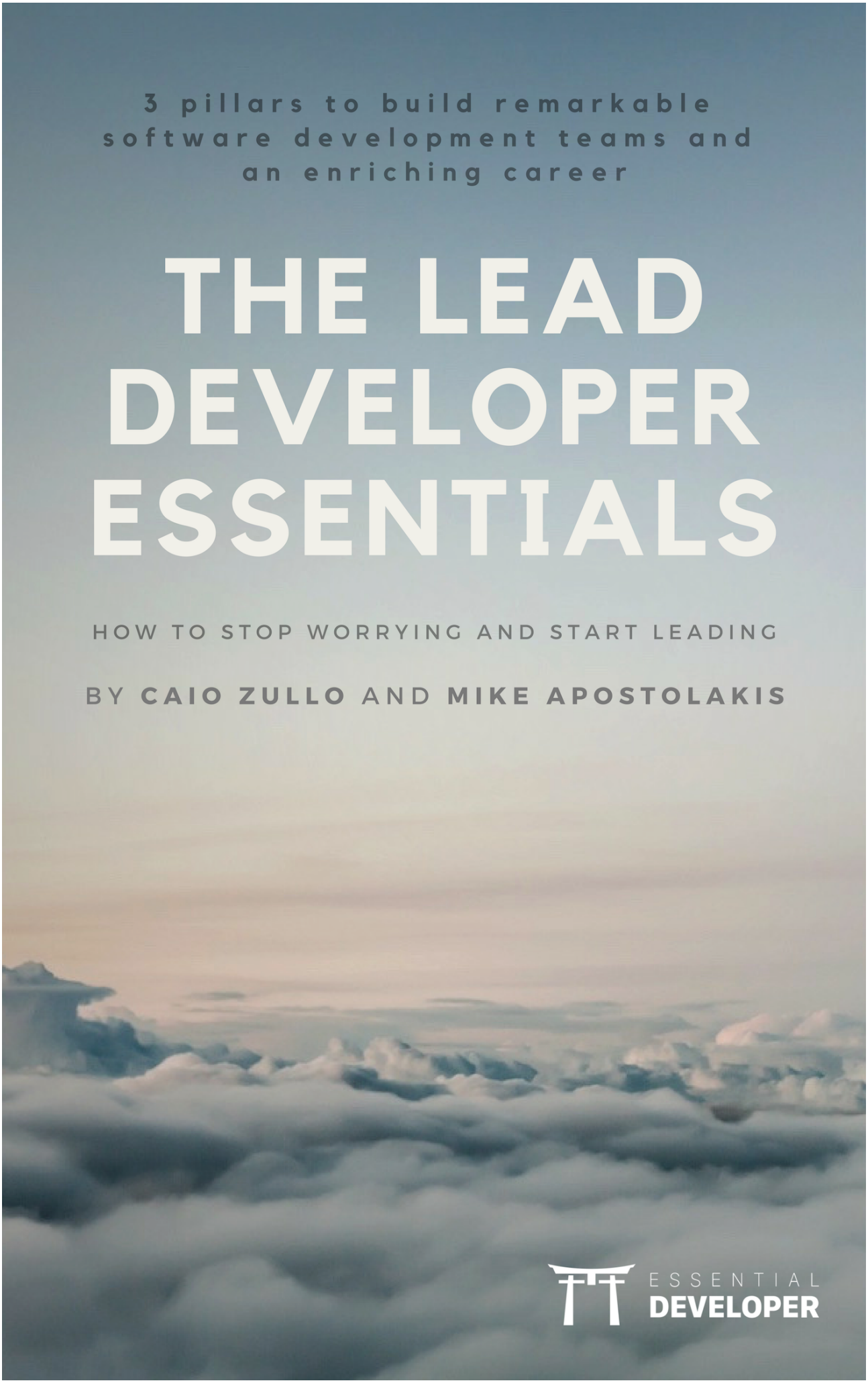
3 pillars to build remarkable software development teams and an enriching career

# THE LEAD DEVELOPER ESSENTIALS

HOW TO STOP WORRYING AND START LEADING

BY **CAIO ZULLO** AND **MIKE APOSTOLAKIS**



ESSENTIAL DEVELOPER

# COPYRIGHT NOTICE

# Table of Contents

# Introduction

# Author’s note: Caio

By the end of my first year as a software Team Lead, I had reached my stress limit. My personal life and my career were in total imbalance, and I was letting people down. It became clear that the path I had taken was utterly unsustainable. Most of my actions were reactive and added no measurable value to the overall business. There were no metrics I could use to quantify the impact of my work and I could not easily justify my decisions. Many times I was micromanaging my team to maintain their high quality and didn’t even have time to deliver my own features to deadline.

