6 Steps to Speed Up Developer Velocity

Getting Faster by Getting Smarter

Introduction

In 2020, Mckinsey & Company surveyed 450 enterprise-level companies to understand the impact of developer velocity on success. The results were clear and incontrovertible – developer teams that work faster lead to much better results across the board.

Companies that scored highest on Mckinsey's developer velocity index (DVI) - which tracked 50 parameters ranging from architecture to employee satisfaction – showed revenue growth four to five times higher than the average, created 60% more shareholder results, and produced 20% higher operating margins. That is not accounting for the 55% higher scores on innovation metrics.

This also matches our own findings. In 2022, we surveyed over 300 senior managers at enterprise-level companies and found that most spend significant time waiting for dev-critical tasks like builds to complete. However, teams that have the right environment, tools, and processes noted that they can save up to two hours per day (or 64 days a year) in time wasted.

McKinsey defines developer velocity as "the ability to drive transformative business performance through software development."

In simpler terms, developer velocity is the speed at which teams can reach their milestones and complete work in a given period.

For example, if we measured a three-week sprint where a team completed 14 milestones, we can (generally) assume that is a good baseline. Measured over time, you can build an average that provides some flexibility in how work is assigned, but which lets you maximize teams' efforts.

What do we mean by developing fast?

Measuring development speed sounds simple – just track how much gets done in a set period. But that simple definition ignores several distinct factors that impact velocity, and more importantly, that impact the results of said speed.

If a team hits 20 milestones in a two-week sprint, most people would nod in approval over a job well done. But what if those 20 milestones are full of shoddy code, cut corners, and buggy products? Or what if the developers get overworked and can't meet the next deadline? Does velocity still matter in that case?

There's also the issue that fast isn't necessarily good.

If your team is going too fast and hits a wall, the damage will be more severe than if there had been some restraint.

So perhaps it is better to talk about controlled velocity, or more moderation.

Understanding developer velocity requires that we define it properly. So, from the start, it is important to establish that developer velocity is not just about SPEED. More than the methodologies teams deploy to work faster, it is about creating an environment where work is done well and within timelines.

Developer velocity – a new definition

But before we get too abstract, let's settle on a definition (or definitions) of developer velocity. The problem is that velocity isn't a monolith term – different methodologies track speed in unique ways (even if the goal is the same). This is largely due to the way work is spread out and workflows managed. Let's explore a few of the most popular methodologies:

Agile

In this case, we call it "sprint velocity" and it's measured quite simply – find the average of completed user story points for the most recent sprints (usually three to five). This is a standard (and basic) measure, but it's good for planning future sprints in terms of work capacity. Even so, it doesn't escape the issues we've mentioned.

Scrum

Scrum uses two types of velocity measurements, each with their own issues. The first is actual velocity, and it is measured almost identically to agile velocity (average of the number of story points completed in the previous sprints). On the other hand, there is expected velocity – a forward-looking metric that, while moderately useful, can be quite tricky.

In this case, developers estimate how many user story points they can accomplish in a set of sprints. So, for example, a team might decide they could complete 150 user story points in three sprints.

In this case, their expected velocity would be 50 story points per sprint.

These measures are very straightforward, and at face value are a great way to plan and understand teams' bandwidth when it comes time to start a new project. But they can become problematic when we over-rely on them, or if we are not applying these measures correctly. To get the most out of them, let's first understand the most important question remaining – why do we need to measure developer velocity?

Why do we measure velocity?

The clearest answer is that we measure velocity to track developer productivity and maximize development time. Developer team leads need to understand how much work their team members can handle per sprint, how to track their success, and how to maximize teams' productivity.

But focusing too much on productivity alone can be detrimental to a team's long-term success. Velocity should never be your objective. When you're too concerned with how fast you can go, it's easy to forget that going fast only helps as far as it makes your team better. Thus, perhaps we're asking the wrong question. Instead, let's ask why you SHOULD measure developer velocity:

To understand realistic bandwidth for dev teams

It sounds contradictory, but there is a real value in understanding the healthy levels of productivity for a team. This allows longer-term planning and smarter sprints.

To find areas for improvement and solutions

Developer velocity is a fluid measure, and as such can provide important insights into how certain tasks or projects impact results. This can help teams better prepare and find the right tools and solutions to avoid pitfalls. mental health can lead to better results, faster.

To ensure teams aren't being overworked

The McKinsey survey we mentioned above noted that out of a basket of factors that impact velocity, psychological well-being was consistently rated as the highest or one of the highest. Finding the right balance of work and mental health can lead to better results, faster.

