HAYSTACK

Git Analytics for Engineering Leaders.

Book of Engineering Management

Top questions,
how to find the answers,
and things JIRA won’t tell you.

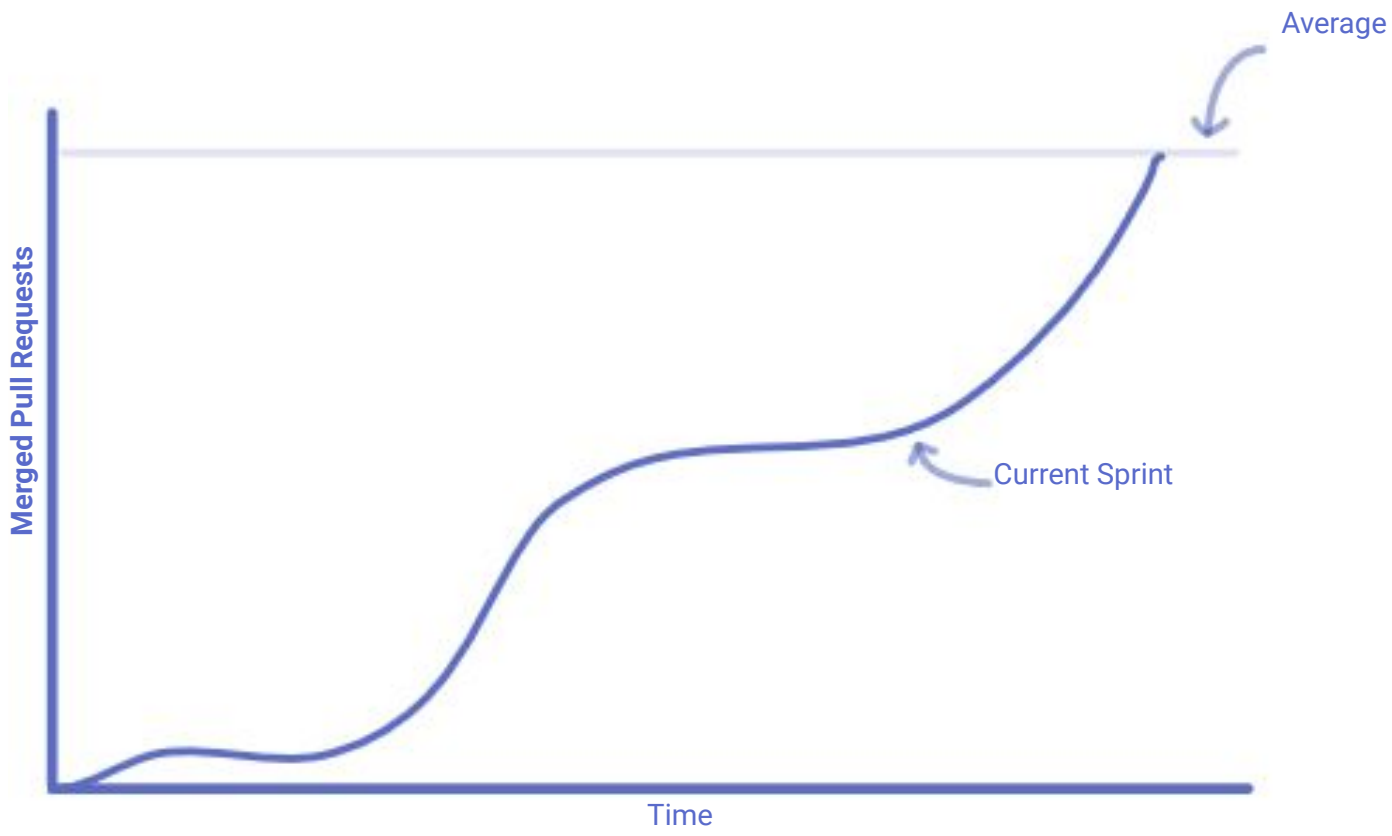
Hint: They’re already in git.

Table of Contents

Are we on track to deliver?

How to spot blockers,
visualize risk,
and keep sprints on track.

Are we on track to deliver?



Merged Pull Requests This Sprint

The number of *Merged Pull Requests* helps visualize if you're on track to hit your goals.

What to look for

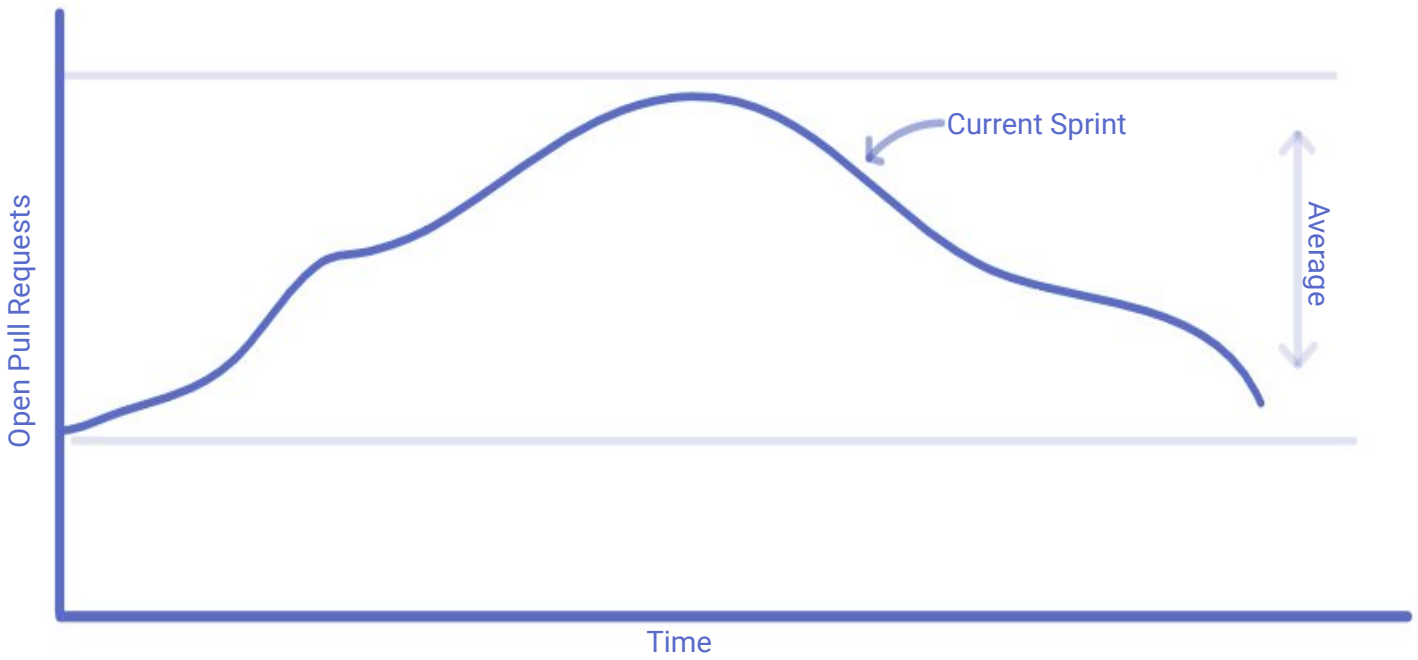
Look for these healthy patterns when checking your sprint status:

1. Steady rise in *Merged Pull Requests*
2. Increasing towards the team average

A constant rise in *Merged Pull Requests* shows a healthy pattern of opening and closing manageable chunks of work. It typically shows that the team is moving at a consistent rate and not running into critical blockers.

Note the team's *Average Number of Merged Pull Requests* (horizontal line in the chart above). The horizontal line shows the team's typical output. Each sprint, you can see how the current sprint matches up to the team's historical performance to make sure you're on track.

Are we on track to deliver?



Open Pull Requests This Sprint

The number of *Open Pull Requests* helps visualize if you're on track to his your goals

What to look for

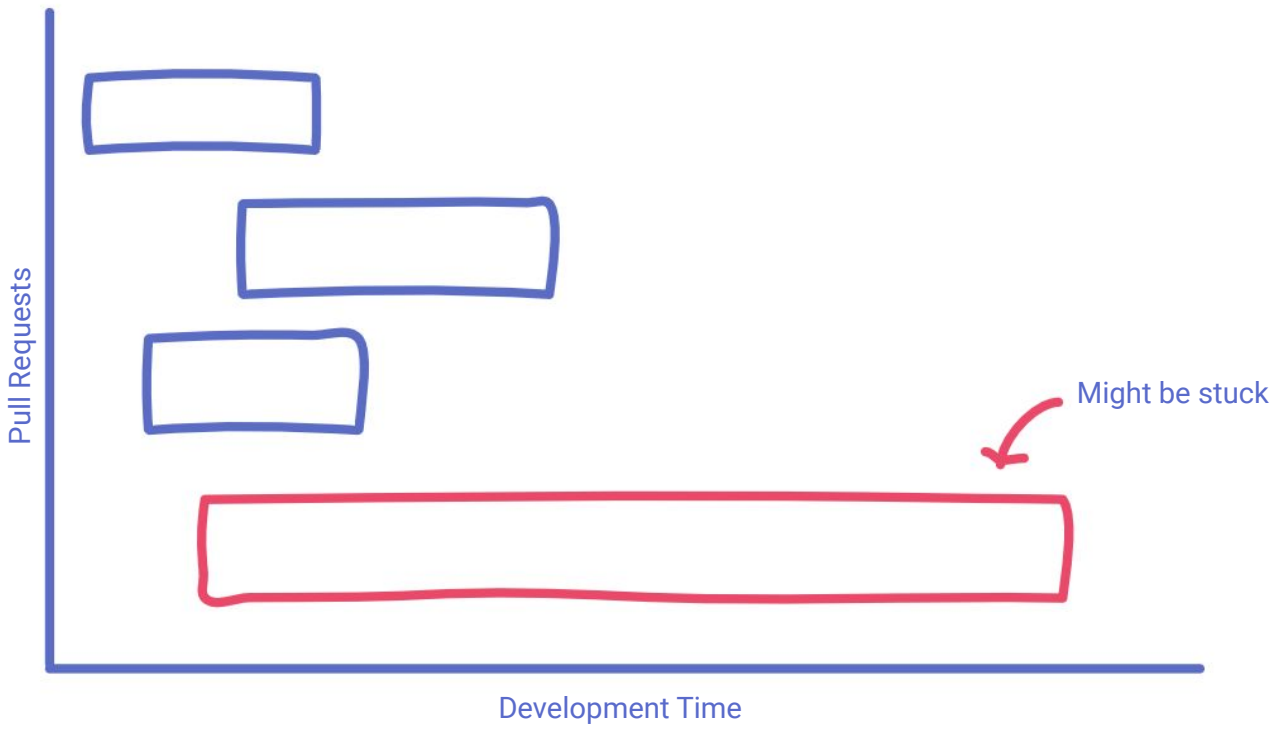
Look for these healthy patterns when checking your sprint status:

1. Oscillation in *Open Pull Requests*
2. Rise in *Open Pull Requests* near the middle of the sprint
2. *Open Pull Requests* staying within the min/max range

Oscillating *Open Pull Requests* signals a healthy pattern of opening and closing manageable chunks of work. A rise in *Open Pull Requests* near the middle of the sprint shows that your team is on pace to deliver.

Keeping *Open Pull Requests* within the team's average range (shown in the chart above) ensures that your team is taking on the appropriate amount of work.

Spotting Blockers



Long Time In Development

Visualizing *Time Spent in Development* helps quickly assess if your team is getting stuck.

