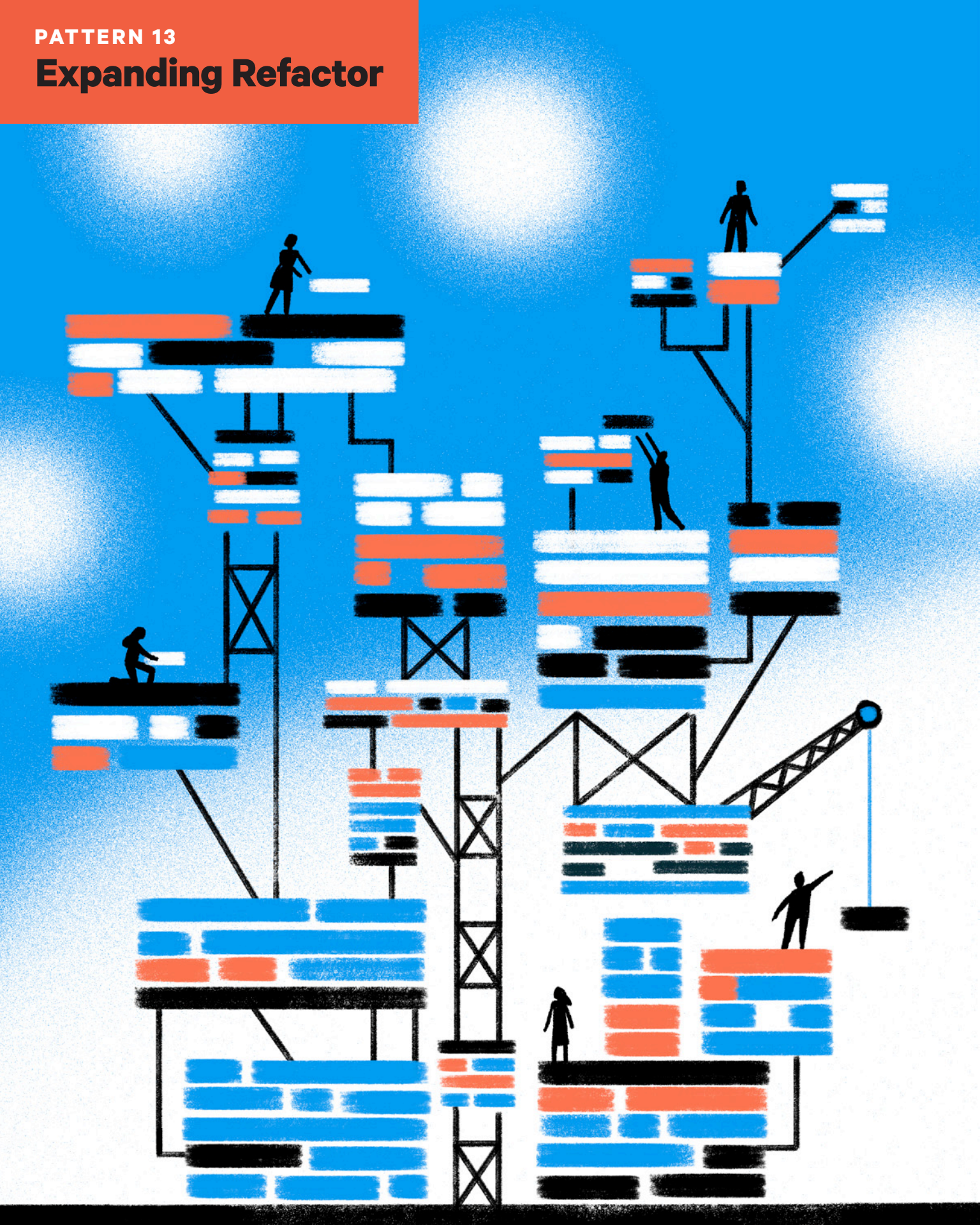
What to do

Ambiguous or changing requirements from a Product Owner can often be a sign that that person is stretched thin. They have too much to work on, so nothing gets their full attention. It's helpful, for that reason, to have a discussion with their manager. Bringing data to the discussion can eliminate skepticism around what's happening and help cut straight to the discussion about how to resolve the situation.

