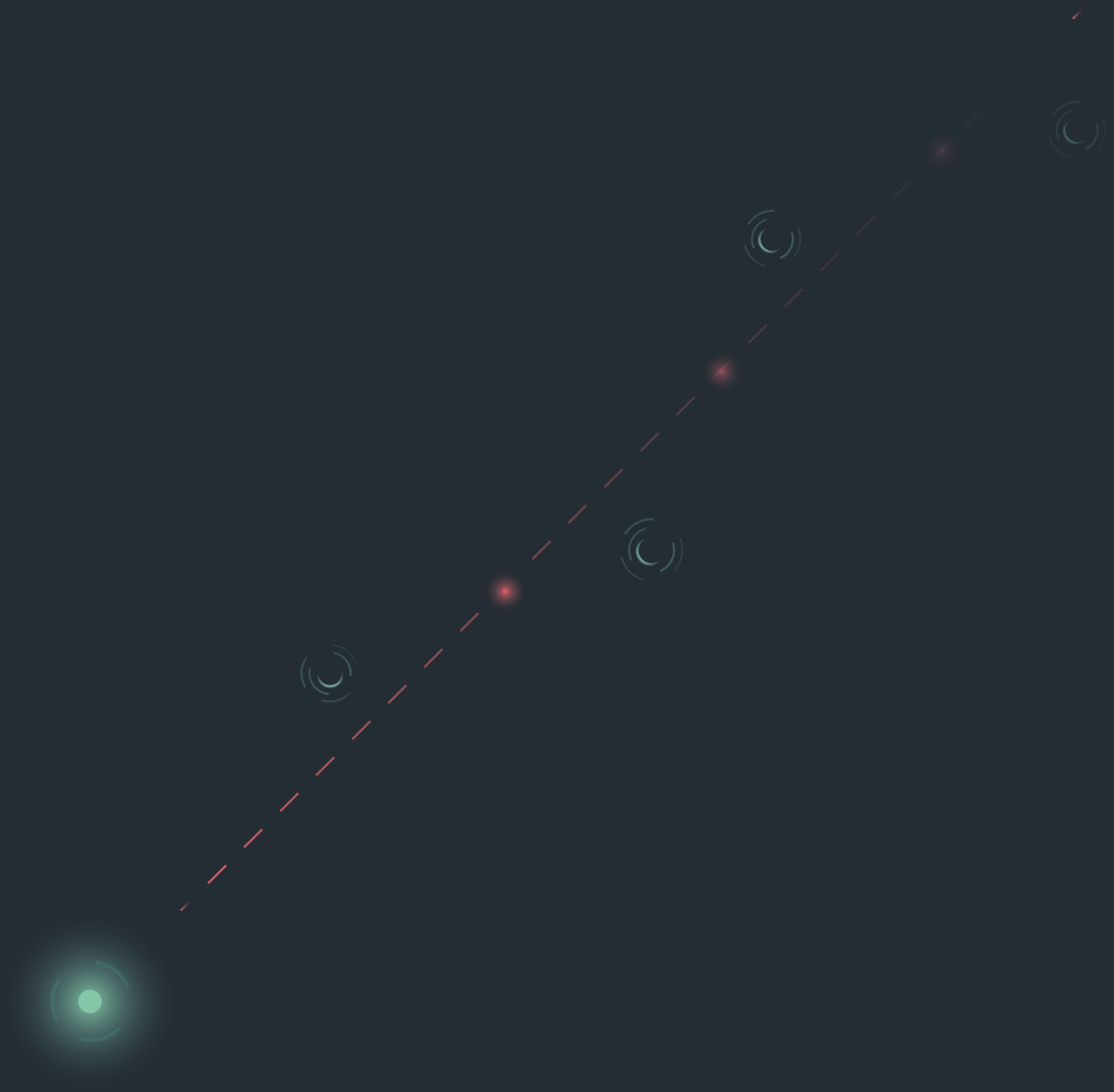
CODACY

The ultimate guide to code reviews

For the anxious VP of engineering



www.codacy.com

Index

Introduction

Quality is the best business plan.
- John Lasseter

Code reviews have become an important component of the modern software development workflow. With code reviews, teams are able to spot problems early, 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, the more code reviews we do, the better code quality we will have in the end. But are we sure of this?
Code reviews play an important part in our workflow but they are also many times a source of frustration for teams as they don't spend enough time shipping features.