What to look for

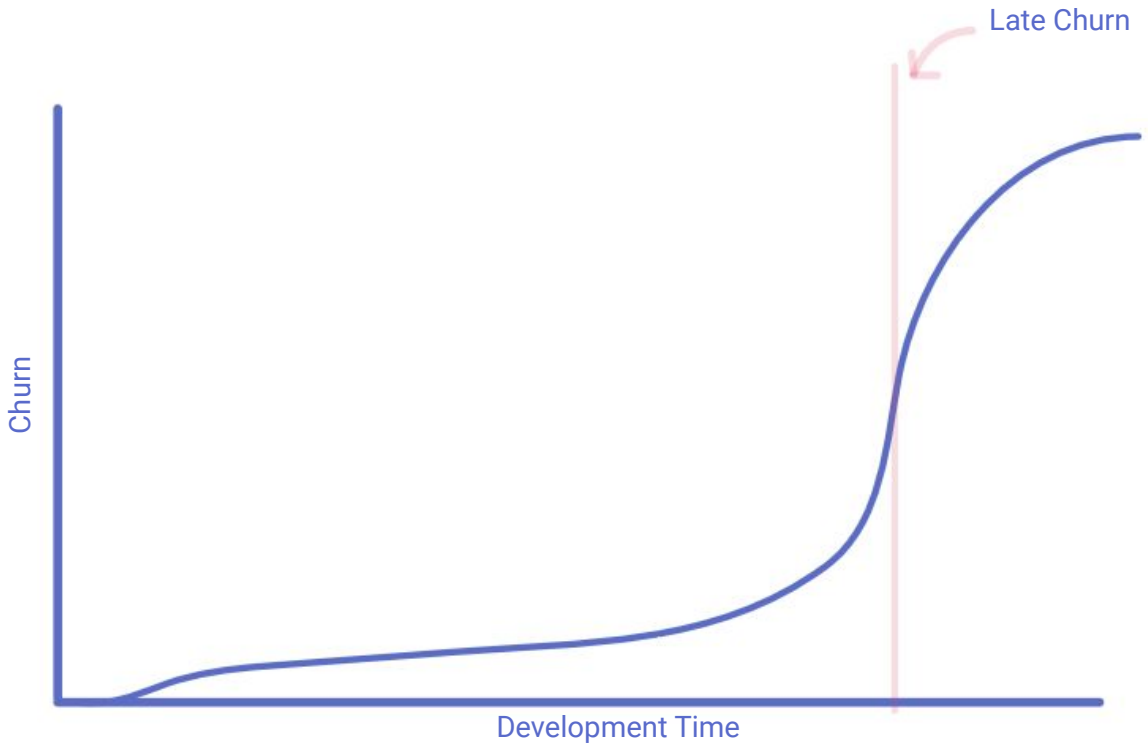
Look for these unhealthy patterns in your *Sprint Report* that can signal your team is getting stuck:

1. Unusually Long *Time Spent in Development*

Long *Time Spent in Development* can signal many things. Although it's not always a bad thing to spend a long time in development, we can use this as a way to surface some potential bottlenecks before they derail the sprint.

Long development time can signal complex work, a developer is stuck and may need assistance, scope creep, changes in priority or even poorly written tickets. It's important to follow up on long time in development to determine the root cause.

Spotting Blockers



Late Churn

Visualizing *Churn* helps quickly assess the type of work your team is doing.

What to look for

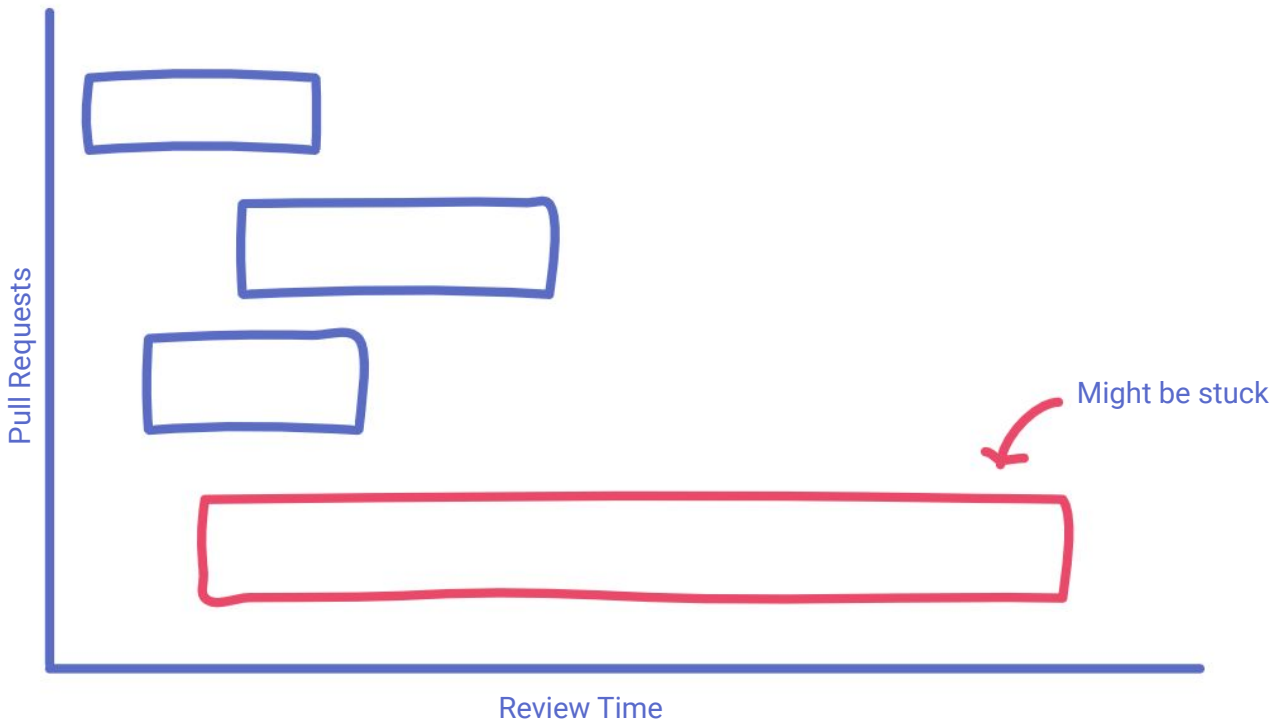
Look for these unhealthy patterns in your *Sprint Report* that can signal your team is getting stuck:

1. *Churn* appearing late in the sprint

Late *Churn* can signal many things. Sometimes late churn can be very healthy. Take the situation of an engineer quickly writing a proof of concept feature then proceeding to clean up and refactor his recent work. This is a great habit!

Having said that, late *Churn* can appear in a few more (undesirable) ways. Late churn can often be caused by changes in scope, poorly written tickets or even a developer being too much of a perfectionist. In each case, you'll want to watch out for late churn, follow-up, and make sure it doesn't derail your sprint.

Spotting Blockers



Long Time In Review

Visualizing *Time Spent in Review* helps quickly assess if your team is getting stuck.

What to look for

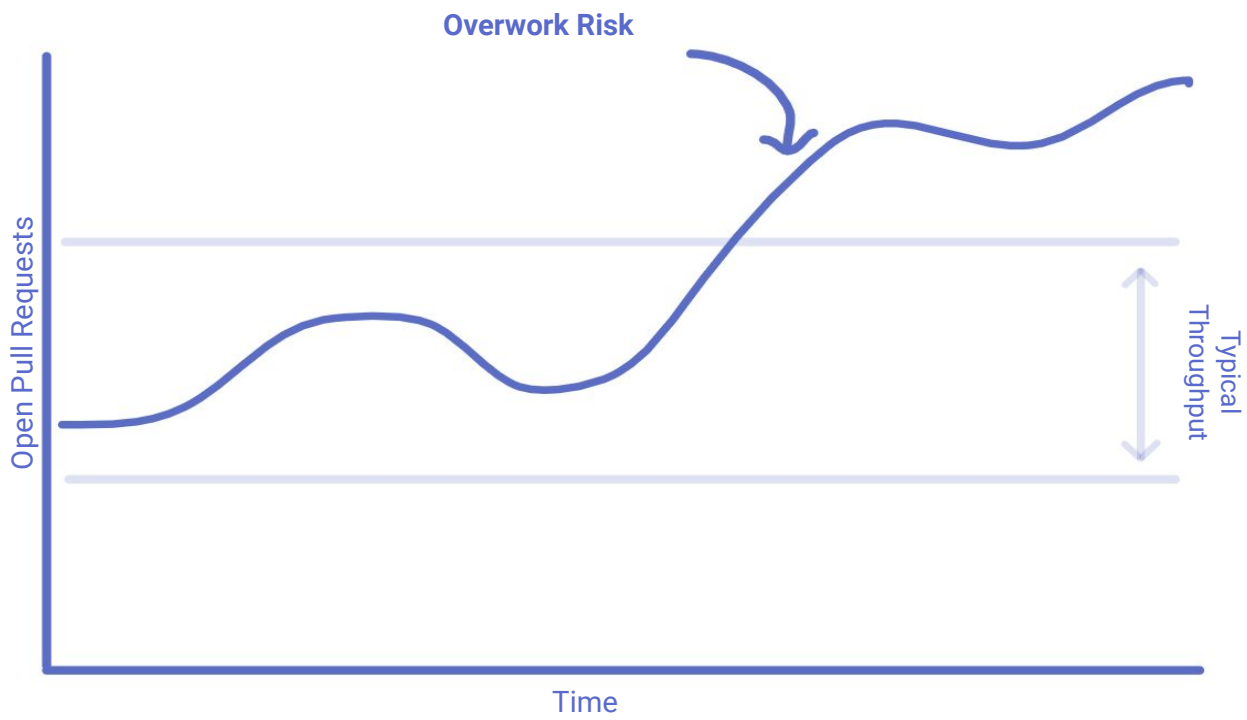
Look for these unhealthy patterns in your *Sprint Report* that can signal your team is getting stuck:

1. Unusually Long *Time Spent in Review*

Long *Time Spent in Review* can signal many things. Although code review is a healthy (and necessary) part of the development process we want to make sure to keep an eye on it.

Long review times can signal large/complex pull requests, idle time blocked by the reviewer and even changes in priority that cause features to get left idle and waiting in the pipeline. In any case, we should follow up and clear up any bottlenecks that might be there.

Overload Risk



Too Much Concurrent Work

Visualizing the number *Open Pull Requests* this sprint helps quickly assess if your team is doing too much.

What to look for

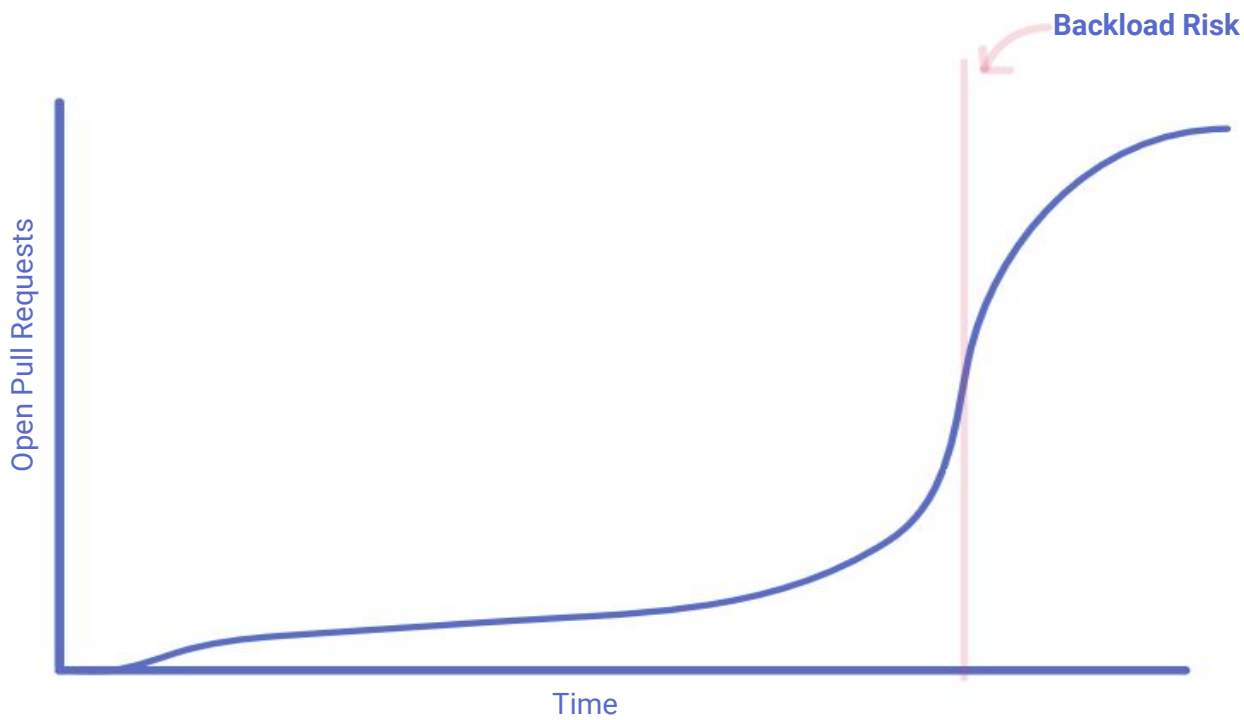
Look for these unhealthy patterns in your *Sprint Report* that can signal your team is getting stuck:

1. Unusually High Number of *Open Pull Requests*

It’s often the case that developers open pull requests, wait for review and move onto the next item in the sprint. This is great since no time is wasted but if left unchecked this situation can lead to overwork and pile ups of incomplete work that can derail your sprint.