I had made a costly common mistake: assuming that my technical expertise would be more than enough for me to excel in this new role.

Almost no one was talking about the challenges faced by software developers who aspired to, and then attained, positions of leadership. It was hard to connect the dots and find the solution. There were no frameworks or Model-View-Whatever acronyms that could save me. I felt alone. My friends and colleagues were trying to justify what I was going through by claiming: “it's fine, there’s nothing you can do,” “that’s how things are,” “software is hard to get right” and “welcome to the business world.” Luckily for me, I didn't settle for any of those excuses.

My first thought was to go back to a senior position, where I could thrive again as a proficient professional. However, that was just running away from the problem.

Stepping down my position would make me financially stagnant. Additionally, the most interesting and worthwhile projects are dependent on collaboration, so working alone wasn’t going to get me anywhere near to professional satisfaction.

I went on a mission to improve my life significantly by reducing stress, going home on time, enjoying the holidays and continuing to increase my wage while doing what I loved—working on  worthwhile projects with remarkable people. But writing clean and good code was the minimum requirement and no one would pay me more just to do that. If I wanted to evolve as a professional and increase my income, I had to learn how to create and lead a highly productive team. I also had to learn how to maximize my team's output based on the available resources while managing risks.

In short, I had to find a sustainable way to make software development scalable.

This was the biggest career challenge I’ve ever taken on and ended up being what I love to do. After hundreds of hours spent researching and studying advanced software

development, leadership, finance, business, economics, marketing and psychology, it was now clear that just being a good programmer wasn't enough. I had to learn how to create solutions that would have a continuous positive impact on the business' goals—including when I was absent from the office!

As I went down the path of learning skills outside technical software development, I noticed that the content consumed by programmers and business people is drastically contrasting. The underlying message differs strikingly, which—in turn—makes both sides unable to work together optimally. Often, developers and businesses work towards goals that have no shared values. I had finally found an exciting and rewarding challenge: I had to bridge that gap between business and developers!

This book is the result of an eight-year investment through research and practice in an array of disciplines, given in a set of structured processes that otherwise can be costly to develop. What I want to present you with is the foundation to a prosperous mindset that will help you increase your income, create outstanding projects, collaborate with ease and maintain passion for your job. The foundation has three pillars: Empathy, Integrity and Economics.

As a young developer, I figured out Integrity and Economics, but failed miserably in the Empathy pillar. I thought I had reached the top (according to my own false metrics/goals), but I couldn't have been more wrong. I just set myself low goals and was afraid of jumping to the next level because it would mean facing a big challenge and probably being bad at it at first. My challenge was to build the Empathy pillar to support a fulfilling life. Throughout my career, I found many developers who lacked one, two, or all of these essential pillars, thus they had an imbalance that kept them from progressing at the speed they wanted. Time and time again I see developers fighting to climb the ladder and falling, as I did. Without a strong foundation, you can be working very hard, but maybe not very smart.

Does such a foundation work everywhere? No. We've seen many software companies that can profit by choosing a path of low-quality products where employees are easily replaceable. Low skill/quality/pay can be easier to scale. Our advice may not work in such places, but we don't want such a dead end job for you. Instead of accepting being a stagnant cog in the system, we want to help you build the skills so you can find/choose more prosperous, creative, and generous places to be.

If you want to achieve out-of-the-curve results in the software industry, I'm sure you'll find in this book the path to your independence, success and fulfillment.

## Author's note: Mike

I first started developing software in my early teens, but it wasn't until my early twenties that I decided to try out software development professionally. I started as a freelancer and soon transitioned to more complex projects that required more people. My goal was to improve and become more seasoned, so I tried my luck in London's job market, where I started working with companies that created global-scale mobile products. While in London, what fascinated me most was the opportunity to not only work in world-class environments but also to observe what these environments consisted of and what the dynamics were that made them world-class.

