The ultimate guide to code reviews Edition II

For the anxious VP of engineering

www.codacy.com

Index

Introduction **3**

Diagnosing the most common concerns **5**
The biggest problem in your project? It's all about perspective 5
What your team actually does 7
Where code quality fits in the picture 10

The single most effective way to improve code quality **11**

Five rules to set up code reviews **13**

Conclusion **23**

Cheatsheet **24**

About Codacy **25**

Introduction

Quality is not an act, it's a habit
- Aristotle

Code reviews are an important component of the modern software development workflow. They make it easy to spot problems early on, disseminate knowledge of code, and share accountability of design decisions, amongst many other benefits.

There is an implicit relationship between code quality and code reviews. Intuitively, you might say that a higher number of code reviews leads to better code quality — but are we sure that's the case? Code reviews can play an important role in our workflow, but they're often also a source of frustration for teams, leaving them with less time to ship features.

Most teams know that code quality correlates strongly with a number of benefits: developer productivity, product robustness, security, extensibility, less time and resources spent on maintenance, and so on.

But for Engineering VPs and team leaders it's a challenge that some decisions which lead to higher code quality don't necessarily have a clear ROI in the short run. In fact, your customers and corporate stakeholders don't care about the technical complexity of your products, the number of browsers you have to test, or how systematic you are with your unit tests. They care that you deliver the product as quickly as possible.

It’s important to keep a healthy balance between code reviews, maintenance and new feature delivery — otherwise the team will end up taking shortcuts while technical debt gradually accumulates. This can have a real impact in terms of developer productivity, product performance and customer satisfaction.

That’s why we created this book — a guide that explains why code quality is important and dives into the right way to do code reviews. For the second edition of this guide we’ve surveyed 263 developers to better understand their most pressing issues, and which development practices worked best for them to consistently high quality code. Some things have changed, some things stayed the same, and we’ve updated our observations to reflect the current state of the developer landscape.

It doesn’t matter what your exact job title is — if you manage a team of software developers and are responsible for delivering high-quality code, this book is for you.

Let’s jump in!