Too many *Open Pull Requests* can signal taking on too much work, scope creep, change in priorities and unexpected issues/bugs coming into the sprint. It’s good to keep an eye out of the number of *Open Pull Requests* since it’s a great indicator of your team’s current workload.

Backload Risk



Late Pull Requests

Visualizing the number *Open Pull Requests* throughout the sprint helps quickly assess if your team is on track.

What to look for

Look for these unhealthy patterns in your *Sprint Report* that can signal your team is off track:

1. Large number of Pull Requests opened late in the sprint
2. Small number of *Open Pull Requests* throughout the sprint

It's often the case that developers aim to finish work by the end of the sprint. This can sometimes lead to opening pull requests too late, which can put your sprint at risk of missing deadlines.

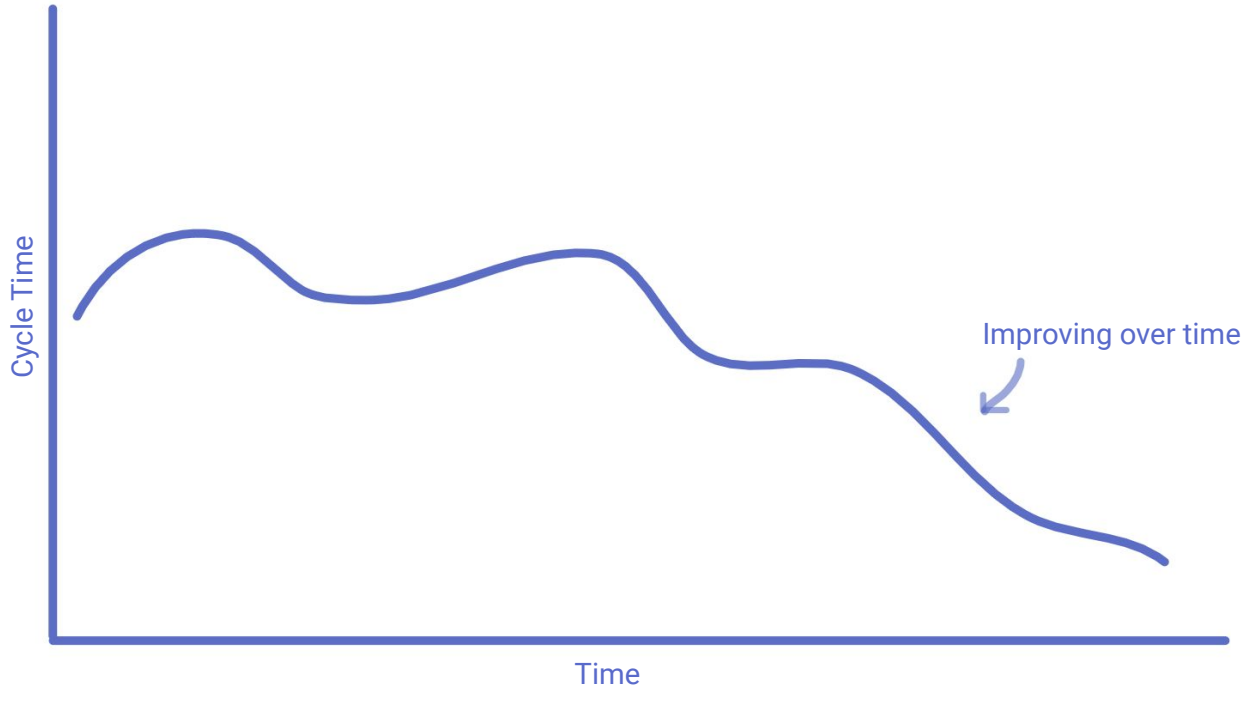
Late pull requests are not always a signal that you're at risk of but it is a good indicator to watch out for. Late pull requests can often run into snags in the review, QA, and deployment process that can cause the work to be delayed.

Also note the number of *Open Pull Requests* throughout the sprint. Be wary of stagnation as it's an early indicator that you may run into backload risk.

How fast do we deliver?

How to measure speed, understand throughput, and visualize trends

How fast do we deliver?



Cycle Time

Cycle Time can help visualize how quickly your team delivers and how that changes over time.

What to look for

Look for these healthy patterns in your *Output Report* that can signal your team is consistent or improving over time:

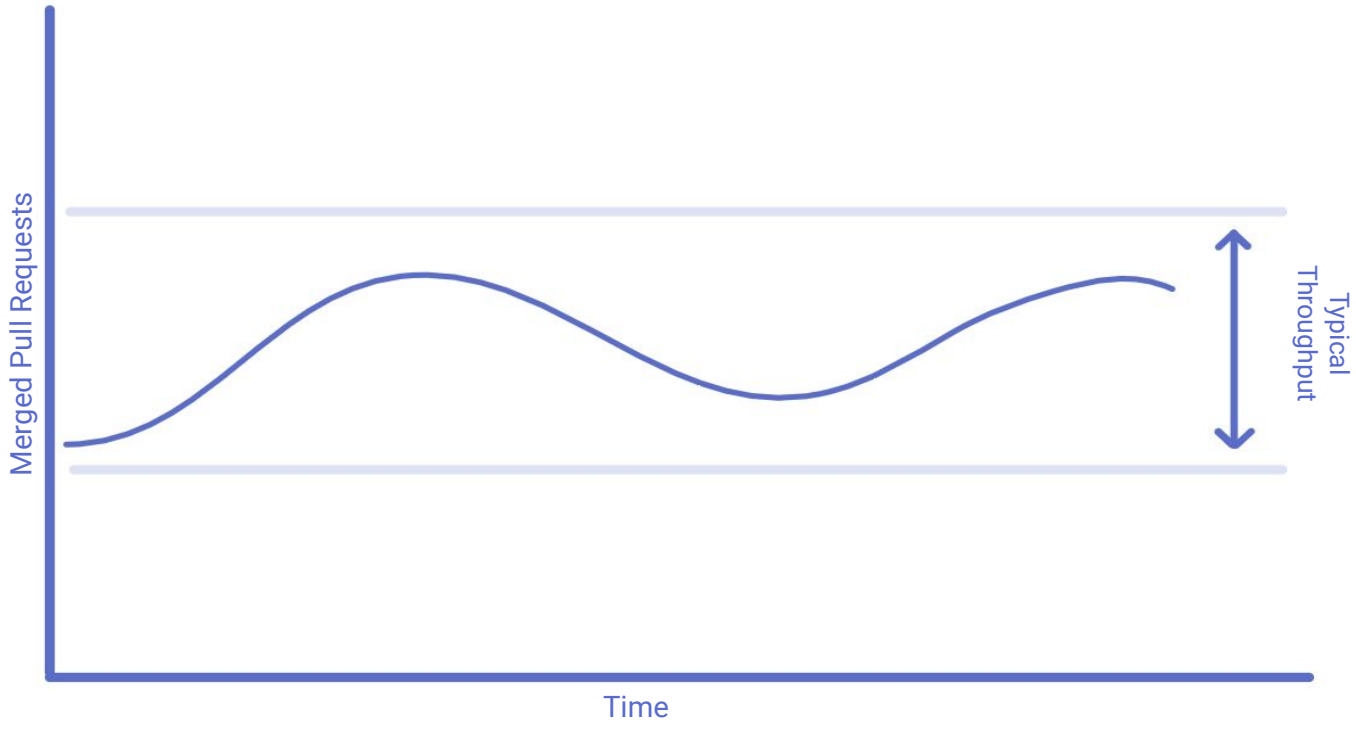
1. Oscillating / Consistent *Cycle Time*
2. Decreasing *Cycle Time*

Cycle time is the time from first commit to merging of the pull request. This gives you a high level view of how quickly your team is delivering. It allows you to visualize the effect of recent changes, track them over time, and see how well your team is improving.

What if it’s increasing?

If you see *Cycle Time* increasing over time, it may be an early indication of inefficiency and opportunities to improve. This can often be caused by technical debt, lack of proper infrastructure or additional tooling or resources needed.

Measuring Throughput



Merged Pull Requests per Sprint

Merged Pull Requests helps understand how much work your team typically finishes during a sprint..

What to look for

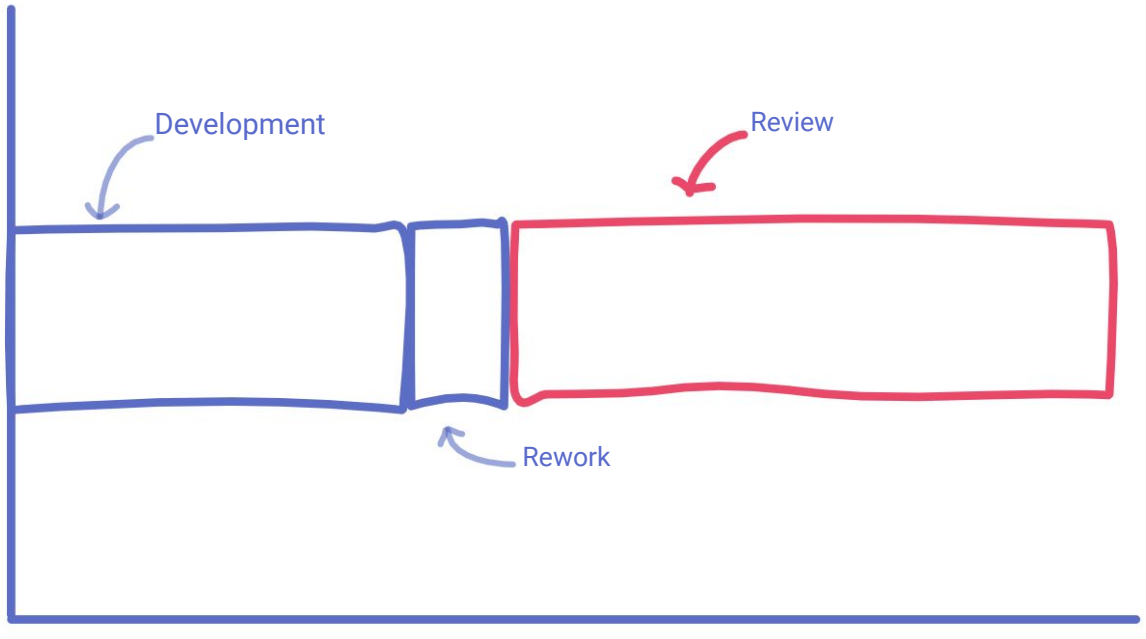
Look for these healthy patterns in your *Output Report* that show your team's typical throughput:

1. Consistent *Merged Pull Requests* over time
2. *Merged Pull Requests* remaining within the typical throughput range

Having a consistent amount of *Merged Pull Requests* shows consistency in your team's throughput. This generally shows that your team is consistent in planning, execution and is a good indicator of healthy sprint planning and estimations.

Merged Pull Requests helps assess how much work your team can typically do in a more objective way than velocity or story points.