To fine-tune developer teams' workflows

measuring velocity is a rear-facing exercise. But it can help understand how minor changes in processes can impact overall work. Maybe it's a simple addition like a layer of code review or making some steps work in parallel. Measuring the real impact of these changes is incredibly valuable

The productivity trap

Before we continue talking about developer velocity, we should take a small digression to talk about a big problem – our overreliance and obsession with "productivity."

Taken on its own, stripped of context, productivity is a good thing. We want to make sure we spend our work time doing things that help us meet our goals and avoid wasting our energy. However, this desire is often used to mask practices and workflows that can become counterproductive quite quickly.

Our fixation on maximizing every second of our days is not new, though it's not entirely a biological imperative. Writing on the psychological impact of our productivity drive, Oliver Burkeman notes that our attention to productivity and spending our time wisely became an obsession as we industrialized. We needed to get more out of every resource and minimize how much we wasted.

While these types of fixations ebb and flow, our productivity drive has changed and mutated, but not necessarily decreased. This is problematic for developers because it creates a situation where productivity itself is the goal, which in turn means that anything less than peak efficiency is considered a failure.

And yet, we cannot just ignore productivity. It's a good (if incredibly basic) barometer of a team's efficiency, and it can help set priorities. If it's kept in proper context – as a short-term measure of **CURRENT** activity – then developer productivity (as a product of velocity) can be a helpful yard stick.

The trap lies in how productivity impacts the workflow when It's misused as a measure. As Burkeman noted,

"becoming more efficient just makes you more rushed, and trying to clear the decks simply makes them fill up again faster."

And it's true – when the goal is productivity and maximizing time, finishing a goal means another one will be right around the corner. Finishing more tasks signals that developers can do even more, so new work is just piled on.

This is the pitfall to avoid – productivity is a great measure if the work pace is kept constant. If we use it as a motivator to create MORE work, then no one can ever be productive, as they will always be chasing the ever-growing pile of tasks yet to do.

Balancing velocity and efficiency

So, to recap – developer velocity CAN be a good measure, but it can also be a bad one if not used carefully. Where does that leave us? At an understanding that a metric is not ever perfect, and it is as useful as we can make it be.

With that in mind, the goal should be to have teams work as efficiently as possible without overemphasizing efficiency as the goal.

That seems confusing, but it's really straightforward. Developer velocity isn't a forward-looking measure. It can help us plan ahead with rough estimates.

"Placing developer velocity in the proper context can reduce the pressure on teams to constantly meet KPIs that are unrealistic and whose reward is simply more work in the future."

Let's use an example to better understand how developer velocity can be useful (or harmful). You have a team of five developers who are in the middle of a new app project. You've settled on an agile methodology to make sure the work gets done but need to understand exactly how to spread out the work to meet your deadlines and go-to-market window.

After breaking down the brief, you set your first sprint goal of 60 story points completed based on previous work (this is a place where developer velocity could be useful, if used properly). However, after three weeks of sprints, you notice that actual velocity is closer to 45-50 story points per week.

Here is where the developer velocity dichotomy comes in. Your team is "underperforming" their previous developer velocity measure. What do you do? On one hand, if you're using velocity as a KPI, then your team is clearly not being productive. They need to do more to catch up.

On the other, you can say "well, it seems like something might be impacting our work." This might lead to exploring what's changed between the previous project and this one – whether that's new tools in the toolchain, a variation on established processes, fewer team members, or simply the pace of work being too high and simply correcting back to a more reasonable cadence.

So, what do you do? You can demand that your team become "more productive" and keep the pace of work at what is clearly an unachievable level, thus creating more problems, more stress, and fewer results. In this case, measuring developer velocity will work against your team. They'll be overworked, stressed out, and less likely to deliver good work.

Or, you could take the opposite route. Instead of castigating your team, you might notice that this project is more complex, that you're down a team member, or even that the processes that work in one case might not be ideal for a different project with its own scope. In this case, developer velocity is a good way to identify a problem and work to solve it. Maybe your team is feeling stressed with pressure to deliver too much without a chance to check for quality. In that case, adjusting your story points per sprint lower can reduce stress and lead to better work.

This leaves two questions. The first is when and how to use developer velocity, and you can find our answers on that above.

The second, and perhaps more practical, is how can you improve your team's developer velocity without slipping into negative territory?

How can you accelerate your development team?

So far, we have remained abstract about the issue, but let's get practical. If simply forcing your team to work faster won't get you results, then what can you do? Here are a few strategies that might help.

1. prioritize developer well-being

Going back to the McKinsey study we quoted above, the biggest factor that determines developer velocity in the company's multi-faceted metric is psychological safety.