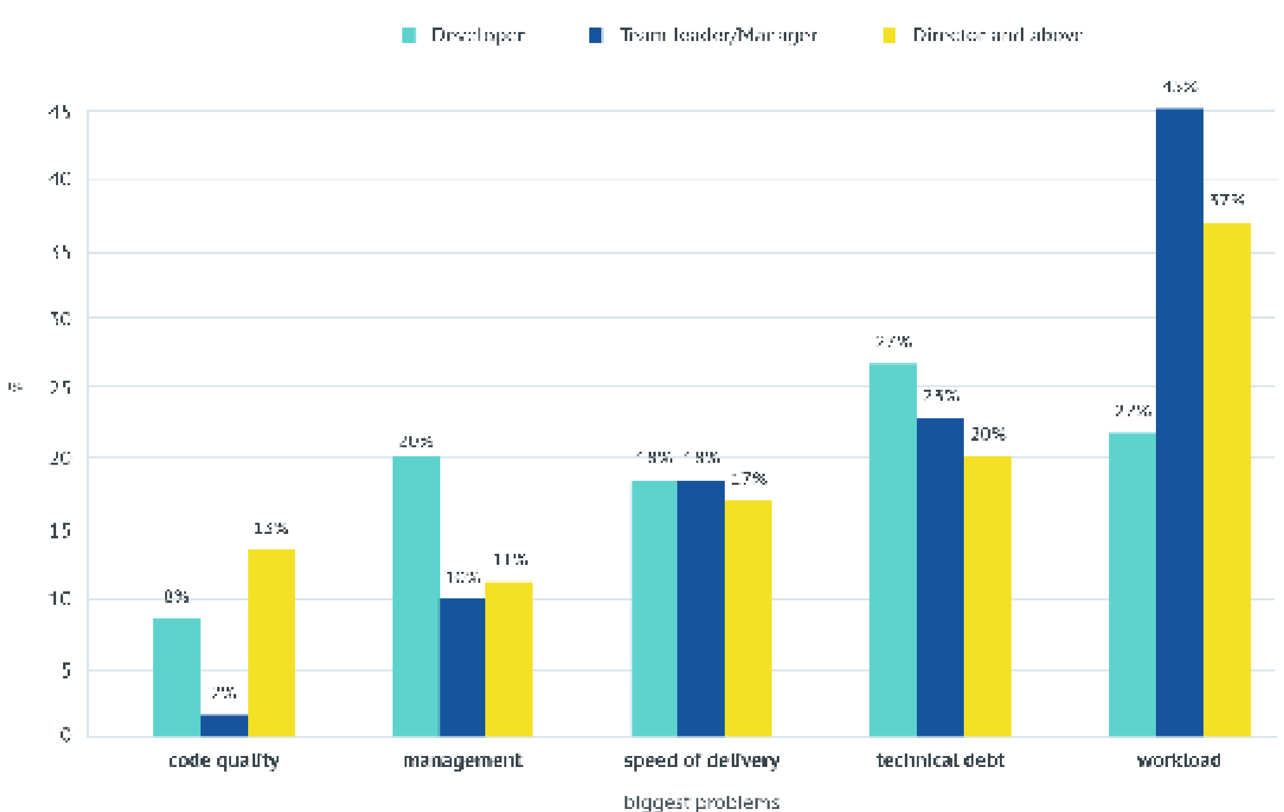
Diagnosing the most common concerns

Whether you're an Engineering VP, manager, team lead, CTO or Chief Engineer — you probably have a good idea of the biggest issues your team is facing. But are the issues specific to your company, its size or the industry?

Let's take a look at some of the biggest challenges team leaders face today through the eyes of our survey respondents.

What's the biggest problem in your project?

As part of the survey we asked engineers and team leaders what they saw as the **biggest roadblock** in their project. Respondents were forced to pick only one issue to identify the most pressing one. These were the results:



KEY INSIGHT 1

The more senior the respondent, the more speed of delivery is important. Conversely, the less senior the respondent, the more technical debt is important.

Last year's data showed us that the more senior the respondent, the more important speed of delivery is to them. That changed – today, speed of delivery is important to all stakeholders, no matter their position. Something that didn't change is that the less senior someone is, the more they care about technical debt.

In general, team managers see code quality as less important than last year – while it was marked as the biggest problem by 11% before, this time around only 2% saw it as their principle roadblock. For developers the importance has halved, meaning that they share this sentiment.

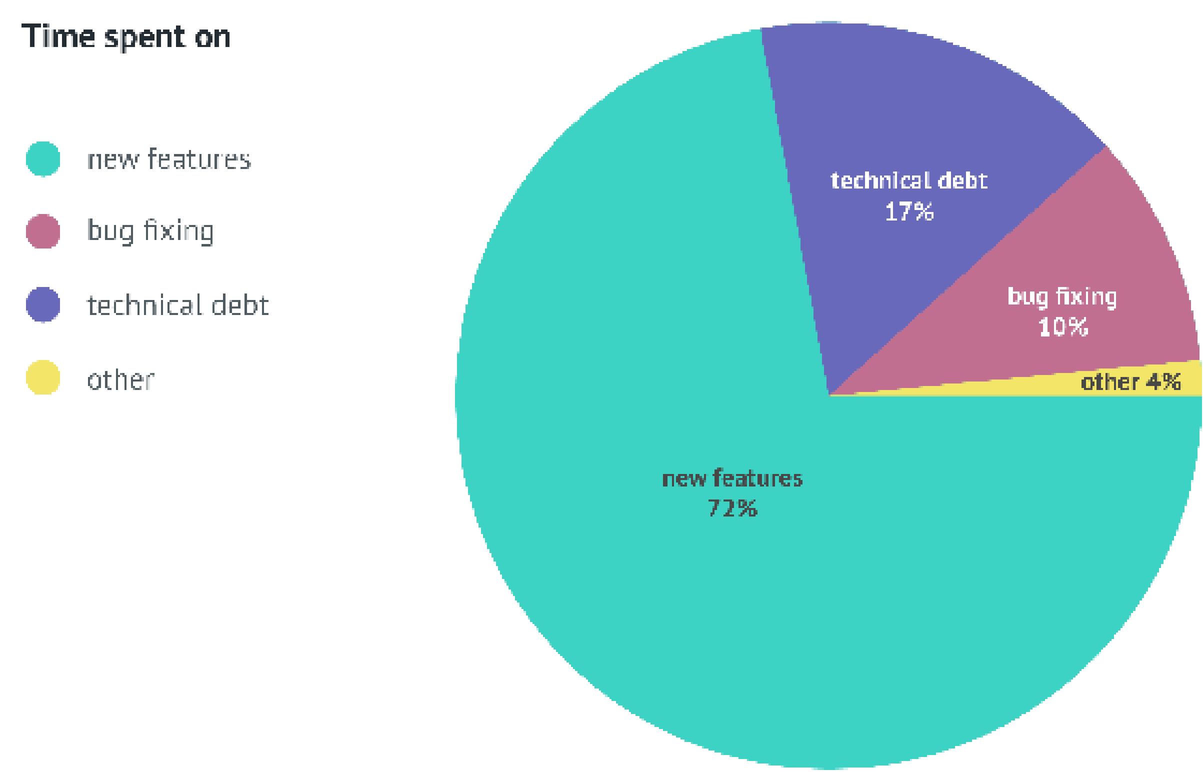
All of this begs the question: are the incentives of team managers and Engineering VPs different than those of developers?