The handling of the situation should generally be left to the Product Owner's manager. If it's an issue of too much work, it can help to eliminate the individual's work in progress. Otherwise, it may simply require coaching around any areas they tend to overlook when creating specs.

PATTERN 13

Expanding Refactor



13. Expanding Refactor

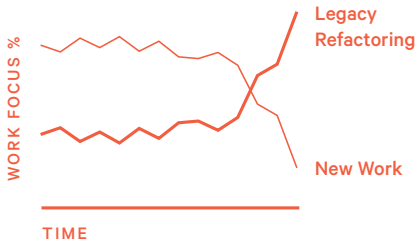
Expanding refactors happen when a planned effort to improve or revise a section of code triggers a dramatic widening of scope.

What was intended as an optimization exercise, becomes a wholesale rewrite.

How to recognize it

A small amount of legacy refactoring is healthy. It's when you notice a whole slew of changes in areas that are unrelated to the feature at hand.

Look at the *Work Log* for outsized code commits in sets of files that seem completely unrelated to the feature at hand. Talk to the engineer, expanding refactors are rarely driven by the product teams.



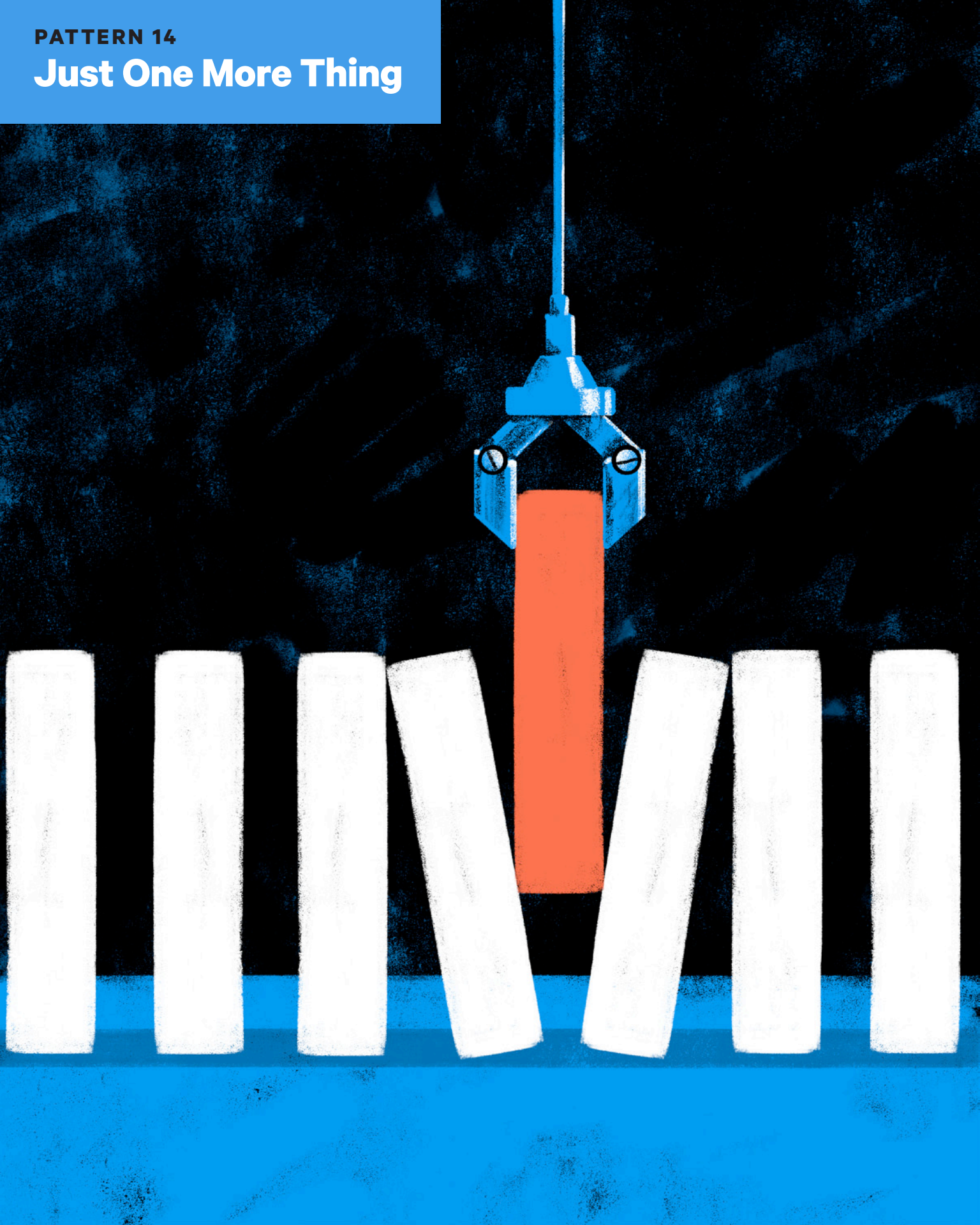
What to do

Open the topic up for discussion with the team. Ask team members to make a case for and against the refactor, and then come to a conclusion about whether it's best to move forward with the project, drop it, or tackle it with a different approach.

It can also be useful to provide standards around what success is — what "done" looks like. That way, everyone's clear around what the project is and isn't, and so the expanding refactor doesn't consume too much of your team's time and energy.

PATTERN 14

Just One More Thing

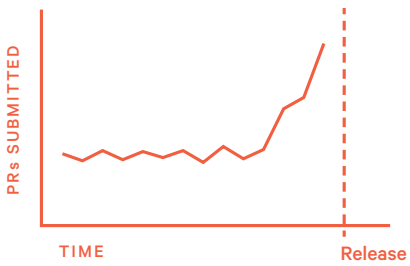


14. Just One More Thing

"Just One More Thing" refers to the 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, 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.



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 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.