Unfortunately, having only studied the craft of software development didn't put me in a position to understand other essential non-technical aspects that made these environments so good. To prevent my research from becoming too broad outside the realm of software development, I decided to reverse my course of action to understand what doesn't make an environment world-class. In other words, what are the things my team and I should avoid if we want to create great products and achieve a sense of self-fulfillment?

My experiment seemed to work, since I started to correlate behaviors to certain outcomes. It became apparent the first ingredient that a successful team had was a sense of empathy from all its members. To identify other valuable (or lousy) traits, it was quite important for me to understand the decision-making process of my teammates. By observing non-technical behaviors of more seasoned engineers, I started noticing that some traits made developers progress much faster, while other traits systematically made developers unhappy and stagnant. By practising the former traits, I accelerated the rate of improvement of my career and surpassed developers who were years ahead of me both financially and hierarchically. One of the most unexpected positive side effects that came from learning and practising these traits was the ability to successfully coach and mentor other developers to a rewarding career path. By providing them with a mixture of technical and social fundamentals of software development, my students were able to obtain a high-performance career, often surpassing my achievements.

After I felt comfortable navigating through software engineering decision-making processes, I went on to study the business side, then economics as a whole. From a developer's point of view, it proved to be a lonely and tough journey in an ocean of unknowns. Despite the difficulties, my eagerness for learning from other disciplines helped me understand and regard software development from a whole new perspective.

With time, it became clear to me that economics is not only a fundamental part of the business mindset, but it should be an integral part of developers' daily operations as well. For example, I realized code is a much more complex idea than I previously

thought. I used to perceive code as text that sits in my screen, day in and day out, and waits for me to edit it. Now, instead, I see code as more than raw commands to a machine or a communication tool among developers. Code encapsulates the continuous input of team members to either contribute to a profitable (asset) or unsustainable (liability) software product for the firm.

As part of my contribution to this book, I have cherry-picked the essential topics I encountered in my studies over the past years and, along with Caio, I have tried to present them in the most straightforward and structured form possible.

I believe software is a vital part of our society and daily lives and, when approached as such an important activity, developing software can be a rewarding and fulfilling journey. My goal with this book is to share with you the traits of highly successful software developers and expand your imagination to expose aspects of your daily decision-making processes you might not even be aware exist. I certainly hope the knowledge you'll get from this book contributes to your journey of continuous growth and accompanies you in your pursuit of mastery.

## Current reality

Many software teams are dysfunctional. Perhaps you have noticed this as well, but first—before it becomes clear how beneficial it can be to fix this—let us explain what we believe the main causes of this are.

Developing successful commercial software is not the easy walk that many people in the industry think it is. Creating successful software products is a rewarding but complicated and long-lasting process that requires discipline and deep understanding of many fields—not just the latest framework or design pattern that is currently trending on Twitter.

The programmer, like many other roles, is valued based on experience and seniority. This meritocratic model can theoretically work well for regulated industries as professionals must follow specific rules and meet agreed standards to be able to perform and progress in their careers. However, when the demand for developers in an unregulated market like ours surpasses the available supply, such a model can lose its worth.