Not really.

What your team actually **does**

Technical debt and speed of delivery are two sides of the same coin – less technical debt means more time spent on productive work. Because of this, your developers will be happier, more productive and the product will have a shorter time-to-delivery.

We asked developers on how they use their time and broke it down into categories:



Time spent producing new features has significantly increased. Last year, we saw that teams spent a large amount of their time fixing bugs and focusing on technical debt. In total, they were spending 45% of their time doing something else other than working on new features. This has changed dramatically in our new survey – the time spent on fixing bugs and technical debt has greatly reduced to take up only 26% of their time. As a result, they're spending 21% more time on building new features for the product, marking a significant increase.

There's also a notable difference in how time spent is being perceived across seniority levels – people in more senior positions generally think that more time is being spent on addressing technical debt, while developers feel more time is being spent on fixing bugs. There's also an inverse relationship between the perception of the importance of technical debt versus the actual time spent on it. More senior stakeholders put less importance on technical debt as a problem in their projects, while less senior employees feel that more time is being spent on it.

KEY INSIGHT 2

Teams are spending more time on building new features – while they previously spent 45% of their time on this, it has now dropped to 26%. At the same time, they're spending 21% more time on creating new product features.

In our research we also noticed that companies of over 50 people spend most of their time working on new features. The importance of technical debt slowly increases with company size, but it stays the same for companies up to 50 employees. Also, the total time spent on fixing bugs becomes less with larger companies – but it doesn't matter how old the company is.

Young companies that are under a year old dedicate the most time to bug fixing when compared to older companies, while businesses between 1 and 2 years old spend a lot of time on technical debt – a whopping 39%. This is probably caused by having to repay for time spent on accelerated feature development in their first year.

A short word on controlling factors – we found that company size and age had an impact on the above results. In sum: All else being equal, smaller and younger companies put greater emphasis on speed of delivery, while larger and older companies put greater emphasis on technical debt. This is to be expected, as older and larger companies have older and larger codebases to maintain. Also, over the years more software developers have contributed to the code, many of whom might not be around by the time new integrations or features are added. As the system grows, so does the variance between the original designs and the current system setup, as well as the degree of entropy of the codebase.

KEY INSIGHT 3

As a whole we can say that the larger and older the company, the greater the emphasis on technical debt is – if a company is smaller and younger it tends to focus more on building product features and fixing bugs, but there is a much larger focus on technical debt after the first year. The companies that spend the most time (70+%) on new product development are young (less than a year old) and settled companies (over 10 years old). The data shows that while younger companies end up dealing with technical debt in their second year, their times spent on technical debt increases from 7% to 39% – all the while, their perception of technical debt being the company's biggest problem doubles.