All teams intuitively know that great 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 you as a VP of Eng and team leader, the challenge is that some of the decisions leading to higher code quality do not have 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. What they care about is that you deliver the product, ideally yesterday.

If there is not a healthy balance between code reviews, maintenance and new feature delivery, shortcuts will be taken and technical debt gradually accumulates, with a real dollar impact in terms of developer productivity, product performance and customer satisfaction.

This book focuses specifically on code quality and the right way to do code reviews: we surveyed 682 developers to understand what the most pressing issues were, and what development practices worked best to maintain the highest possible level of quality.

Regardless of what route you came by, or what exactly your current job title is, if you manage a team of software developers and are accountable for delivering the goods, this book is for you.

Let’s jump in!

Diagnosing the most common concerns

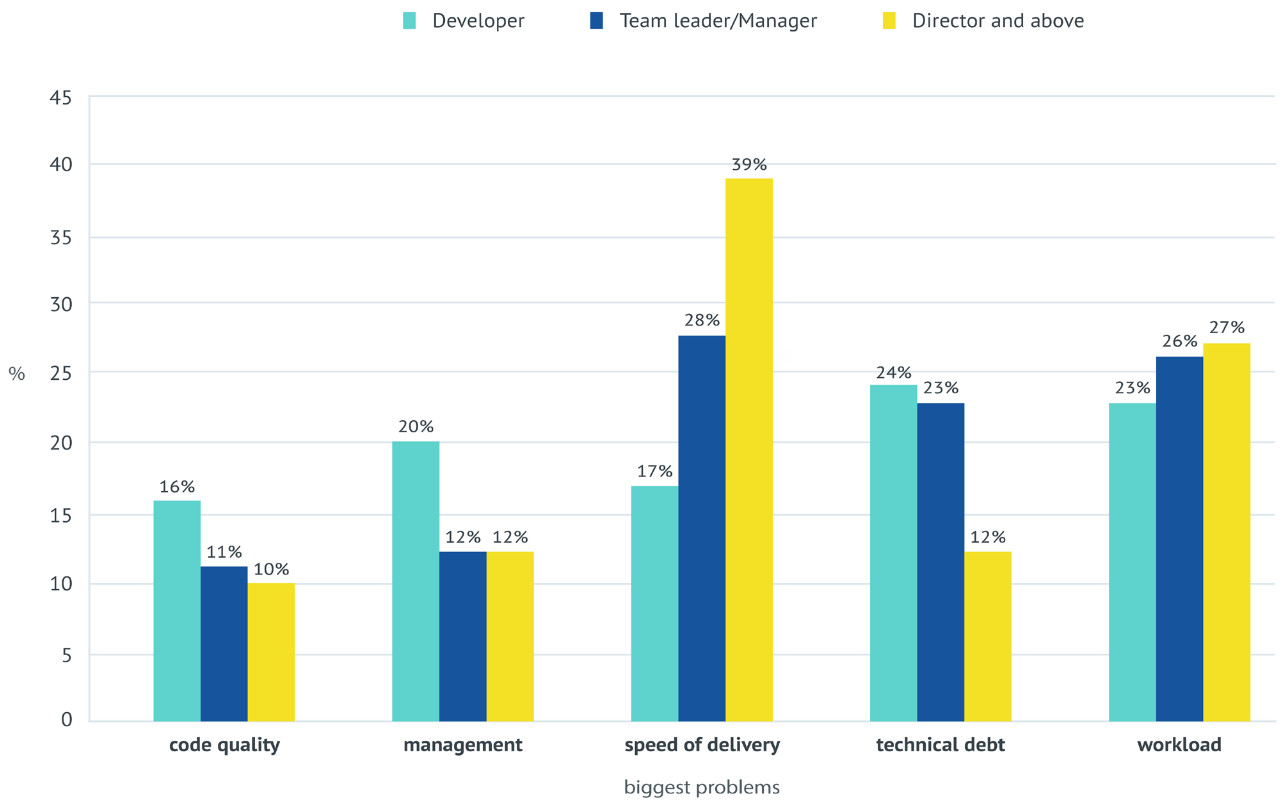
What concerns you the most? Whether you're a VP of Eng, Manager, Team leader, CTO or Chief Engineer you probably have a good idea of what the main issues your team and project are facing. But are the issues specific to your company? Company size? Industry?