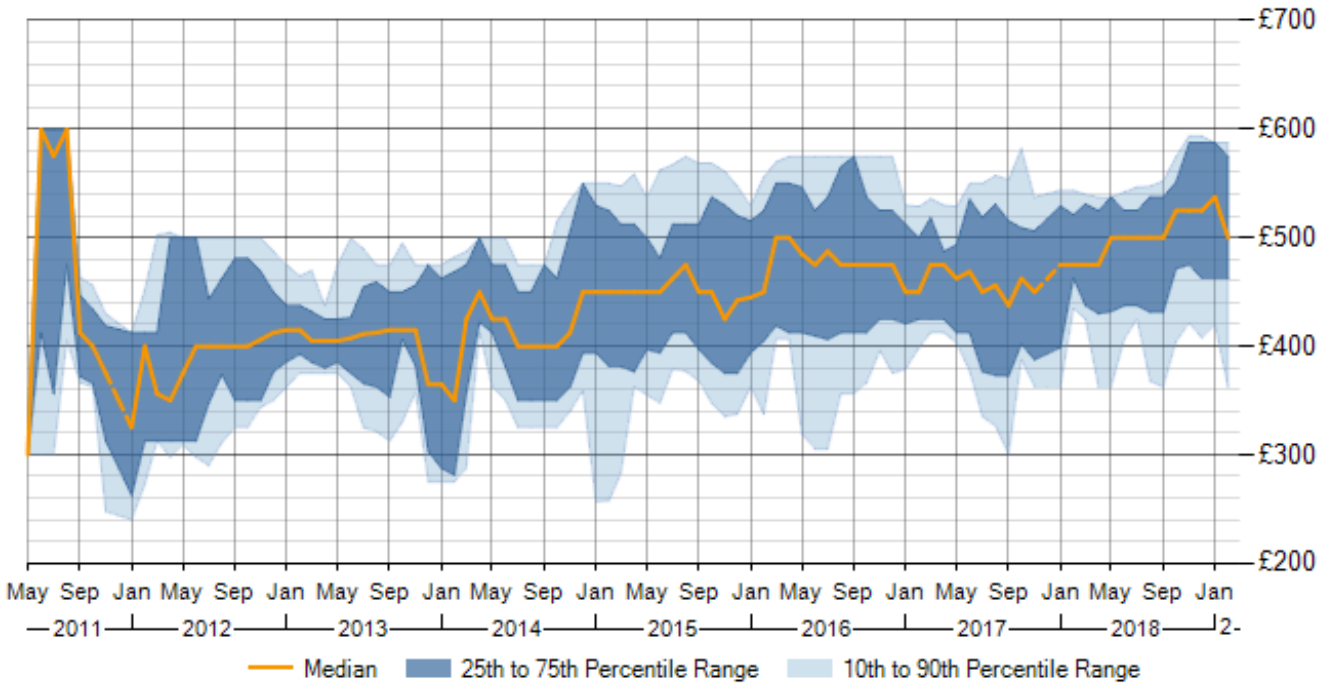
As companies rush to create highly competitive products, the way of selecting, evaluating and training candidates has lost its integrity. Companies depend on talented employees who can realize their vision, but because of the rising demand for talent and the shortage of supply, the screening and educational process in many cases has

naturally weakened. Because of this system-wide issue, companies seek outstanding leaders who are technically proficient and, at the same time, can build strong, functional teams.

## A quick look at the numbers

Over the years, salaries of software developers have had many ups and downs. Because of the high demand in an unregulated industry where everyone can quickly become a "senior," median salaries appear to have stagnated compared to the rising cost of living in most western markets. When the demand for experienced developers is high and the barrier to become a “senior developer" is low, the nominal compensation usually goes up in the beginning, but tends to go down with time as the supply relative to the demand increases as well. With low barriers, the title becomes meaningless. Can we even compare a senior with three years of experience to a senior with ten years of experience? It's a case of: "œIf everyone is a senior, no one is a senior," which makes it very hard for experienced developers to establish an edge.