Where code quality fits in the picture

To understand the influence of code quality in the development workflow, we asked developers to rate the code quality of their current project on a scale from 1 to 5. Strictly speaking, this rating only reflects the perception of code quality — it's a subjective evaluation that's highly dependent on the person's background and experience.

The outcome of this is that two developers could hold very different views about the same snippet of code. However, our experience at Codacy when dealing with thousands of developers on a daily basis shows that there's a strong correlation between developer perception of code quality and the actual quality of the code. That's why we've established that the code quality rating is therefore, for the purposes of this guide, a good enough proxy for code quality.

Higher code quality should lead to less time wasted on fixing bugs or tackling technical debt — that's why the code quality rating should correlate with the amount of time spent on building new features, dealing with bugs and technical debt. Our research corroborates this idea: we looked at how the time spent building new features, technical debt, and fixing bugs and they all vary with the code quality score.

In our research we found that when the code quality score increases, the time spent on bug fixing decreases almost logarithmically. For everyone who has a perceived quality of 5, the time spent bug fixing increases. Then there's the resources spent on creating new features, which increases a lot between code quality 1 and 2, but sees growth slowing down at higher code quality ratings — even though it keeps increasing. Interestingly, companies with code quality scores 4 and 5 spend 81% of their time developing new features.

Higher code quality results in less time spent fixing bugs or working on technical debt, and more time spent on building new features. These results are broadly the same as last year's survey.

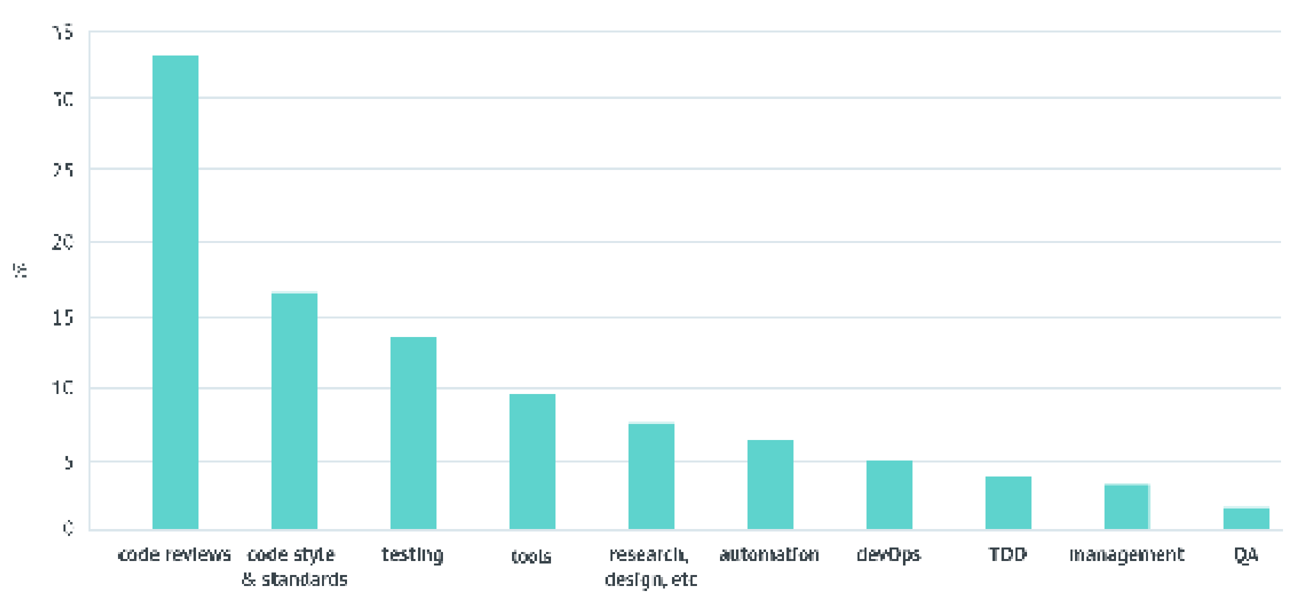
The single most effective way to improve code quality

The ultimate goal of development velocity is to bring value to your users as quickly as possible. Long gone are the days when you could update your software once a year – today development cycles are measured in weeks, sometimes even in days.