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

The biggest problem in your project? It's all about perspective

As part of the survey we asked engineers and team leaders what they thought was the **biggest problem** in their project. Respondents were forced to pick only one issue to identify the most pressing one. The results are summarised in the below chart.

Biggest problem in your project



From the chart we can make several observations:

• If you're the VP of Eng or a team leader, then **speed of delivery** is your biggest issue, followed by workload.

• On the other hand if you're a developer, then **technical debt** is your biggest concern, followed by workload.

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.

This begs the question: are the incentives of team leaders and VP of Eng different from that of the software developers?

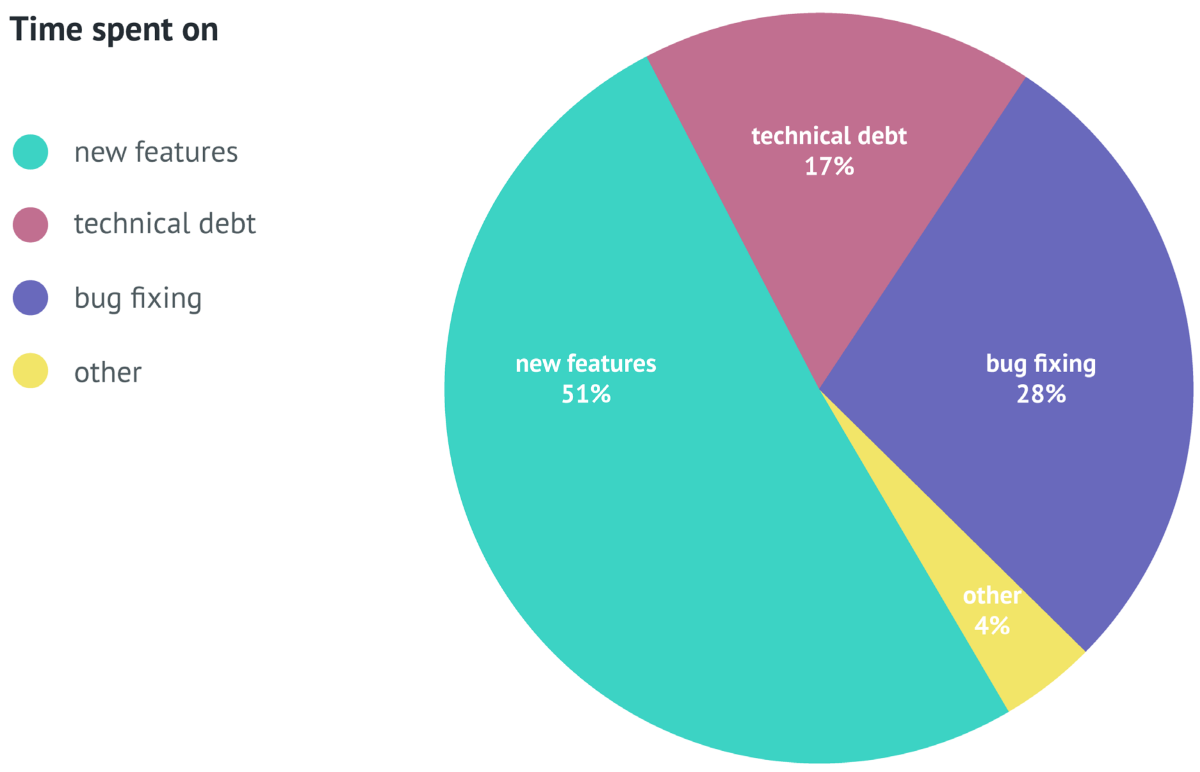
Not really.