Debugging Process



Time Spent in Review

Time spent in *Review* helps show how long your team spends reviewing code during the review process.

What to look for

Look for these unhealthy patterns in your *Process Report*:

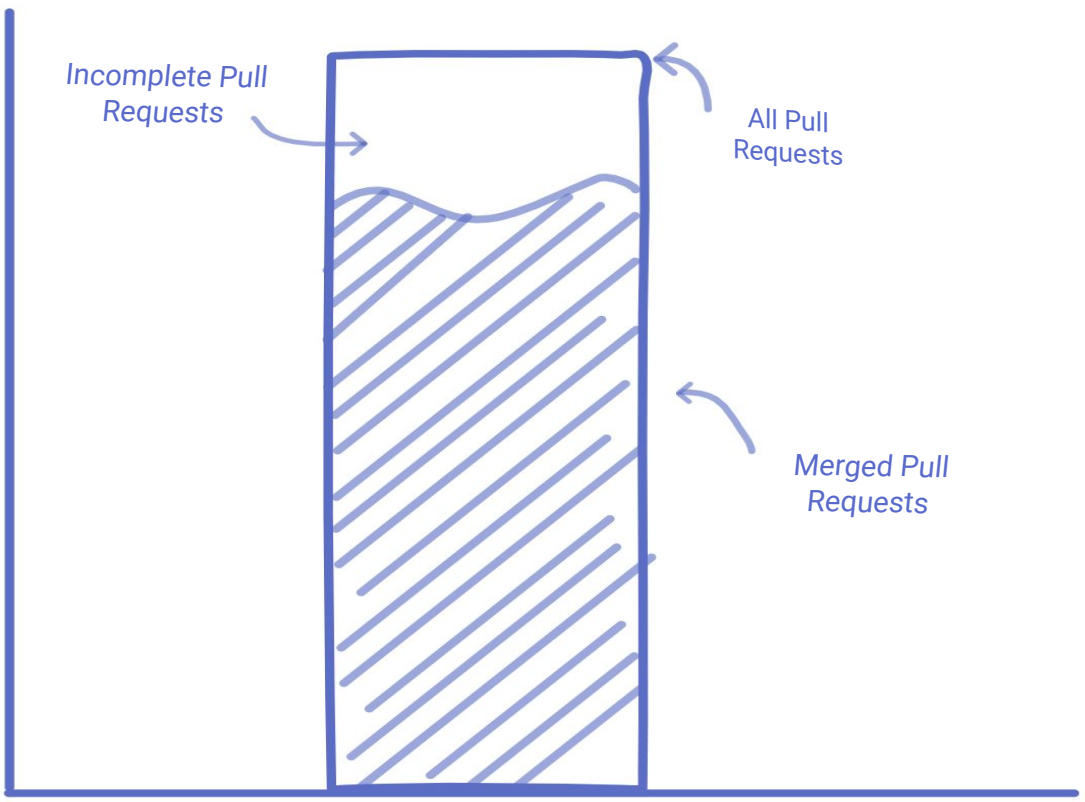
1. Large *Time Spent in Review*

Review is the time spent in the code review process. Although code review is a necessary part of the process, inefficiencies can often cause delivery delays and increases in *Cycle Time*.

Long *Time Spent in Review* can happen for a variety of reasons. Idle waiting time, team review culture, perfectionism and knowledge gaps are some of the most common reasons for bloated review times.

If you see your team is spending large amounts of time in review, head over to your *Code Review Report* to debug the issue.

Measuring Merge Rate



Complete Pull Requests per Sprint

Merge Rate helps understand how much work your team typically starts and finishes during a sprint.

What to look for

It's typically healthy for a team to open and close the same amount of pull requests. Typically what you want to see is::

1. High Merge Rate (>90%)

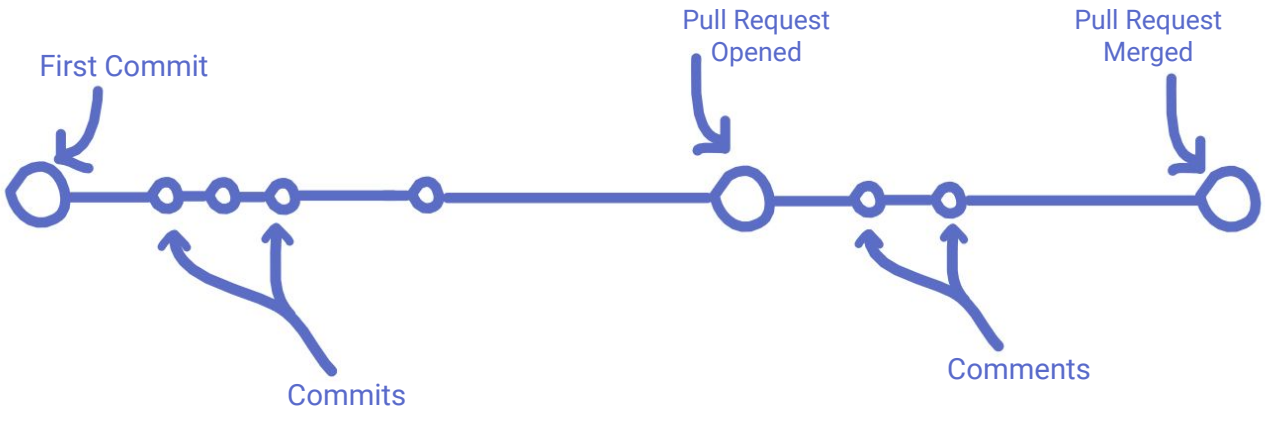
A High *Merge Rate* indicates that the team is opening and closing the same amount of work. This is typically a good indicator of well scoped work that fits into the sprint cycle.

Having a Low *Merge Rate* indicates that *Open Pull Requests* are rolling over from one sprint to the next. Although this can differ from project to project and team to team, it's generally best practice to break up larger projects into manageable chunks that can be completed during the sprint cycle. *Merge Rate* is a great to way to measure if work is rolling over from sprint to sprint.

Where are the bottlenecks?

How to debug your process,
surface inefficiencies,
and promote best practices

Where are the bottlenecks?



Development Process

Visualizing the entire *Development Process* allows you to spot bottlenecks at a glance.

What to look for

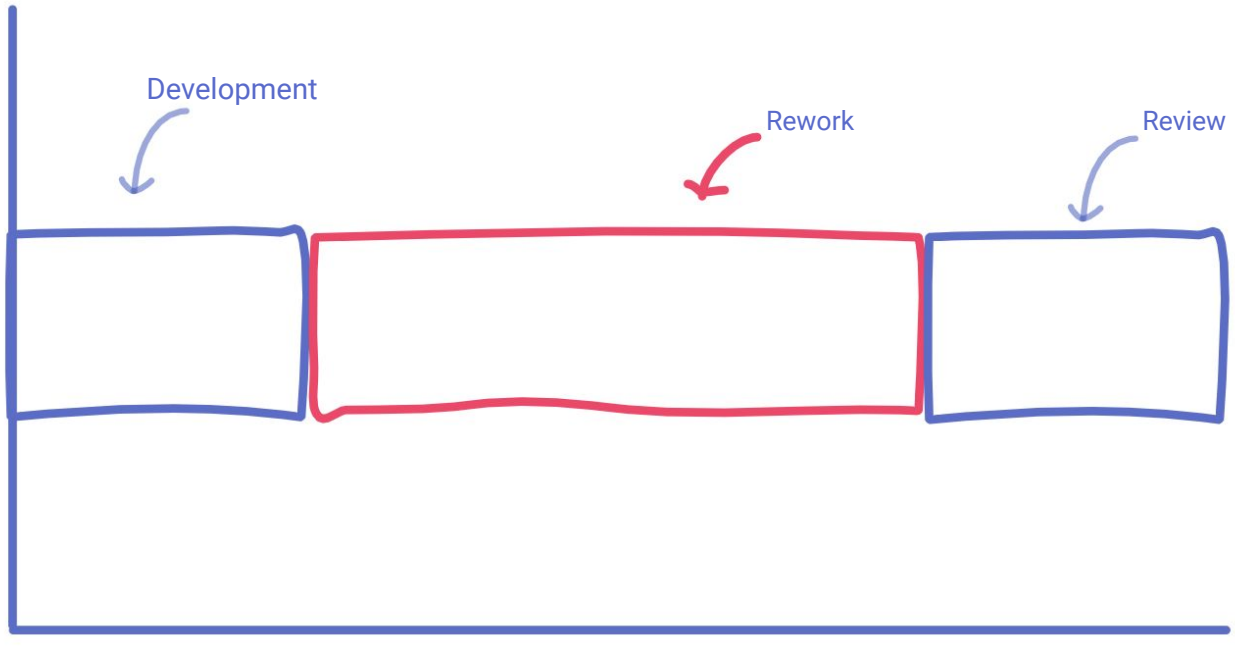
Look for these healthy patterns in your *Process Report* that show your team's typical throughput:

1. Majority of time spent on development work
2. Consistent work throughout the sprint

Most of the development process should be spent developing features rather than reviewing code. Make sure your team is spending a healthy proportion of time on development vs. review.

Showing consistent work (commits, comments, etc) shows a healthy sprint. Of course, there are always exceptions but we've seen this generally indicates that the team is moving along quickly, efficiently, and without blockers.

Debugging Process



Time Spent in Rework

Time spent in *Rework* helps show how long your team spends reworking code during the review process.

What to look for

Look for these unhealthy patterns in your *Process Report*:

1. Large amount of *Rework*

Rework is the time spent in development *after* the review process has already begun. This can happen for a variety of reasons, but you want to make sure your team isn't spending too much time in *Rework*.

Not all *Rework* is bad but be sure it's not a consistent habit for your team. Generally spikes in rework tend to indicate sprint changes, poorly written tickets, bugs/defects found in the review process or overly critical code review (perfectionism).

If you see any of these patterns, make sure to work with your engineering and product teams to find the core issue and reduce the time spent in *Rework*.

Surfacing Inefficiencies

Commits Over Time

Visualizing *Commits over time* can show inefficiencies in the process.

What to look for

Look for these healthy patterns in your *Process Report* that show your team's typical throughput:

1. Large gaps in commit timeline

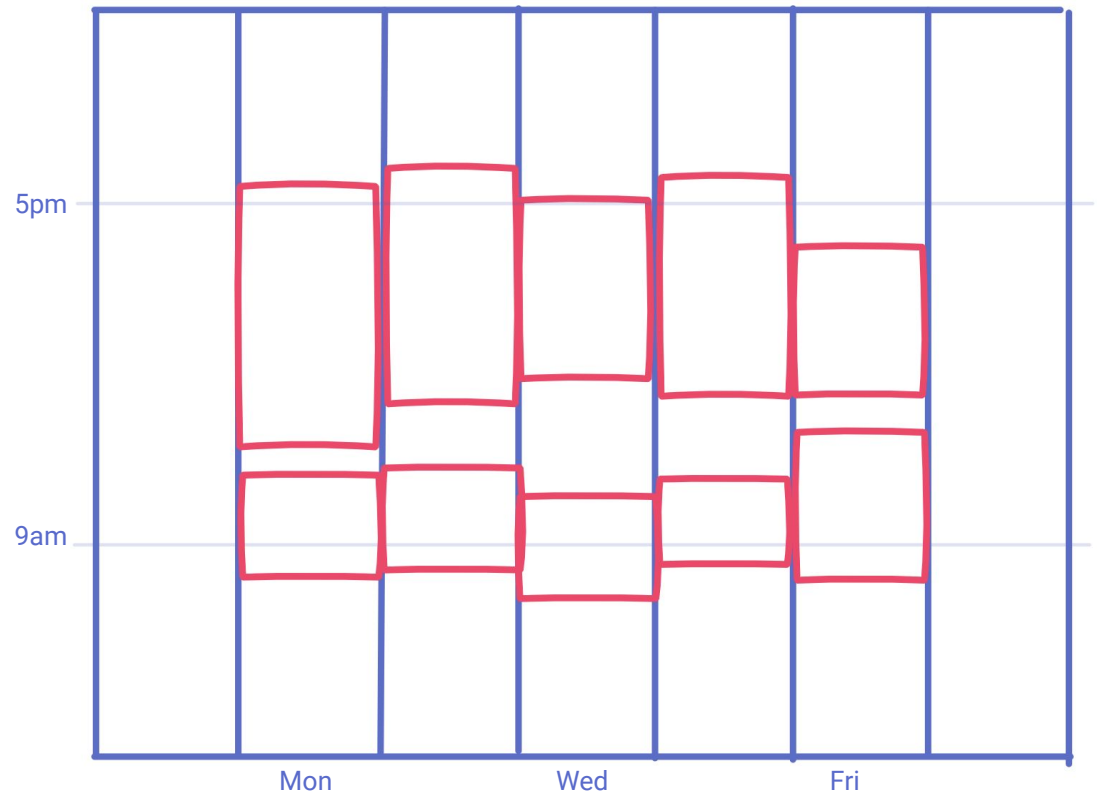
Gaps in the commit timeline are useful to visualize. This can happen for a variety of reasons (not all bad) such as taking time to design or architect proper solutions.