This creates the ultimate engineering challenge – balancing quality with speed of delivery. Based on our latest data, companies are increasingly doing a better job at this. We've also seen that higher code quality significantly increases the amount of time spent on building new features, while reducing the time spent on non-productive work.

All of this begged the question which change in their developers' processes had the biggest impact on code quality. We decided to ask our panel:

What change in your development process had the biggest impact on code quality?*



*It's important to note that the question was open-ended to avoid leading the respondents into specific answers.

KEY INSIGHT 5

Code reviews are the single most important thing you can do to increase code quality.

The above insights applied to all types of respondents, whether they worked in small, large, young or old companies – it's time to dig deeper and zoom in on how you specifically can get the most out of your code review process.

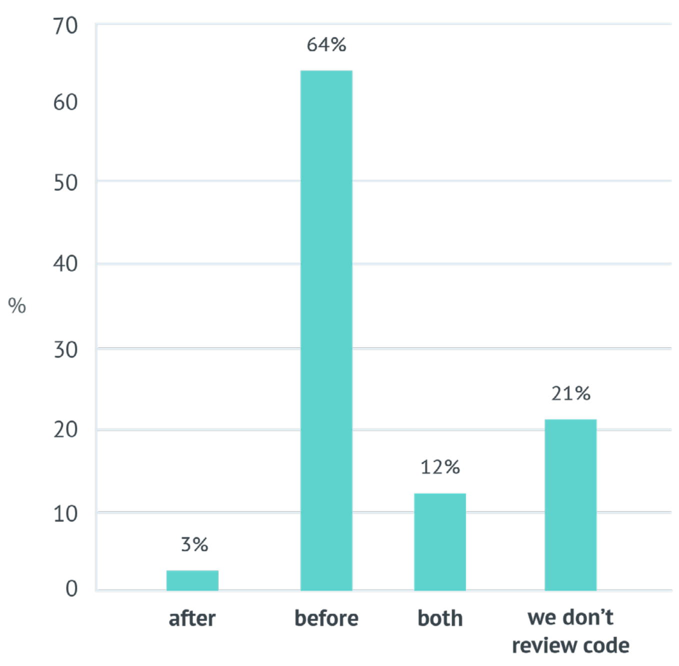
Five rules to set up your code review process

We used our survey to quiz all respondents about a number of code review rules. These included:

1. Should you review code before or after deployment?
2. Who should review the code? Anyone, the owner of the project or a senior developer?
3. How many hours per week does each reviewer spend on code reviews?
4. Should code reviews be blocking?
5. How strict should code reviews be?

Question 1: Should you review code before or after deployment, or not at all?

Our data shows that it's clear which approach has the highest prevalence — just look at the chart below:



But what does this mean? To answer this question, the next step is to investigate whether the different approaches have an impact on the developer's output. We found that doing code reviews before deployment is far more beneficial than not doing code reviews at all. Surprisingly, doing code reviews both before and after deployment performed less well than only doing code reviews before; in fact it performed just as well as doing code reviews after.

For the sake of clarity, we've summarized the statistics in the table below (statistics for "both before and after" and "only after" were merged into one column since they were practically the same.)

	Code review before deployment	Code review after or both before and after	I don't do code reviews
% time spent building new features	54%	50%	46%
% time spent fixing bugs	29%	33%	31%
Need more time to maintain code	66%	58%	68%
Perceived code quality score	**3.3**	**3.3**	**3.2**

Whether respondents reviewed code before or after deployment or not at all, more than half felt they needed more time. Everyone who reviewed code after deployment most often felt that they needed more time to maintain code — more than those who didn't review code at all. People who take care of code review before and after deployment performed the worst, as 58% of respondents felt they needed more time to maintain their code.

This brings us to the following insights:

1) Those who didn't do any code reviews at all performed worst in every single way – they spent the least amount of time on building new features, they spent the most time on fixing bugs, they needed the most time to maintain their code, and had the lowest perceived quality of their code.

2) On the other hand, those who did code reviews before deployment performed best on all metrics.