What your team *actually* does

Technical debt and speed of delivery are two sides of the same coin: sooner rather than later, less technical debt means more time spent on productive work, therefore happier, more productive developers, and shorter time-to-delivery.

To confirm this observation, let's look at the data.

First, we asked developers how they split their time. The breakdown looks like this:



Despite the emphasis on speed of delivery, your team spends 28% of their time fixing bugs, and 17% of their time on technical debt. That is, they spend 45% of their time not building new features.

KEY INSIGHT 2
Developers spend 45% of their time fixing bugs or addressing technical debt, instead of building new features.

A word on controlling factors: In our research 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 whereas 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 may not still 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
The smaller or younger the company, the more likely it is to prioritise speed of delivery at the expense of technical debt. On the other hand, the larger and older the company, the greater the emphasis on technical debt.

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. Now, strictly speaking, this rating reflects the perception of code quality; it is a subjective evaluation that is highly dependent on the respondent's background and experience. The upshot is that two developers could hold very different views about the same snippet of code. However, our experience at Codacy dealing with thousands of developers on a daily basis shows that there is a strong correlation between developer perception of code quality and the actual quality of the code. The code quality rating is therefore, for the purposes of this guide, a good enough proxy about code quality.

Higher code quality should naturally lead to less time wasted on fixing bugs or tackling technical debt. The code quality rating should therefore 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 varies with the code quality score.

We found that when the code quality score increases by one point on a scale from 1 to 5, developers spend 5 percentage points less of their time on fixing bugs and 1 percentage point less on **fighting technical debt**. The direct consequence of that is that when the code quality score increases by one point on a scale from 1 to 5, developers spend 6 percentage points more of their time **building new features.**

To summarize:

KEY INSIGHT 4
Increased code quality results in less time spent fixing bugs or fighting technical debt and more time spent on building new features.