Although not all cases are bad, keep an eye out for large gaps in the timeline. They can often be an early indicator for a developer being stuck or moved to a different project. It's a good way to visualize the effect of blockers and changes in the sprint.

When are we productive?

**How to protect ‘deep work’,
promote consistency,
and avoid burnout**

When are we productive?



Activity Heatmap

Visualizing your team's *Activity Heatmap* allows you to visualize when your team is most productive.

What to look for

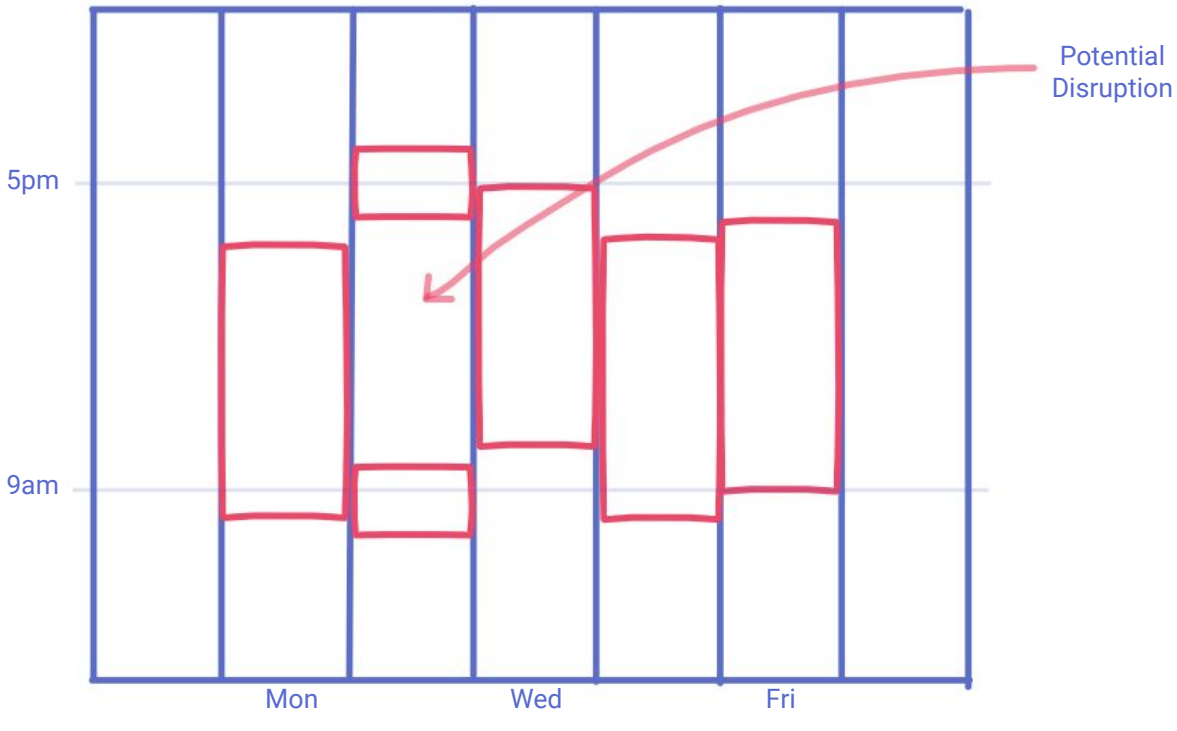
Look for these healthy patterns in your *Process Report* that show your team's typical throughput:

1. Large blocks of activity (at least 4 hours)

The activity heatmaps gives you a high level view of how your team works. You can easily see how your team's work is affected by things such as meetings, deployment schedules and even a noisy office culture.

Use the *Activity Heatmap* and work with your team to remove unnecessary distractions. Use the heatmap to optimize meeting times, deployment schedules and anything else that proves to be a distraction to your team.

Protecting ‘Deep Work’



Gaps in Activity

Visualizing gaps in your team’s *Activity Heatmap* to surface potential disruptions.

What to look for

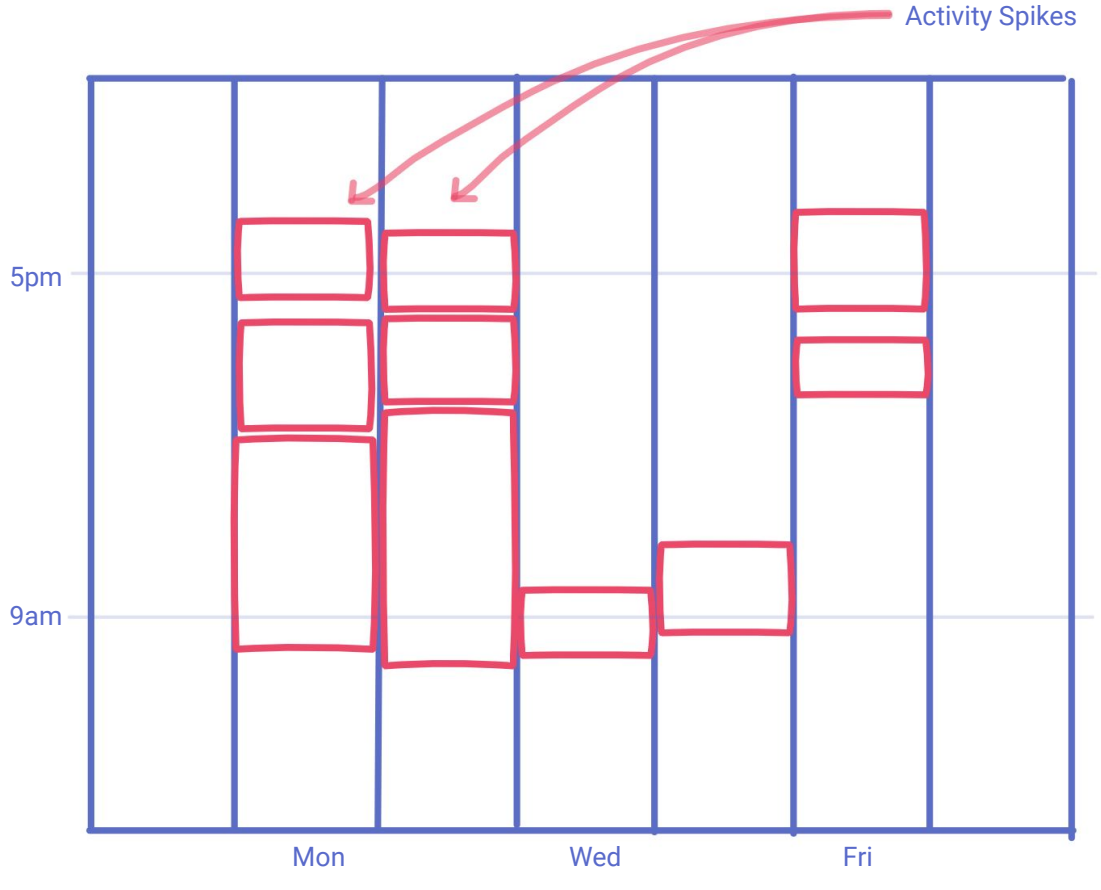
Look for these unhealthy patterns in your *Output Report* that show disruptions in your team’s productive time:

1. Gaps in activity
2. Small, sporadic periods of activity

Deep work is important for developers. It allows them to focus and do their best work. Although meetings can be useful, it’s important to know when they are cutting out of your team’s most productive times.

Use the *Activity Heatmap* to understand when your team works. Work with them to optimize meeting schedules, deployment trains and anything else that might be getting in their way.

Promote Consistency



Activity Spikes

Visualizing spikes in your team's *Activity Heatmap* and surface opportunities to optimize.

What to look for

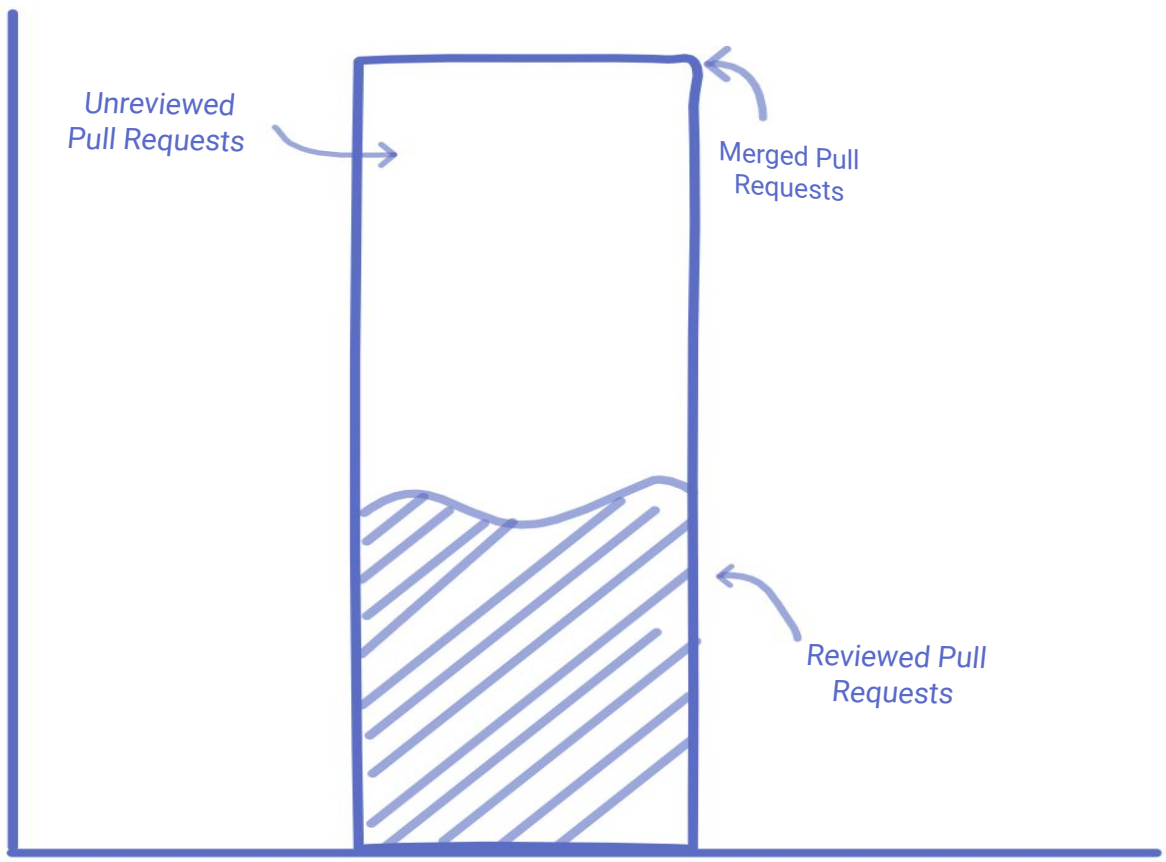
Look for these unhealthy patterns in your *Output Report* that show disruptions in your team's productive time:

1. Inconsistent activity

The graph above is from a team that has a scheduled deployment every Wed. As you can see, Mon and Tues is when most of the work happens.