Organizations that prioritize building safe, open environments where developers can experiment, fail, and learn are much likelier to have faster developers

One of the biggest tenets of psychological safety is a "shared belief that risk taking in the pursuit of innovative problem solving is permitted and protected."

But more than just the development side, understanding that developers are humans who may not always perform at peak efficiency for a multitude of reasons can help teams measure developer velocity more effectively.

This means creating times for teams to stop working, building processes, and finding tools that can reduce things like crunch and excessive overtime. Prioritizing developers' health also means that they'll be more motivated, more open to experimenting, and better suited to deliver great quality code on time.

2. Failing is good; failing fast is better

There's no such thing as perfect software. In fact, most software is the product of repeated failures. The difference is good software is the result of learning from these mistakes.

Whether it's a customer complaint or a build error or simply a comma that shouldn't be there, failing is good if the outcome is improvement. If you can create cycles that turn failures to positive outcomes faster, even better.

In product terms, even small steps like AB testing enable faster feedback on proposed features, changes, and updates. Methodologies like test-driven development and static code analysis let development teams find errors faster and fix them much earlier in the product lifecycle.

Shifting to smaller iterations and smaller story points lets teams find fast solutions and spend time iterating and improving them.

3. Build structures, not mud balls

It really doesn't matter what development and code management strategy you choose, having a defined structure and processes will help your team reduce friction and work faster. Nothing impedes your work like a giant ball of messy code and architecture. The key goal is maintaining conceptual integrity.

So when you're building your infrastructure, it's critical to consider several factors. Start with your code architecture, but also consider aspects like maintaining proper documentation and release notes, effective communication channels, and clear processes that are easy to replicate and iterate.

Having clearer code documentation means finding errors and bugs is much easier, while communicating effectively lets teams clearly identify issues, share ideas, and collaborate. Reduce the number of failure points and opportunities for messy processes, and you can make your team happier, and more effective.

4. Pay your technical debt

Like we mentioned above, perfect code is a pipe dream. No matter how well you build your team, how immaculate your architecture, and how voluminous your documentation, any code that you build for a long time will accumulate technical debt. A "quick fix" here, a feature built on obsolete architecture, a lack of testing for an update, and suddenly, your software is failing (and not in a good way).

The more technical debt you accrue, the more your team has to divert their efforts to solve it.

In turn, they'll have less time to do the work they should be doing – building new software.

Automating parts of your workflows – whether in testing, bug finding, or code analysis – can help reduce these issues down the line. Similarly, constantly auditing your processes and technology can help you keep your debt low.

5. keep your branches well pruned

Another side effect of long-running code is that you'll eventually have branches on branches for every part of your codebase. This isn't inherently bad – it lets teams work on issues in parallel and solve issues faster. But when these branches aren't effectively managed, dev teams will spend a lot more time dealing with code inflation and working around long branches that haven't been merged, or that shouldn't be open.

Having an effective branch management strategy lets your developers work faster by keeping the code more accessible and reducing the annoyance that comes with having to stalk through a forest of code branches.

One strategy is to keep changes small, effectively reducing the size of branches and making them more manageable and readable. For code branches you simply can't break down, finding a refactoring approach (such as adding new levels of abstraction) can help you reduce their size that doesn't impact code quality, but does make life easier.

6. Use KPIs the right way

It is hard to avoid KPIs, and they can be especially useful – in moderation. They can help understand development progress and delivery status, but they can also be red herrings.

Over-relying on KPIs (such as developer velocity) can make teams miss the forest for the trees. Hitting a KPI is a good motivator, but it is not – and should not be – your team's primary objective. We have already talked about velocity, so let's use another example: code coverage.

There's no doubt that code coverage is important, but when you're focusing to hard on meeting a specific goal, you can start making naïve tests that cover the code, but not any relevant or interesting scenarios.

Instead, focus on setting realistic and relevant goals based on the main objective of creating software and use KPIs as a tool, not an end.

Avoiding the pitfalls of going too fast

Developer velocity is an inescapable measure in our fast-paced development world. Always-on apps and software that is continuously being updated means teams rarely have time to breathe between projects. This is why developer velocity can be a great tool.

Understanding velocity can help teams know what issues cause slow downs, when certain team members might need a lighter workload, or even which processes are more conducive to good and fast work.

On the other side of the coin, relying too much on productivity metrics or focusing to heavily on meeting KPIs and not goals can result in slower work, or faster work and unhappier developers.

It's critical to find the right combination of work practices, environment, tools, and processes that can give teams the psychological safety, technical capacity, and effective workflows to really build amazing and (nearly) perfect software.

For more information:

incredibuild.com

Phone. +972-3-6912440 | Email. info@incredibuild.com