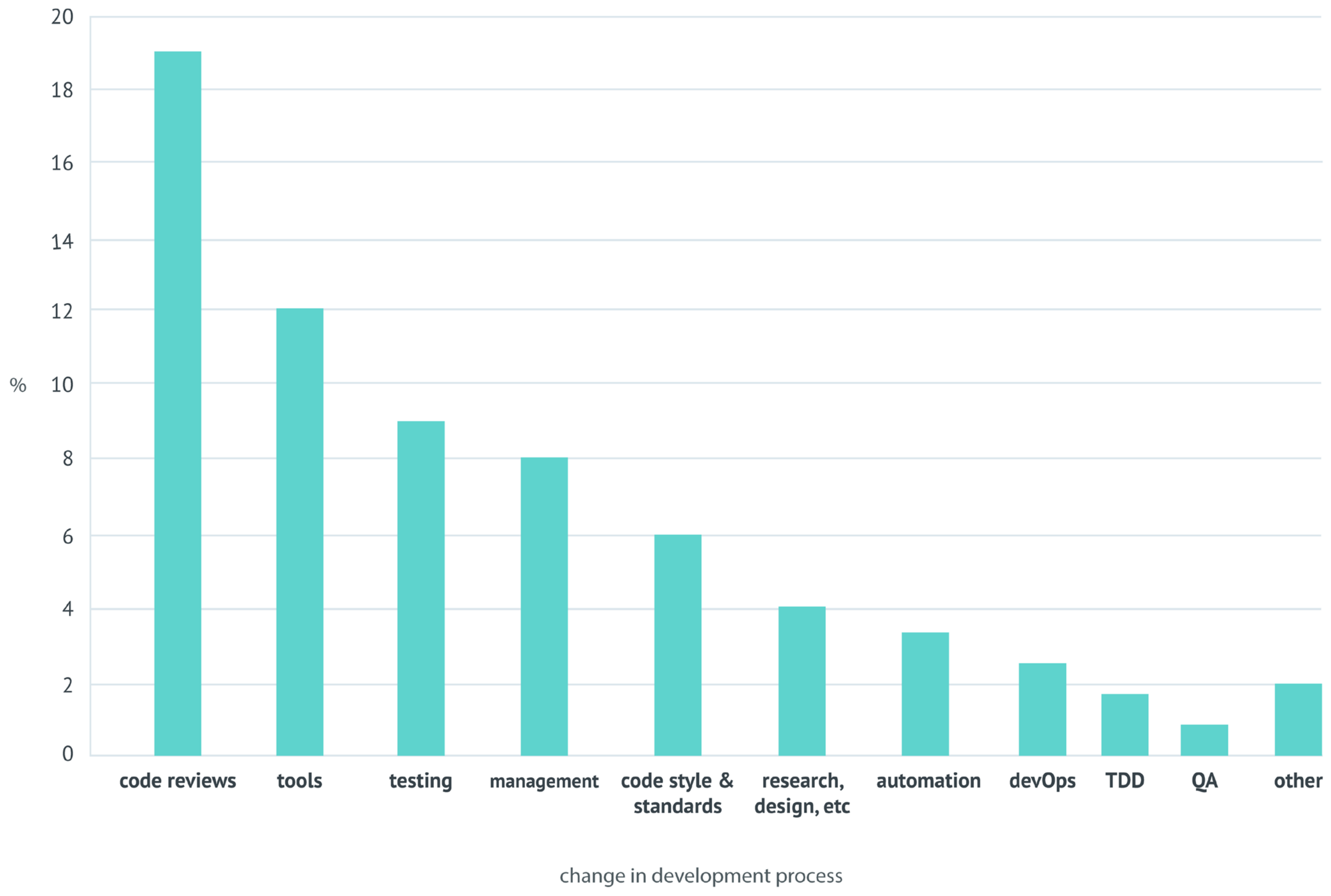
The single most effective way to improve code quality

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. Nowadays development cycles are measured in weeks, sometimes in days.

So you face the ultimate engineering challenge: balancing quality with speed of delivery. We have seen however that **your team spends, on average, 45% of their time on fixing bugs or tackling technical debt.** We have also seen that good code quality increases significantly the amount of time spent on building new features while reducing the time spent on non-productive work.

So to start off, we asked developers **what change in their development process had the biggest impact on code quality.** This was the result:

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



*The question was open-ended to avoid leading the respondents into specific answers.

As you can see from the above chart, code reviews are the number one development process that impacted code quality. This brings us to the next key insight:

KEY INSIGHT 5
Code reviews are the single most important thing you can do to increase code quality.

The above insight applied to all types of respondents, whether they worked in small, large, young or old companies. So let's dig deeper and explore how to get the most out of your code review process.