3) As explained above, those who did code reviews after, or both before and after, did better than those who didn't do code reviews – but not as well as those who reviewed before deployment. This might come as a surprise – after all, you would expect the code quality of something that has been reviewed twice to be higher than when it's only been reviewed once.

However, our experience at Codacy indicates that this isn't the case – companies that review both before and after are more likely to have loosely defined and enforced code review processes. In other words, it's a case of quality over quantity – it's better to review code thoroughly once than to do it badly multiple times.

Based on the above insights, we're able to formulate the following rule:

RULE 1
Don't skip code reviews. Your team will be less productive if you do. And review code **before deployment**. Not after.

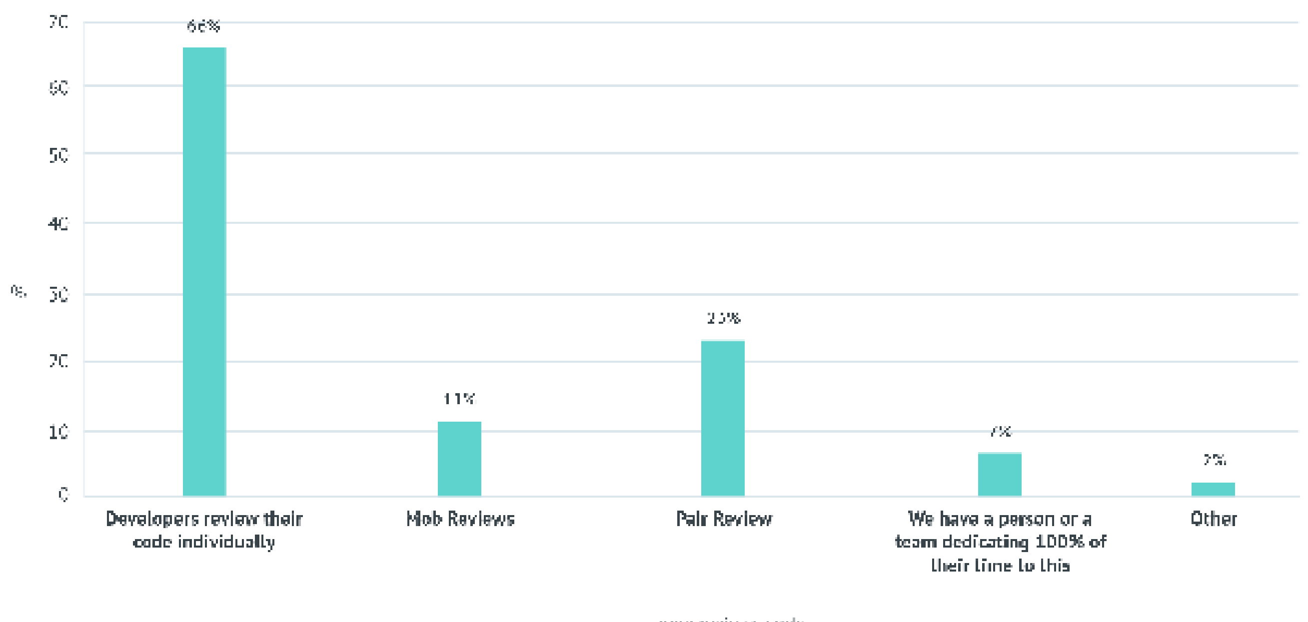
Question 2. How do you review code?

This question is designed to help us determine how the team should do code reviews, aiming to make your team as efficient as possible.

Let's have a look at the possible answers:

- Developers review their code individually
- We use mob review
- We use pair review

The table below shows the prevalence of each option among our respondents.



Most respondents review their code individually – 74% prefer to do it this way, while pair review comes after and mob review is the least popular option.

Again, this could come as a surprise – you might expect companies to only have their most experienced developers review code, essentially making sure that the mistakes of new talent are fixed by older generations. That's not the case, however, as individual code review is totally dependent on the employee's own experience.

To find out why, let's look at the impact of every approach on the following proxies to assess developer productivity and satisfaction – % of time spent on building new features, % of time fixing bugs, % of time working on technical debt, and whether or not they need to assign more time to maintain the code than they currently do.