PATTERN 15

Rubber Stamping

15. Rubber Stamping

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 it

Rubber Stamping is most noticeable in the *Review and Collaboration* reports. Watch the *Review Workflow* for PRs that opened and closed in a preposterously short period of time, with a very low level of *Receptiveness*. Low levels of engagement in reviews can also be seen in the *Involvement* and *Review Coverage* metrics.

When review happens later, as opposed to right after the PR is opened, there will be little to no back-and-forth in the comments (see the *PR Resolution* report). If there were no comments on the PR, this PR will show as Unreviewed.

The *Team Collaboration* metrics will also provide insight into the time and energy that team members are allocating toward the review process over any given time period. These reports will help watch and manage the trends in the long-term.

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.

PATTERN 16

Knowledge Silos

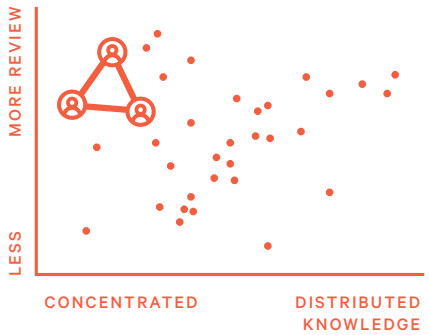
16. Knowledge Silos

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 *Team Collaboration* metrics, 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 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 only review each other's code, the team's *Knowledge Sharing Index* will trend toward 0. If the majority of the team reviews each other's 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 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. 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.

PATTERN 17

Self-Merging PRs

17. Self-Merging PRs

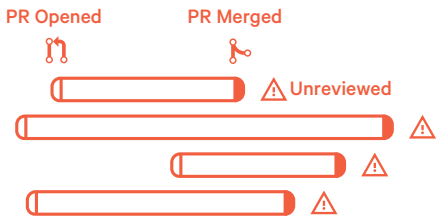
This pattern refers to when an engineer opens a pull request and then approves it themselves. This means no one else reviewed the work and it's headed to production!