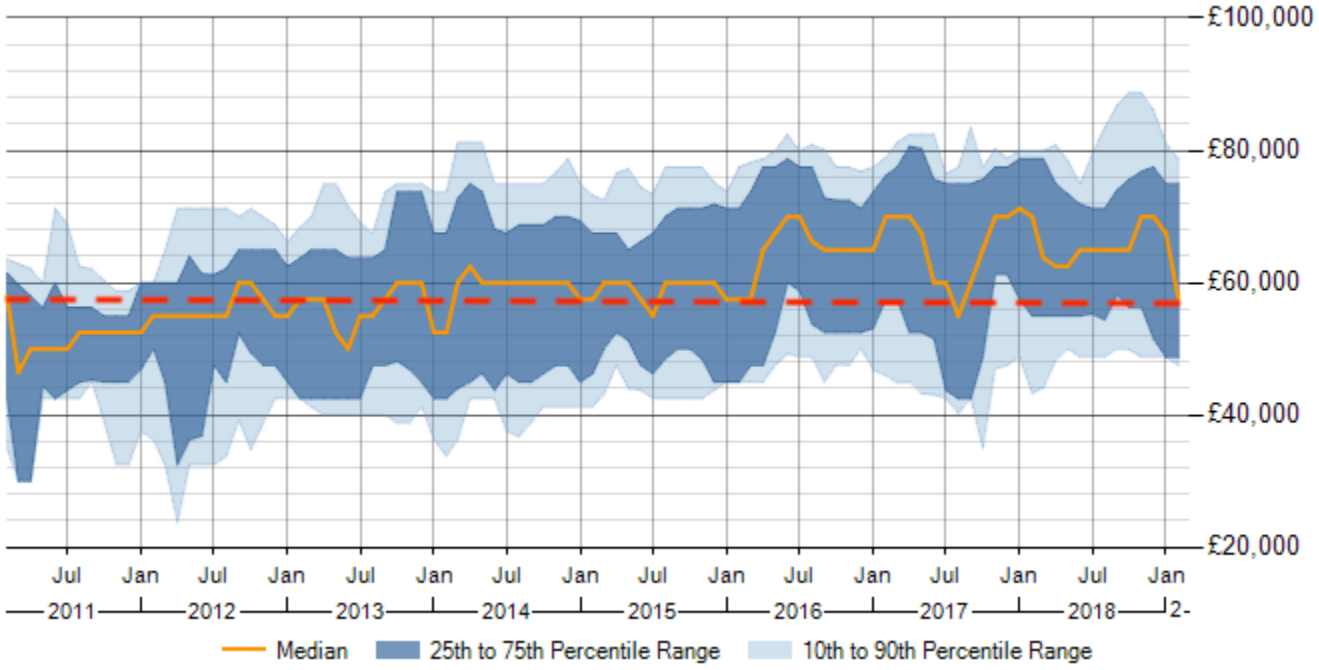
Let's have a look at the iOS industry, for example. The following charts show trends of iOS developer salaries, based on seniority, during a period in the UK (2011 to 2019). A similar movement can be observed in the US and other western markets. In the UK, the median daily rate of a Senior iOS Developer (contractor) during Q2 and Q3 of 2011 (high demand, low supply) surged to £600. It has dropped since to an average of around £450 until the beginning of 2019 (high demand, higher supply).




itjobswatch.co.uk source

During the same period, the annual salary of permanent Senior iOS Developers was

averaged at around £60,000.




[itjobswatch.co.uk source]

People who work in permanent roles with the title of "iOS Developer" had an average annual salary of approximately £50,000, and those with a title of "Junior iOS Developer" had an average annual salary of approximately £30,000.

In short, the beginner professional can **double** their first wage over the course of **24 months.** The mid-level professional also has a powerful financial incentive to progress, as the difference in pay to senior roles can be an astounding **40% compensation bump**.

These may seem like phenomenal returns—especially for someone who has just graduated from college and joined the workforce—but as you grow, *your definition of success also grows*. For example, if we consider the steady rise in the cost of living, tuition, mortgages, rent, healthcare and providing for family and the future, the stagnated Senior Developer compensations don't seem to generate the financial freedom many are looking for.

For seniors looking to step up in their career, it's not clear anymore what the next step should be. This is in contrast with some years ago when they were first employed and steady growth would naturally come with time.

Some common questions about their career future are:

- "I have to become a manager to progress?"

- "I be a proficient Lead Developer?"
- "I be a proficient Tech Lead?"
- "I be a proficient Team Lead?"
- "I be a proficient Principal Engineer?"
- "Will all these new responsibilities pay off?"

Manager, Lead and Principal Engineer are all leadership roles, something that many developers fear. However, it has become clear that the next step for a senior looking to progress is to step up and become a competent leader.

If you need more incentive, the top 10% of lead roles can make an astonishing **34% more than the average senior**, signifying a welcome £25k increase in annual income, at a minimum.

# Putting it all together

Now, imagine for a second the opportunities for the top 1% and what the reasons are for their outstanding performances.

Based on our research, **the issues that professional developers often label as "career problems" are primarily economic ones.** There is one simple question that can shed some light on and potentially **change their mindset:**

All things being equal, if all software developers have the same amount of days in their career, **how can someone with the same title profit so much more than another?**

By talking to recruiters, companies, high and low paid developers, and by conducting an extensive market analysis, we discovered that one of the answers to maximize the returns in the long term is to be able to provide maximum value to the environment the professional operates in.

Providing maximum value to maximize returns may sound obvious, but, surprisingly, *most developers don't invest in skills that will generate those returns*.