The results are displayed in the table below, with green highlights showing the best scenarios, and red the less favorable ones.

	All developers get some time to review code	Owners of the projects or modules	Some senior developers get some time to review
% time spent fixing bugs	52%	57%	56%
% time spent on technical debt	22%	22%	21%
% time spent building new features	37%	30%	27%
Should have more time to maintain code?	68%	52%	62%

It seems that the approach to have all of the developers review code is also the best one – it's the option that has developers spend the most time doing productive work, like building new features as opposed to fixing bugs or tackling technical debt. However, people following this approach feel like they need more time to maintain code than teams that do mob reviews.

Code reviews serve two purposes – in the short term it helps improve code quality, and in the long term it facilitates knowledge transfer between engineers, which leads to better, happier, more productive developers. Last but not least, having all developers included in this process is also an empowerment strategy.

As a result, although it may be tempting to limit the task of reviewing code to more experienced developers, in reality it makes sense to allow for all developers to make some time to review code.

This brings us to the second rule:

RULE 2

Make sure all of your developers get to review code from time to time, as this will make them feel empowered and improves their skills.

Question 3: How many hours per week does each reviewer spend on code reviews?

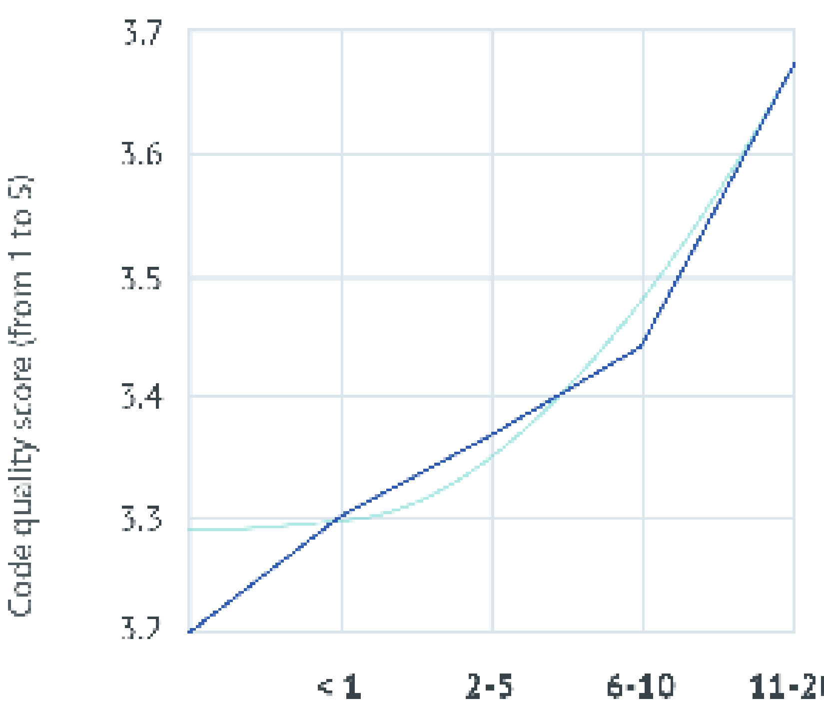
This question is all about time management. Since workload is the second most important concern for engineering teams, understanding how much time they should spend on code reviews is crucial. That's why we asked our respondents — and yet again, we were surprised by the results.

According to our results, the more time is spent per week on code reviews, the higher the code quality score ends up being. The biggest increase happens between 6 and 20 hours per week, where the code quality score takes a sudden jump from 3.45 to 3.7.

This brings us to our third rule:

RULE 3

Make sure your developers spend half a day to one day per week reviewing code — this is the sweet spot for time spent versus high code quality.



Question 4: Should code reviews be blocking?

An intriguing question for an Engineering VP is if it makes sense to make code reviews blocking – this means that code, commits and pull requests can only be deployed when they have been reviewed. While it has obvious benefits in terms of code quality, you might wonder about the impact on your speed of delivery.