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 it

Self-merging is easy to see because the submitter and the reviewer are the same people. In GitPrime, these instances will show up in the team's *Unreviewed PRs* metric as well as in their *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.

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.

PATTERN 18

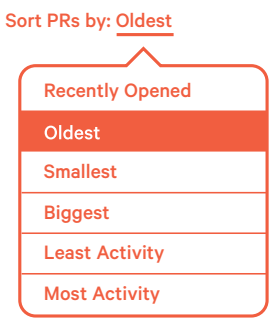
Long-Running PRs

18. 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 it

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 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'.

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 (see *Responsiveness*), 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.

PATTERN 19

A High Bus Factor

19. 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.

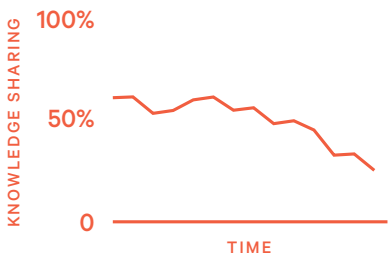
Having a low bus factor is risky. A High Bus Factor means that there is a greater distribution of knowledge and know-how about the code across the team. When more than one engineer knows about each area of the team's code, there's more optionality for managers to assign tasks and more people that can 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 it

A team's distribution of knowledge can be visualized with the *Knowledge Sharing Index*. 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. This also 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.



What to do

Knowledge Distribution 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* (i.e., a low bus factor, higher risk), see the *Review Radar* for opportunities to get team members more involved in the review process. When you see the behavior you want to see in the review process, consider recognizing that in a team-wide meeting.

PATTERN 20

Sprint Retrospectives

20. Sprint Retrospectives

Retrospectives are a common practice that offer an easy way to continuously improve: take time to reflect, as an individual or a team, on a project, action, or occurrence.

.While reflecting on the goals of the sprint, what actually happened, why it happened, and planning for what's next, use data to provide a more complete view on the team's progress. Instead of looking just at *what* was built, look at *how* it was built. Visualize the development process and watch for trends in work patterns across the team and at the individual level.

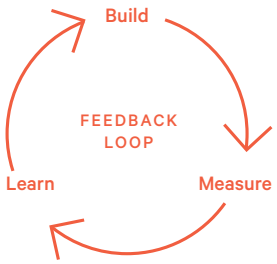
How to recognize it

A good Sprint Retrospective uses data to help people compare what they felt happened during the sprint and what actually happened in the sprint.

What to do

As a manager of managers, it helps to coach managers of individual contributors on the practice of including data in their retrospectives.

If there are specific work patterns you see in the team that you either want to see more of or want to manage away from, consider showing them what those behaviors look like in the data, how to watch for them, and what to do when they see them.



Encourage them to include data in discussions with you, and with others in the organization, and show them how to do so.

In short, Sprint Retrospectives are about watching for and managing work patterns. It's about recognizing achievement, spotting bottlenecks, and debugging the development process with data.

"A world without GitPrime is one without hard data and no real evidence of how things are going. As engineering teams get larger, it's more complicated; more repos, more code, more people. Without a tool like GitPrime that pulls it all together, it's almost impossible to get a handle of what's going on."

— Adam Abrevaya, VP Engineering, CloudHealth by VMware

"In Engineering we're good at measuring the what. GitPrime gives us is the ability to measure how we get there. I can't imagine a company like ours operating without GitPrime metrics."

— Mathew Spolin, VP Engineering, AppDirect

About GitPrime

GitPrime is the world's leading Engineering Logistics Platform. We help organizations of any size use data to debug their development processes, so engineers can spend less time waiting on others and more time working on what matters.

Today more than 400 enterprises, including PayPal, Atlassian, and Adobe, rely on GitPrime to keep a pulse on the health and productivity of their teams.

GitPrime aggregates data from git repos, ticketing systems, and pull requests and transforms them into easy to understand insights and reports. With visibility into the software development process, we help you map initiatives to outcomes, adjust processes with confidence, and advocate for your team with facts instead of feelings.

Visit us at gitprime.com to start a 15-day trial or chat with someone on our team about how GitPrime can help.