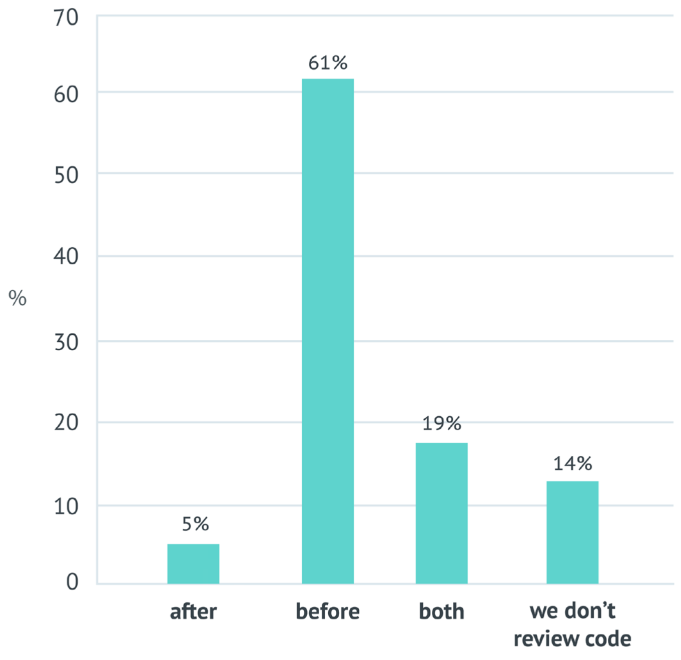
Five rules to set up a code review process

In our survey we quizzed our respondents about a number of code review rules. These were:

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

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

When it comes to which approach has the highest prevalence, there's little ambiguity, as the chart below shows:



code review before or after deployment

The real question, however, is "so what?"

To answer this, the next step is to investigate whether the different approaches have an impact on the developer's output. What we found is 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 doing code reviews before only; in fact it performed just as well as doing code reviews after.

For the sake of clarity, we've summarised the statistics in the table below (statistics for the "both before and after" and "only after" were grouped in 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%	47%	44%
% time spent fixing bugs	25%	30%	36%
Need more time to maintain code	60%	69%	76%
Perceived code quality score	**3.6**	**3.6**	**3.3**

This brings us to the following insights:

1) Those who did not do code reviews at all **performed worst** on all possible dimensions: they spent the **least amount of time on building new features**, they spent the **most time on fixing bugs,** they needed time to maintain code the most, and perceived the **quality of their code as the lowest.**

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

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 did the reviews before deployment. On the surface of it, this comes as surprise: after all you'd expect the quality of the code that has been reviewed twice (both before and after) to be of higher than that of the code reviewed only once. Our experience at Codacy indicates that this is not the case because companies that do code review both before and after are more likely to have loosely defined - and enforced - code review processes. In other words, this is the classic example of a "quality not quantity" situation: it's better to review the code thoroughly once than do it not-so thoroughly several times.

On the basis of the above insights, we are able to formulate the following rules:

RULE 1

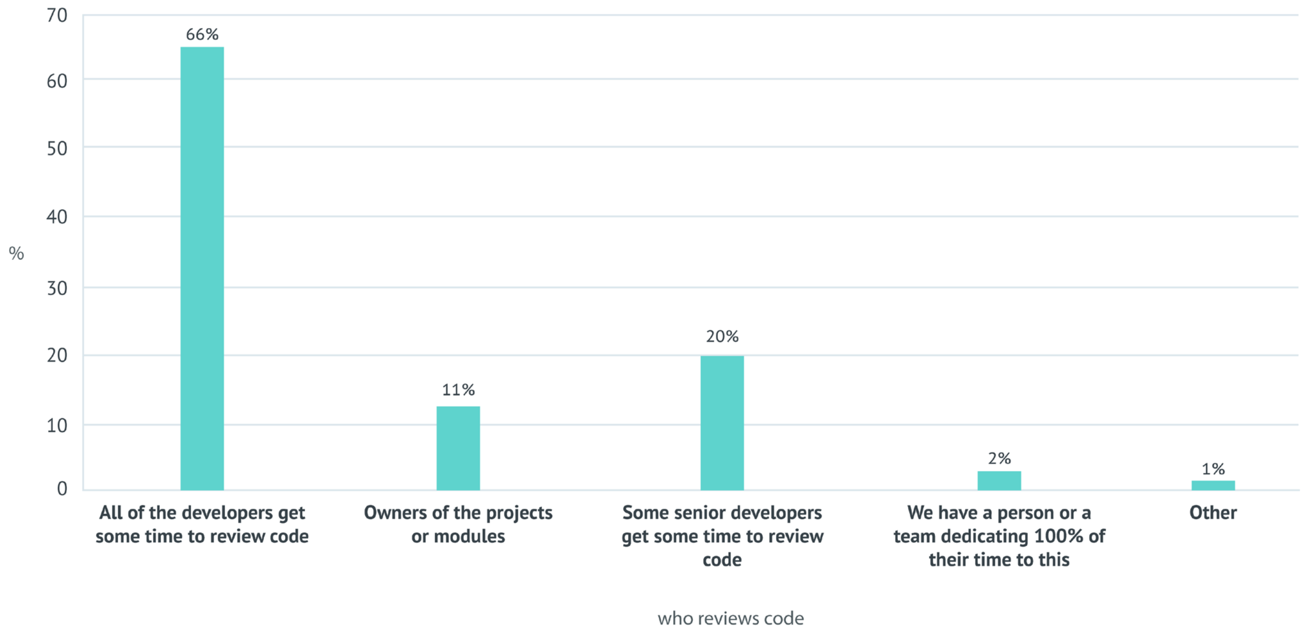
Don't skip code reviews. Your team will be less productive if you do. And review code **before deployment**. Not after.