Use the *Activity Heatmap* to understand when your team works and why that is. In the case above, you may want to try a Tues/Thurs deployment schedule and measure if it improves your team's consistency throughout the sprint.

Moving Too Fast



Low Review Rate

Review Rate is the percentage of merged pull requests that have been reviewed. Use *Review Rate* to see how comprehensive your team's review process is.

What to look for

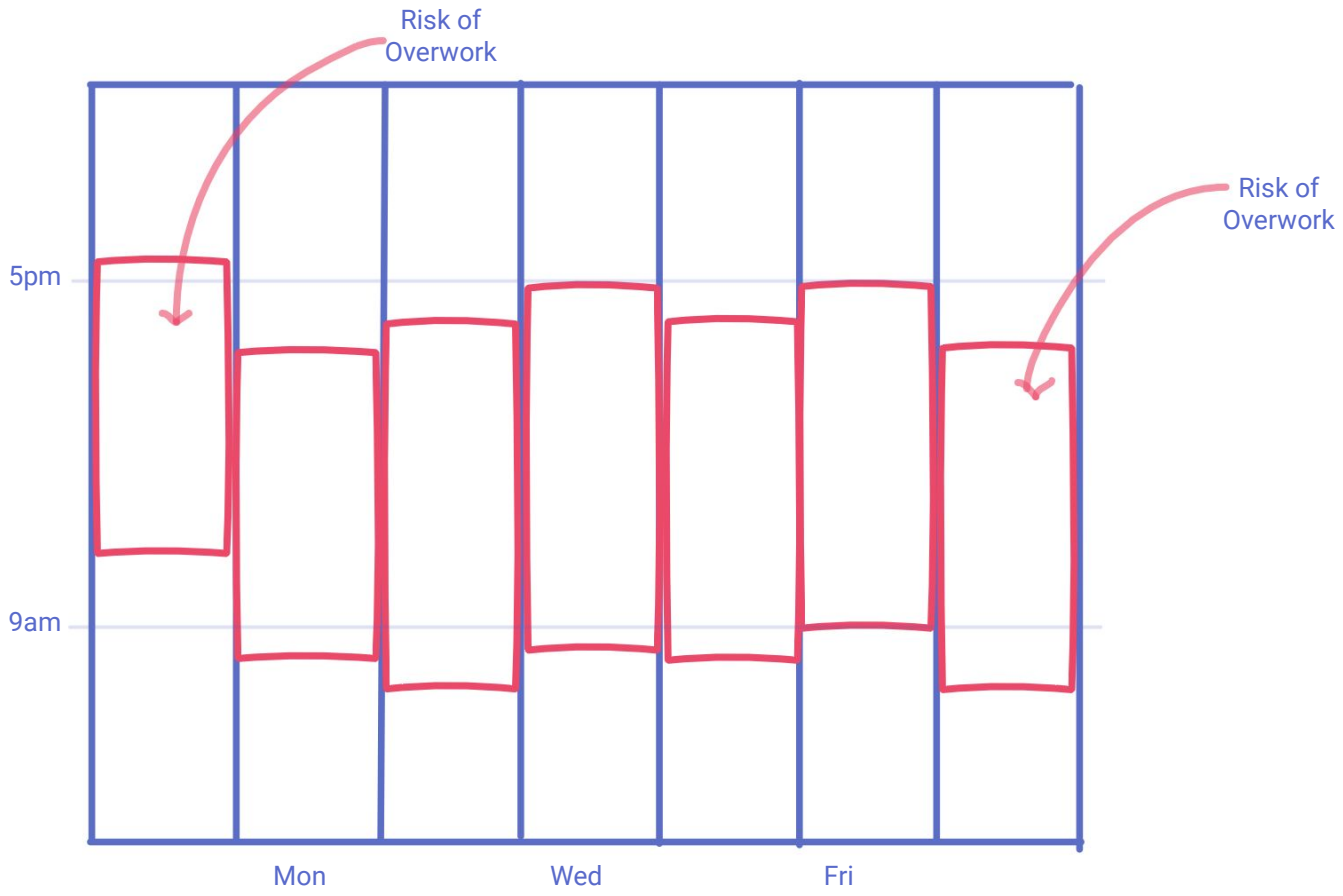
Look for these unhealthy patterns in your *Code Review Report* that show how well your team reviews code:

1. Low Review Rate (<100%)

A Low *Review Rate* indicates that the team is merging unreviewed code. This is typically done through self-merging which shows a lack of quality control.

Sometimes this can be necessary as it allows code to be merged quickly but this is not a healthy long term practice. Preferably every new code addition goes through the review process. Self-merging can cause increase in defects, prevents knowledge sharing and over emphasizes speed over quality.

Avoid Burnout



Unsustainable Working Patterns

Visualizing *Throughput* can show if your team is overloaded with work.

What to look for

Look for these unhealthy patterns in your *Output Report* that show your team's workload:

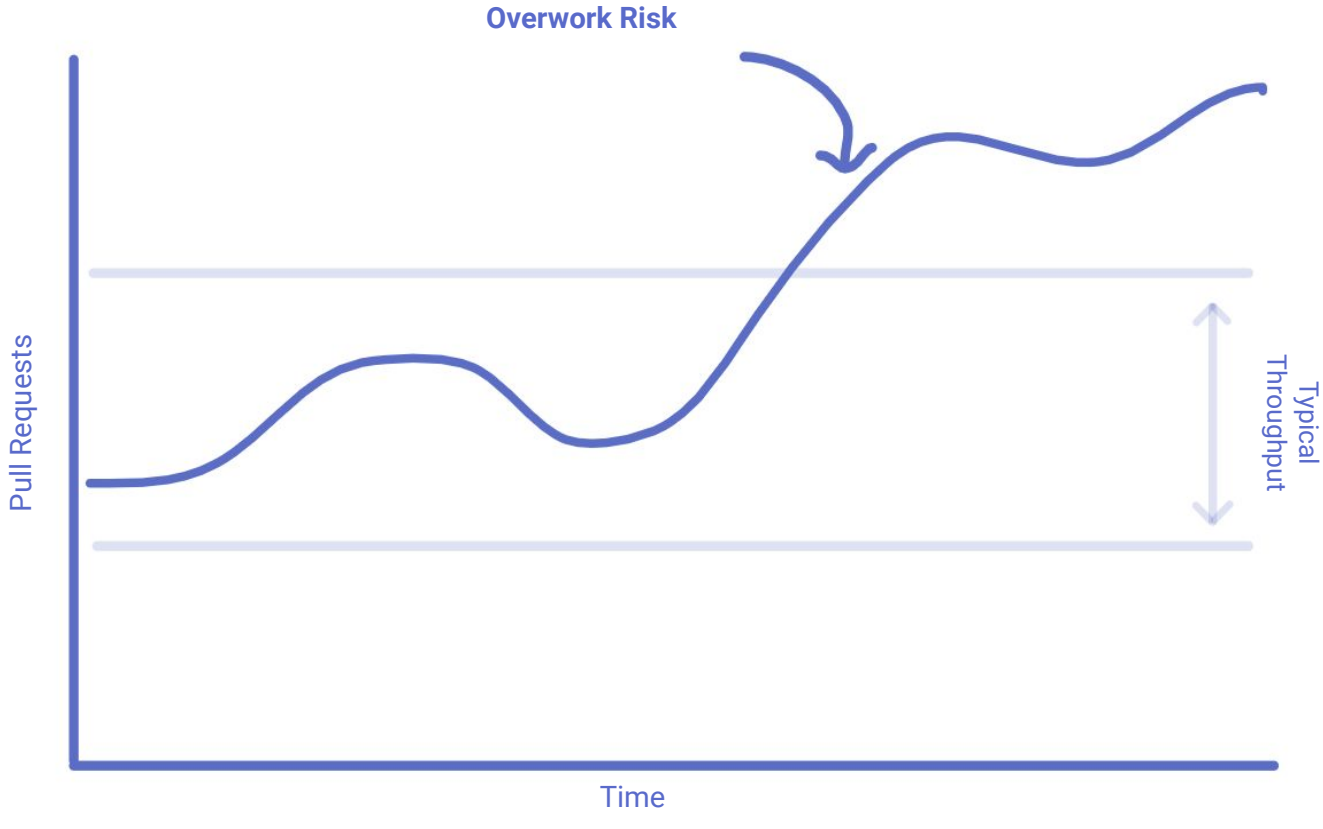
1. Abnormal activity outside working hours

Every person has different preferences on when they like to work. With that said, keep an eye out for abnormal working times as they can be an early signal of burnout.

If your team seems to be consistently working through the weekend, it's a good idea to dive into why that is. This can be a personal preference in some cases, but make sure it's not a consistent pattern.

Rest is good for productivity. Make sure your team is getting the breaks they need and prevent burnout.

Avoid Burnout



Work Overload

Visualizing *Throughput* can show if your team is overloaded with work.

What to look for

Look for these unhealthy patterns in your *Output Report* that show your team's workload:

1. Active pull requests above typical threshold

Your team's workload can often change for a variety of reasons. With that said, it's good to keep track of how much work you're taking on.

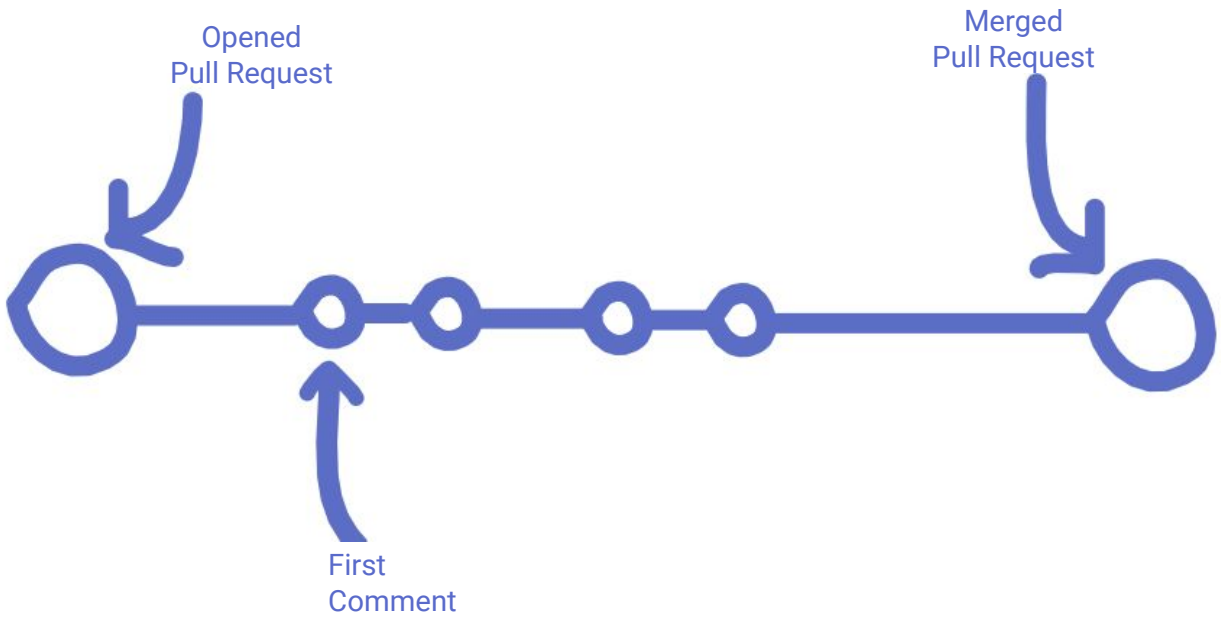
If your team seems to be taking on more than their typical throughput, it's a good idea to dive into why that is. This can often be due to automation, refactors or optimization in the process.

If none of these are the cause, then your team may be taking on an unsustainable amount of work. It's good to keep an eye on this and make sure that the workload doesn't stay above their throughput for long periods of time as it may result in burnout.

How well do we review code?