As we both did at the beginning of our careers, programmers tend to think that, as long as they keep improving technically, they will achieve a high standard of life and job security. Over time, we realized this passive strategy was putting our careers in jeopardy. Becoming technically proficient is extremely valuable for starting positions; however, as you grow to more senior roles and bear more responsibilities in order to meet expectations and provide long-term worth to the business, exceptional technical skills become the **minimum requirement**.

Instead of investing in acquiring the skills that will get them to the next level, many developers try to increase their salaries by changing jobs frequently, typically every six to 12 months, assuming that a richer CV will bring higher returns. Another popular option is to go into private contracting. Although the daily rates may initially seem more attractive than permanent wages, when you account for hidden costs and unexpected employment complexities, this option proves to have its downsides as well.

Those actions do generate improvements, but on a micro scale. In this book, you'll learn how, by acting only on these micro improvements, *many developers are missing out on the real opportunities out there*.

Luckily, whenever there's a problem there's an opportunity. There is a plethora of ways to build a profitable and flourishing professional life. Companies are willing to pay the right talent unimaginably high rates. If you're thinking of a number right now, it's probably off by half or more—these offers don't get delivered to your inbox.

When we decided to establish a new and improved growth mindset, we started exploring and learning proven methodologies from other disciplines, such as psychology, management, marketing, finance, business and economics. The combination of those disciplines helped us better understand the daily challenges we are faced with, and how to **lead** the effort to achieve success both individually and as a team. Although the disciplines we've studied for this book can be very different, we identified three main pillars that support a prosperous and fulfilling path in all of them: Empathy, Integrity and Economics.

## Empathy

Although both businesses and developers have similar motivations, such as financial prosperity and growth, there seems to be a lack of understanding on how both parties can benefit fully from their collaboration, resulting in a mutual loss of value. At the same time, anyone who is willing to better understand the incentives and problems of the other side is looking at an excellent opportunity and the foundation of a more rewarding collaboration.

## Integrity

If being technically proficient is the minimum for senior developers, how can we reach the next level and succeed in the long term? Integrity is the foundation when it comes to providing outstanding value in the industry and creating a successful portfolio with provable (and reproducible) results. Integrity is built from a strong work ethic and

professionalism—qualities and principles that we never compromise for quick conveniences.

We believe that to achieve a level of high performance as a leader, and to be unified as a team full of people holding the same values, we must first be a role model for the whole team. Every action we take can influence the collective outcome. At the same time, the collective actions will affect our individual outcomes, so, when possible, we must carefully choose to work with people who have the same set of shared principles.

## Economics

Many professionals fail to connect their daily activities with the relationships and underlying mechanics involved in the economy. A number of developers have a static (and wrong) perception of their role. They believe their job is "just to write code." In reality, developers have a much bigger part to play in the economics of software development, as their work has a direct correlation with growth, product success and the sustainability of businesses.

When developers focus solely on coding, they miss the opportunity to cultivate other valuable skills. For example: understanding and contributing to the product backlog and road map, setting short-term and long-term goals in parallel to the vision of the business and forming more valuable cross-team communication channels to promote transparency. And who can blame them? Job descriptions don't usually include these kinds of processes.

Throughout this book, you'll learn how to connect all the pieces together and establish unique, valuable skills. By building up a strong foundation in economics, you can become an intelligent risk manager and recognize how your decisions can increase the probability of higher returns.

## The goal of this book

The goal of this book is to get you adequately prepared to highlight and position yourself to take advantage of current and future professional opportunities.

Professionals with a strong foundation of Empathy, Integrity and Economics will position themselves as an **essential element** in the software industry. They can recognize correlations between the procedures they use and their impact on the team's performance. They understand how the whole system works—from investors and executives to product owners and software developers, as well as the market they operate in and the economy as a whole.

We firmly believe that instead of making arbitrary career choices, you should proceed as if every step is an investment. In this book, you'll learn a strong foundation of how the job market works, how to identify opportunities and how to significantly contribute to the growth and sustainability of businesses. The goal is for you to break free of salary caps and stop being dependent on market trends to achieve a prosperous and fulfilling professional life.