Question 2. Who should review code?

The next question is designed to help us determine who in the team should do code reviews, with the aim of making your team as efficient as possible.

First, let us look at the possible answers to the question:
• All of the developers get some time to review code.
• Only the owners of the projects or modules review code.
• Some of the senior developers (but not the owners) get some time to review code.
• We have a person or a team dedicating 100% of their time to this.

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



By far the most common approach is having ***all*** **the developers** spend some time to review code, with 66% of respondents reporting that this was the approach taken in their company.

Now this is a surprise.

You would expect companies to task their most experienced developers to review code, essentially making sure that the young apprentices are mentored by the old, experienced druids of the village.

That's not the case, even accounting for factors such as company or team size, company age, programming language used, etc.

To find out why, let's look at the impact of each of the approaches 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 favourable 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	25%	30%	30%
% time spent on technical debt	16%	20%	16%
% time spent building new features	54%	49%	48%
Should have more time to maintain code?	58%	63%	70%

It seems that the approach which consists of having *all* of the developers review code is also the best approach: it is the option in which developers spend the most time doing productive work, i.e. building new features as opposed to fixing bugs or tackling technical debt. Those who follow this approach also feel less the need for extra time to maintain code.

A possible reason for this was best expressed by one of our respondents: *"The value of code reviews is not just in the feedback but the interaction between the team."*

This makes sense. A code review serves two purposes. In the **short term** it helps improve code quality. In the **long term** it serves the purpose of facilitating knowledge transfer between engineers, which means 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 the more battle-scarred 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:

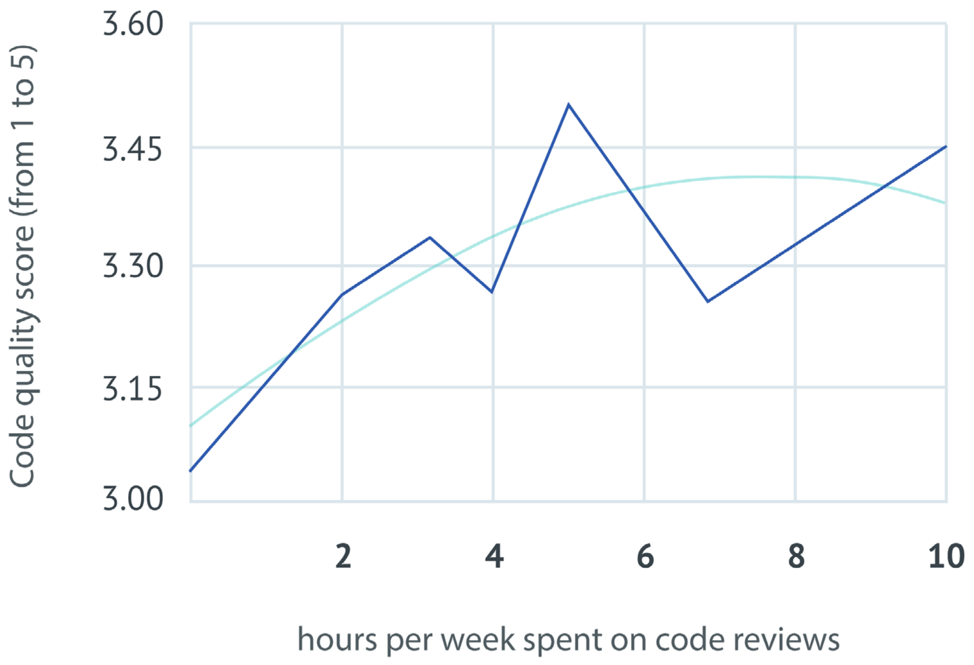
RULE 2

Make sure *all* your developers get to review code from time to time. This will bring a feeling of empowerment and improve team learning.

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.** So we asked our respondents and again we were surprised by the results.

On average, developers who did code reviews spent 4 hours per week doing so. The key point however was that the law of diminishing returns applied: the benefits -in terms of code quality - quickly increased initially. However, past a certain point (5 hours per week and per person), the perceived benefits cease to accrue.



This brings us to our third rule:

RULE 3

Make sure your developers spend between half a day to one day per week reviewing code. No more, no less.

Question 4: Should code reviews be blocking?

One question that you as a VP of Engineering might ponder is whether **it makes sense to make code reviews blocking** (i.e. only deploy lines of code/ commits/pull requests that have been reviewed) . The benefit in terms of code quality are obvious. You might be nervous, however, about the impact on your speed of delivery.

It turns out that for 72% of the respondents, code reviews are blocking. What's more, those **who said "yes" had a code quality score 10% higher** than those who said no.

So the outcome is pretty clear: 1) you will not be fired for making code reviews blocking. In fact, most people do. 2) In doing so you'll improve code quality, thereby reducing technical debt. What's not to like? 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?

How detailed and and comprehensive should your team be when reviewing code? Does that impact the ability to ship code or the perceived code quality? How pedantic should you be? 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 response with other variables. Here's the picture that emerged:

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 ok with deploying this") meant that the respondent would spend 31% 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 24% of their time fixing bugs.

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 43% of their time shipping new features. A high score of 5 meant that the respondent would spend 54% 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 greater code quality.**

So being more strict in code reviews correlates with clear benefits: less time wasted fixing bugs, more time on building new features, a perception of better code quality, etc. So it is undeniable that being more strict allows for companies to develop software better. The next 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, 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. This is what you are accountable for.

Maybe you worry that spending time on code reviews would slow things down.

In fact your team currently spends 45% of its time on technical debt and bug fixes. This is a necessary evil but there are opportunities for optimization that will quickly pay for themselves higher code quality translates directly 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 the code of your project.

Quality code means more productive time, and happy engineers.
So just remember the five simple rules for good code review practice.

Cheat Sheet

RULE 1

Do the code reviews before deployment. Your team will end up, on average, spending 7 percentage points% more of its time on building new features compared with those who do after, and 10 percentage points% more than those who don't do code reviews at all.

RULE 2

Make sure all your developers get to review code. This will improve the feeling of empowerment, facilitate knowledge transfer, and improve developer satisfaction and productivity.

RULE 3

The optimal amount of time to spend on code reviews is between 0.5 to 1 day per week per developer.

RULE 4

Make code reviews blocking, that is, don't deploy before they have been carried out.

RULE 5

Be strict and thorough when 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 the integration 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. You can also sign up for a free trial or request a demo.