**Balancing speed vs. quality,
reinforcing best practices,
and creating a healthy review culture**

How well do we review code?



Review Timeline

Visualizing your team's *Review Timeline* gives you a high level picture of how your team collaborates and reviews code.

What to look for

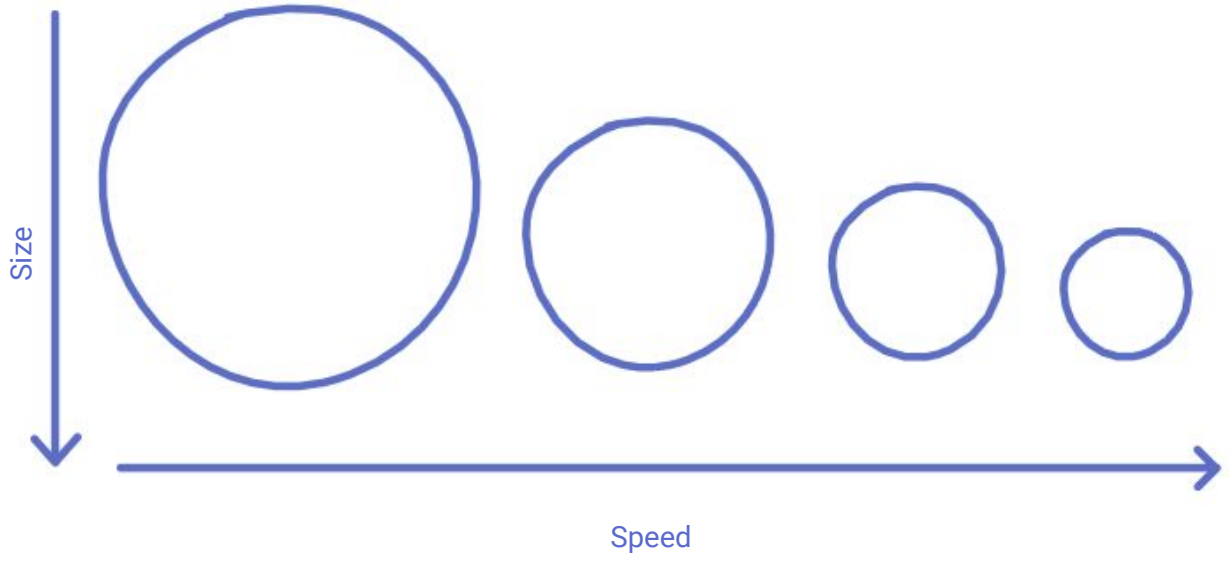
Look for these healthy patterns in your *Code Review Report* that show your team's typical throughput:

1. First comment soon after opening
2. Healthy comment activity prior to merging

The first pull request comment shows the first activity in the review process. Preferably this happens soon after the pull request was opened to reduce the amount of idle time in review.

Look for comment activity to show how your team engages during the code review process. Preferably your team collaborates through comments and moves quickly to merge the pull request.

Balancing Speed vs. Quality



Size vs. Time to Review

Look at *Time to Review* to see how well your team is balancing speed and quality in the review process.

What to look for

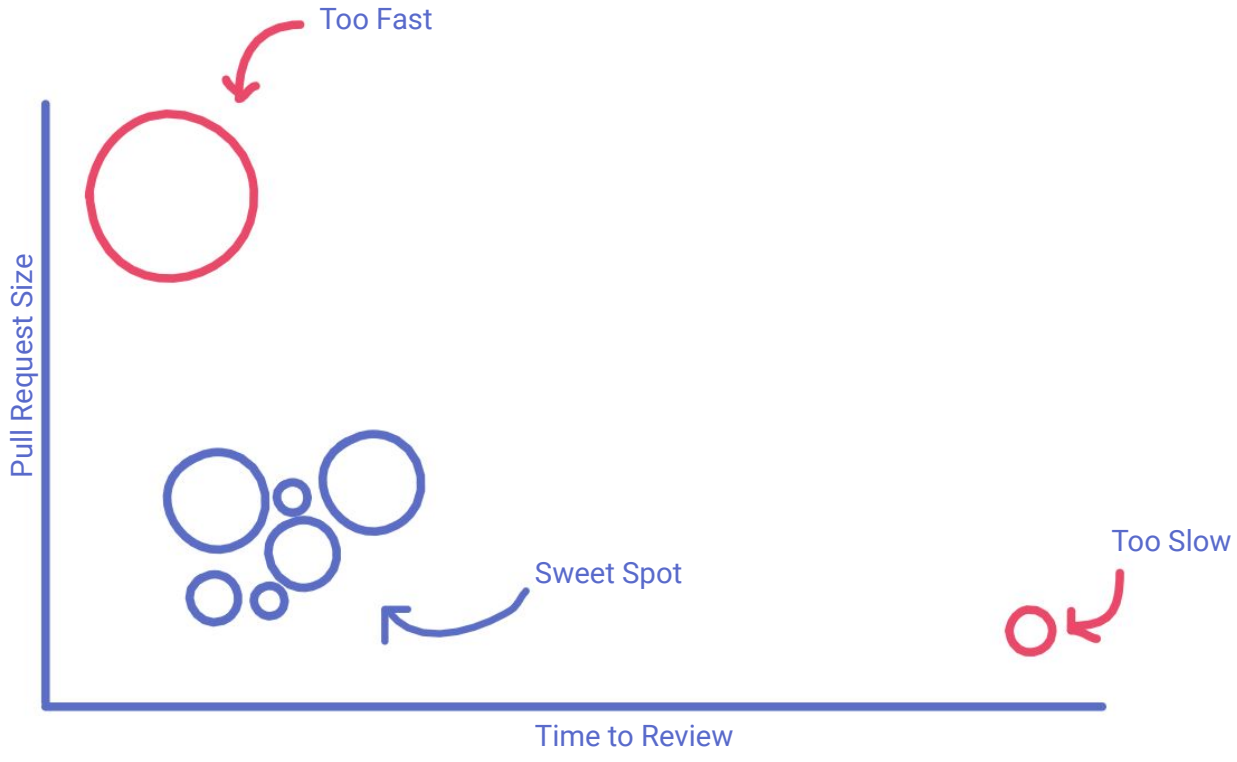
Look for these healthy patterns in your *Code Review Report* that show how well your team balances speed vs. quality:

1. Small Pull Requests with quick *Time to Review*

Sometimes larger pull requests are necessary but it’s good to make an effort to break these into smaller chunks. Large pull requests can have a few negative effects on the review process.

Large pull requests can be overwhelming, prone to mistakes, difficult to review and time consuming. This can often cause slower response times and more bugs slipping through the cracks. Make sure to emphasize smaller, manageable pull requests with your team to encourage a healthier review process.

Moving Too Fast



Size vs. Time to Review

Visualize *Pull Request Size and Time to Review* together to understand how well your team is balancing speed and quality.

What to look for

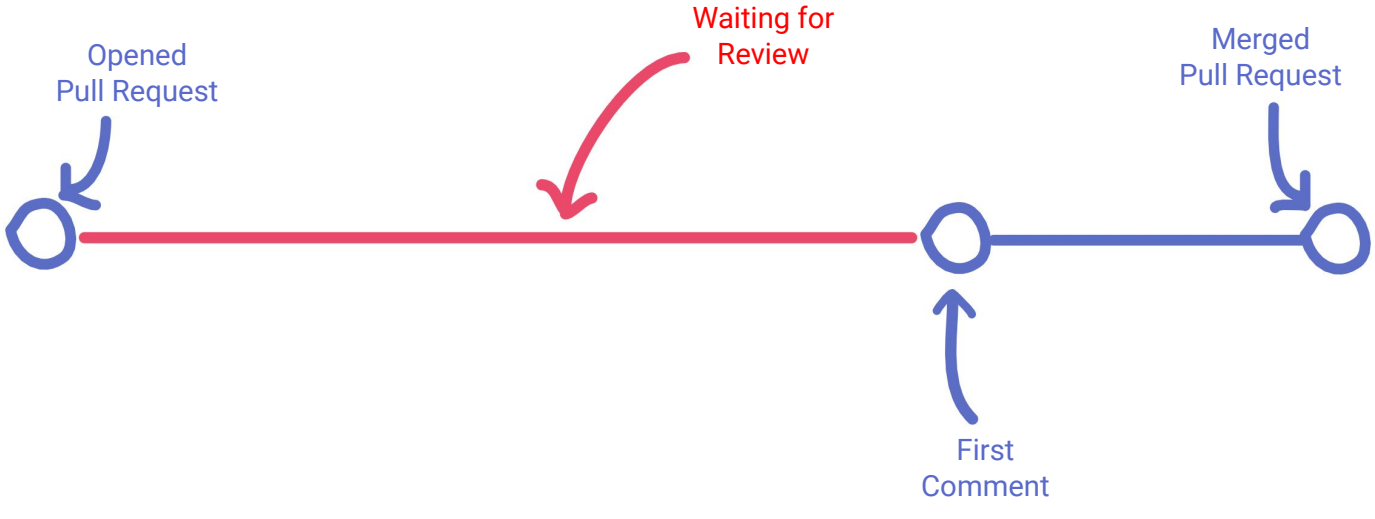
Look for these unhealthy patterns in your *Code Review Report* that show how well your team balances speed vs. quality:

1. Large PR Size, Short *Time to Review*
2. Small PR Size, Long *Time to Review*

Large pull requests will take more time to review. You should always aim for smaller pull requests but in the case where this is necessary, make sure they aren't getting through code review too quickly. This can indicate that the reviewer did not spend adequate amount of time reviewing.

Small pull requests should be smaller, easier to review and therefore move through the review process faster. Watch out for small pull requests that have long review times. There are a lot of reasons this may occur from perfectionism to idle review time. Be sure to follow up on these pull requests when you see them stagnating.

Best Practices



Response Time

Pull Request *Response Time* is used to measure how long it takes the team to start the code review process after opening a Pull Request.

What to look for

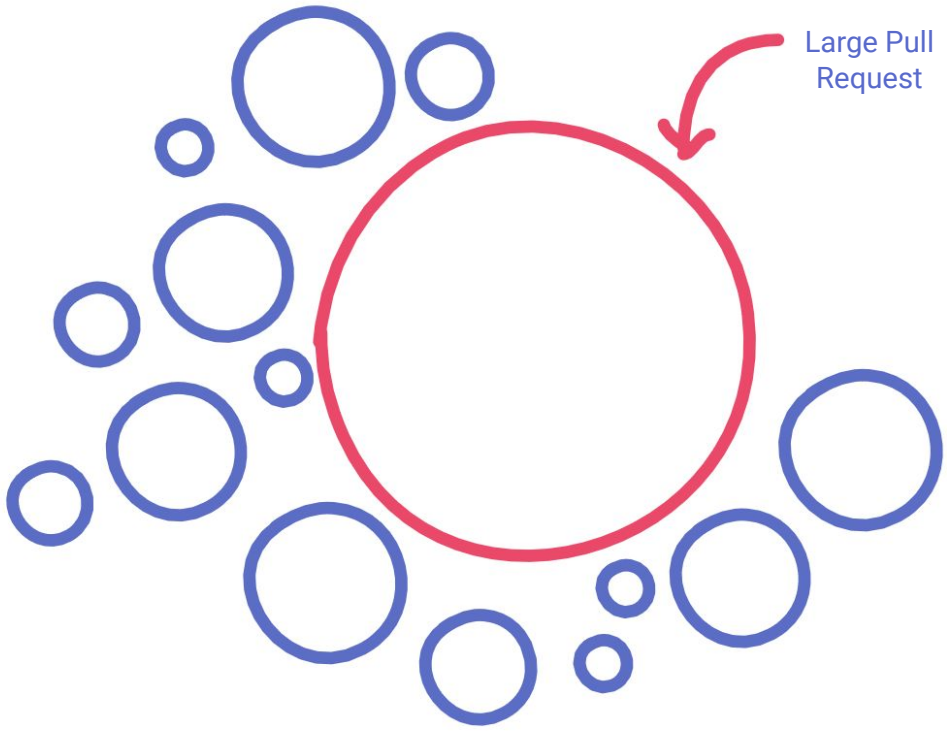
Response Time is calculated from *Pull Request Open* to the *First Comment*. Look for these unhealthy patterns in your *Code Review Report*:

1. Long *Response Time*

Long *Response Time* means the pull request is idle in review for longer than necessary. This can happen for a variety of reasons such as large pull request size, too much concurrent work, or an imbalance in review distribution.