This year, 89% of respondents use blocking code reviews, a significant increase from last year's 72%. However, the perceived code quality of these people dropped – this year, it was only 2% higher than those who don't use blocking reviews, compared to 10% in last year's survey.

This tells us two things:
1) You won't be fired for making code reviews blocking – in fact, most people do.
2) Last year we said blocking code reviews improve code quality and reduce technical debt, but this year's data backs this statement less confidently. Still, we think it's a good idea.

This brings us to the next rule:

RULE 4

Make code reviews blocking and don't deploy before they have been carried out.

Question 5: How strict should you be with code reviews?

There are quite a few questions here — how detailed and comprehensive should your team be when reviewing code? Does it impact the ability to ship code or the perceived code quality? How pedantic should you be? And does strictness have an upside?

To find out, we asked our respondents to rate the strictness of their code reviews on a scale from 1 to 5, and correlated their responses with other variables. This is what we found:

1) The stricter the code reviews, the **less time the respondent spent fixing bugs**: a score of 1 (not strict, i.e. "as long as the build does not break, we're alright with deploying this") meant that the respondent would spend 52% of their time on fixing bugs. A score of 5 (very strict, i.e. "we don't deploy until everything is compliant with our code standards") meant that the respondent would spend 35% of their time fixing bugs. After a perceived code quality of 4, the time fixing bugs increases again.

2) The stricter the code reviews, the **more time the respondent spent shipping new features**: a low score of 1 meant that the respondent would spend 47% of their time building. A high score of 5 meant that the respondent would spend 58% of their time shipping new features.

3) Those who had **less strict code reviews felt they needed more time to maintain code**, and those who had **stricter code reviews reported higher code quality**.

The data points out that being stricter in code reviews has clear benefits: less time wasted fixing bugs, more time spent on building new features, a perception of better code quality, and more. It's undeniable that being more strict allows for companies to develop better software. The logical follow-up question is: what makes a code review stricter?

From our experience at Codacy, we believe that being thorough, following checklists and covering all code of a pull request/commit are definitely part of it. Reviewing functionality, logic and design decisions together with code style, security, best practices and common problems constitute a good code review coverage. This brings us to our fifth rule:

RULE 5

Be strict and thorough while reviewing code — your code quality and velocity will thank you.

Conclusion: A work in progress

In this book we explored code quality concerns and compiled code review best practices for engineering leaders.

As a team leader, your top concern is speed of delivery — that's what you're accountable for. While teams from last year's survey spent 45% of their time on technical debt and bug fixes, this number has now dropped to 26%.

There are also other opportunities for optimization that will quickly pay for themselves — higher code quality translates into more time building new features, less time on bug fixes, and less time dealing with technical debt. Our research shows that code reviews are the single most important practice to improve the quality of your project's code.

Quality code means more time spent being productive, and makes for a team of happy engineers — so just remember our five simple rules to sustain a good code review practice.

Cheat Sheet

RULE 1

Don't skip code reviews — your team will be less productive if you do. Also, review code before deployment — not after.

RULE 2

Make sure all of your developers get to review code from time to time, as this will make them feel empowered and improve their skills.

RULE 3

Make sure your developers spend half a day to one day per week reviewing code — this is the sweet spot for time spent versus high code quality.

RULE 4

Make code reviews blocking and don't deploy before they have been carried out.

RULE 5

Be strict and thorough while reviewing code — your code quality and velocity will thank you.

About Codacy

Codacy is a code review and code quality automation platform used by tens of thousands of open source users and industry leaders, including Adobe, Paypal, Deliveroo, Schneider Electric, Schlumberger, and many others.

With Codacy, developers save up to 50% of the time spent on code reviews: The codebase is continuously analyzed to help developers address potential problems early on in the development cycle. Codacy also provides a range of code metrics to help teams track the evolution of their codebase and reduce technical debt throughout your sprints.

Integration with your workflow is easy, thanks to integrations with a wide range of developer tools (GitHub, BitBucket, GitLab, and more).

Codacy is free for public, open source projects, and has a paid plan for private repositories. Enterprise plans are available for organizations who want to host Codacy on their own servers.

You can learn more about Codacy here and sign up for a free trial.