If you see Long *Response Time*, it’s good to meet with your team and discuss ways to reduce this time. Automation, formalizing review etiquette and onboarding more reviewers are common ways to help reduce review *Response Time*.

Best Practices



Pull Request Size

Visualize *Pull Request Size* to see how well your team breaks up their work.

What to look for

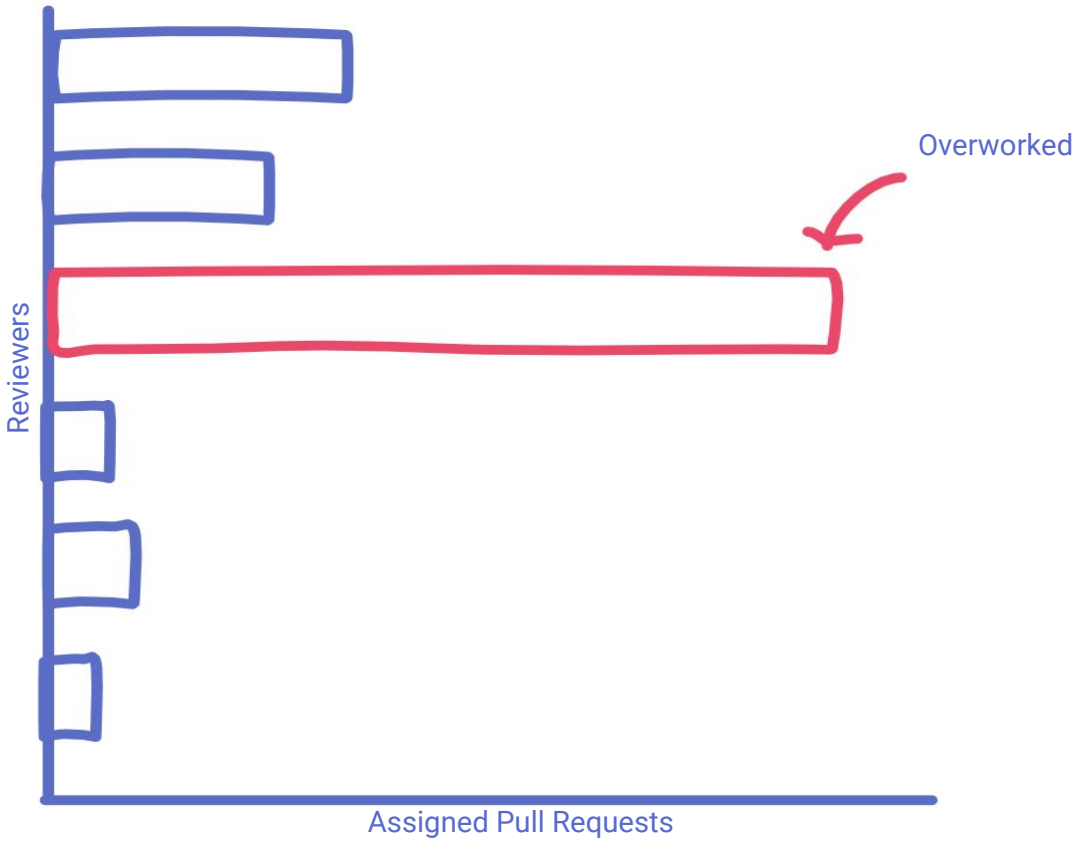
Look for these unhealthy patterns in your *Code Review Report*:

1. Large *Pull Request Size*

Large Pull Requests are sometimes necessary but be sure this isn't the typical behavior. Large pull requests are difficult to review, prone to oversights and time consuming.

If you're seeing your team consistently open large pull requests, it may be time to deep dive and figure out why that is. It's important to emphasize smaller pull requests in the code review process as it will help decrease *Cycle Time* and the number of released defects.

Best Practices



Review Distribution

Visualize *Review Distribution* to see how well your team divides up their work.

What to look for

Look for these unhealthy patterns in your *Code Review Report*:

1. Uneven distribution of *Assigned Pull Requests*

Uneven distribution of *Assigned Pull Requests* can quickly show who is overloaded with code reviews. This can cause longer response times, rushed reviews and overwork.

Imbalanced code review assignments can also be an early signal that your team has knowledge gaps. You'll want to balance out your code review assignments and assign multiple reviewers to each pull request.

This typically helps decrease the potential knowledge gaps on the team and onboard additional developers onto the project. These efforts will help increase the review quality while decreasing response time. It also has the added benefit of decreasing knowledge gaps and speeding up *Cycle Time* in the long run.

How well do we share knowledge?

How to spot bus factor risk, encourage communication, and fix knowledge gaps

How well do we share knowledge?

Knowledge Sharing

Visualize how well your team shares knowledge in the *Code Review Report*.

What to look for

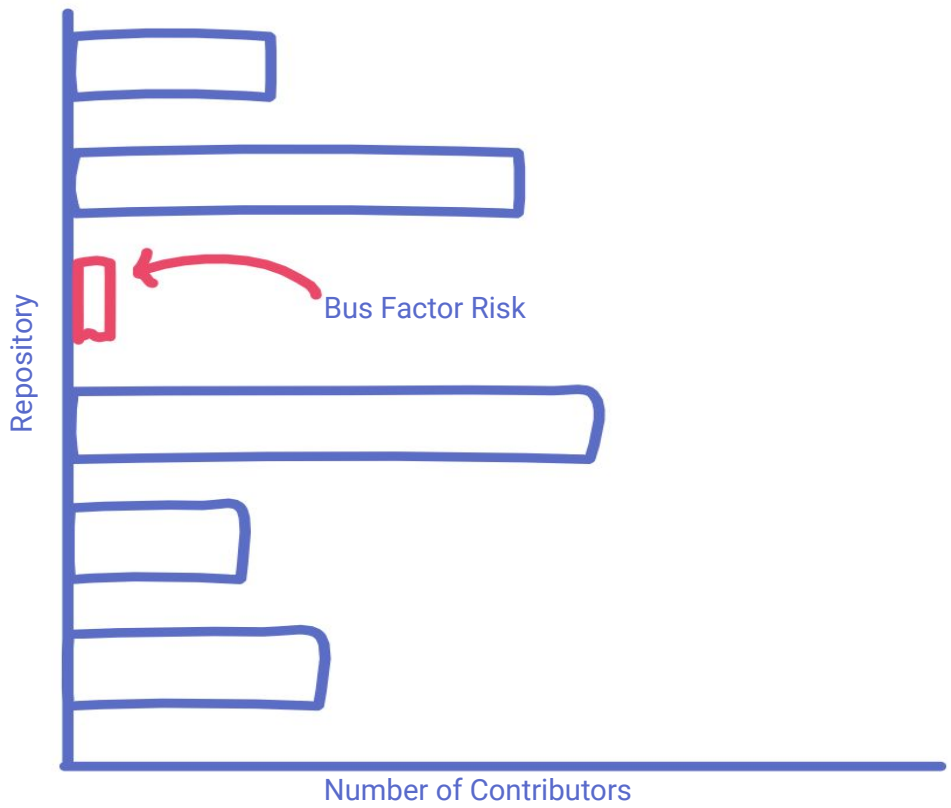
Look for these healthy patterns in your *Code Review Report* that show how well your team distributes knowledge:

1. Balanced distribution of repository
    a. Contributors
    b. Reviewers

Having a healthy balance of repository contributors and reviewers shows strong team knowledge transfer.

More contributors that understand the code shows that knowledge is distributed. Having more developers reviewing code helps to keep knowledge gaps from forming overtime while onboarding new developers onto the code base to reduce bus factor.

Spotting Bus Factor



Number of Contributors

Use *Number of Contributors* per Repository to quickly see where they may be knowledge gaps and risk of a low bus factor.

What to look for

Look for these unhealthy patterns in your *Code Review Report* that show how balanced your team's knowledge is:

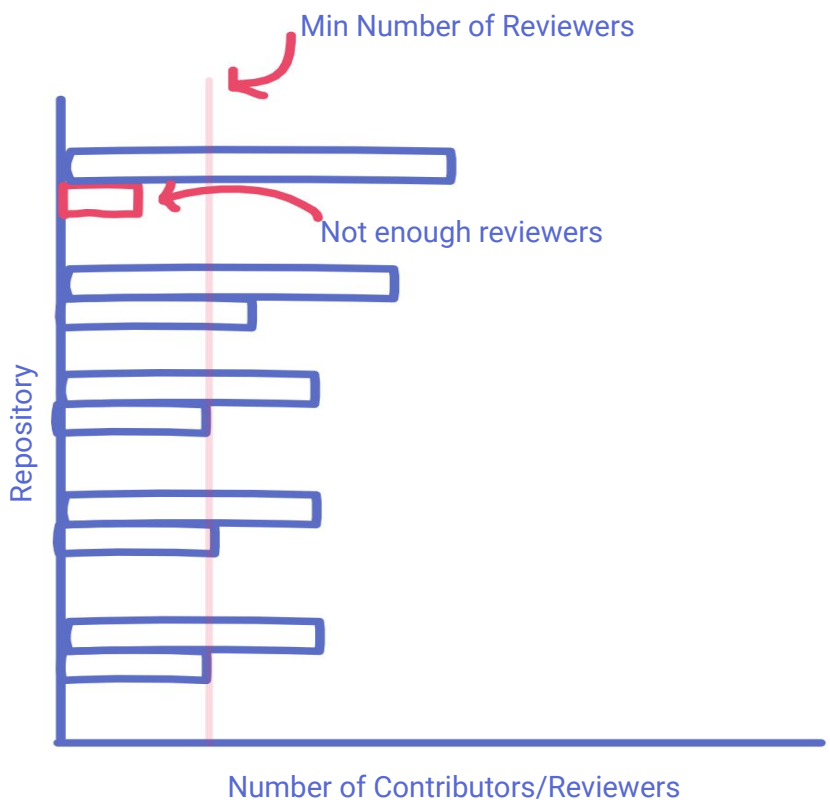
1. Unbalanced distribution of contributors

Having an imbalance of contributors across repositories signals a low bus factor and lack of knowledge distribution. This is typically a bad sign.

Not only is this risky if a sole developer decides to leave the company but it also leads to increased cycle time, review time and inconsistency in performance. Having knowledge gaps also tends to produce overwork, burnout and stagnation in work that can lead to developer unhappiness.

If you see this situation, it's important you take the steps to fix it. Try onboarding more developers onto the repository to provide support. It also helps to assign multiple reviewers to pull requests to help spread the knowledge.

Encouraging Communication



Number of Reviewers

Use *Number of Reviewers* per Repository to quickly see where they may be gaps in communication.

What to look for

Look for these unhealthy patterns in your *Code Review Report* that show how your team distributing knowledge:

1. Low number of reviewers

Having a low number of reviewers on a repository is typically a bad sign. This is an early signal that there are knowledge silos within a team.

Having a low number of reviewers on a repository typically increases response time, review time and cycle time. It also increases the risk of overwork and imbalanced workload.

If you see this pattern, make sure multiple reviewers are being assigned to pull requests. This helps spread the knowledge over time and reduce the burden for the existing reviewers.

About Haystack

We are a team of experienced CTOs, engineering managers, and software engineers. After working with hundreds of engineering leaders, we've identified common patterns hidden in most software teams.

We created this company, and this book, to help engineering leaders identify common patterns to look out for so as to maximize productivity, and minimize burnout.

Haystack was founded in April 2019 in hopes to build advanced pattern recognition solutions built directly into your Git that will take the manual aspects out of detecting these key moments in your team's life.

For more information:
visit us at **usehaystack.io**
reach out to us at **sales@haystack.io**