Throughout the book, we'll use the terms Tech Lead, Team Lead, Lead Developer and Lead interchangeably. What we mean by those titles is a professional with very strong technical background, holding a software development leadership position.

Every company has their own description and hierarchy of titles, so there's no point debating them. For example, many "Lead" job titles just refer to "the best developer in the team." For us, leadership holds much more responsibility and rewards than just being the best developer around (although you may often be the best developer in the team).

Also, having a leadership title does not automatically make someone a leader. To be a leader, you just need followers. Thus, you don't need to wait to be awarded a *lead* title to start adopting the recommendations described in this book. The first step to achieving professional fulfillment and freeing yourself from salary caps is to stop chasing the status of made-up titles and focus on skills you need to acquire, new habits you need to form, and actions you must start taking. The sooner you start, the faster you'll achieve your goals.

**The most ambitious opportunities will present themselves to those who achieve a good mix of technical excellence and leadership.**

Part One

# Economics and Professional Software Development

# Introduction

In this first part of the book you'll learn the connection between economics and professional software development.

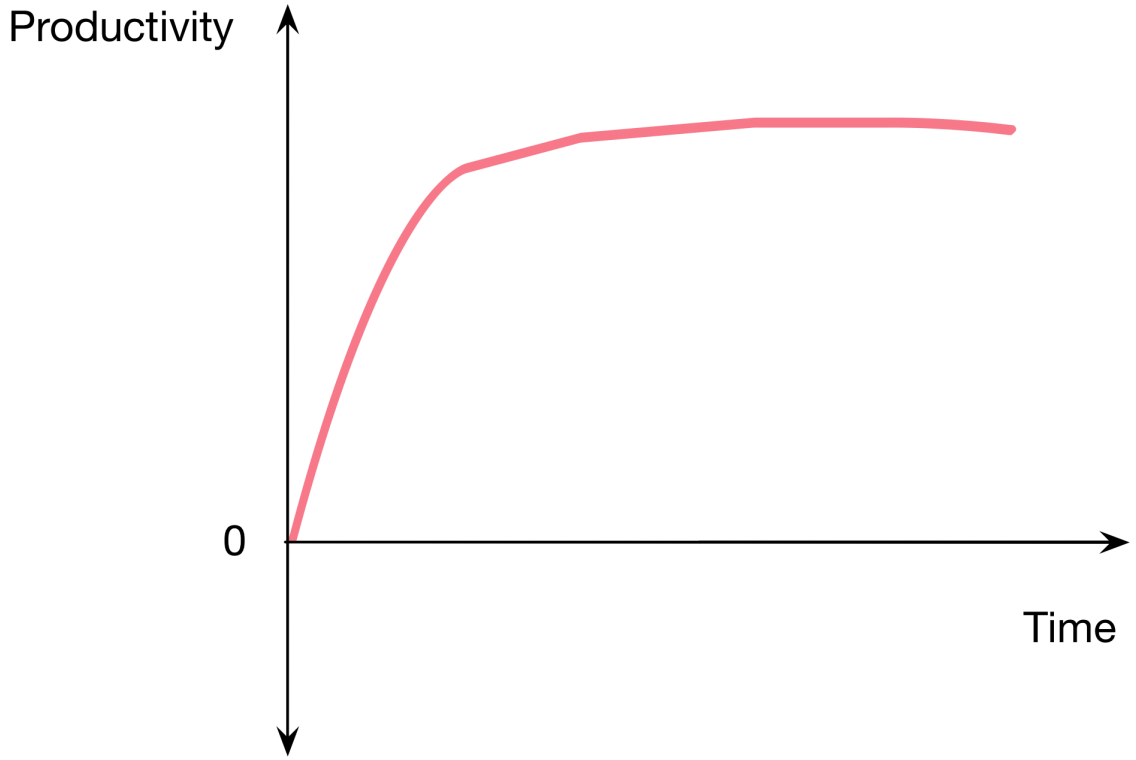
If you are part of a small or large operation that seeks profits and looks to achieve great, bold and audacious results, then it's highly likely you need to understand how economics dictates the underlying mechanics of professional software development and team operations.

The foundation for forming and putting into practise a mindset for realizing ambitious goals is built upon the **notions of top quality and sustainability**. Such an endeavor is challenging and doesn't come overnight. It requires commitment and continuous improvement over time.

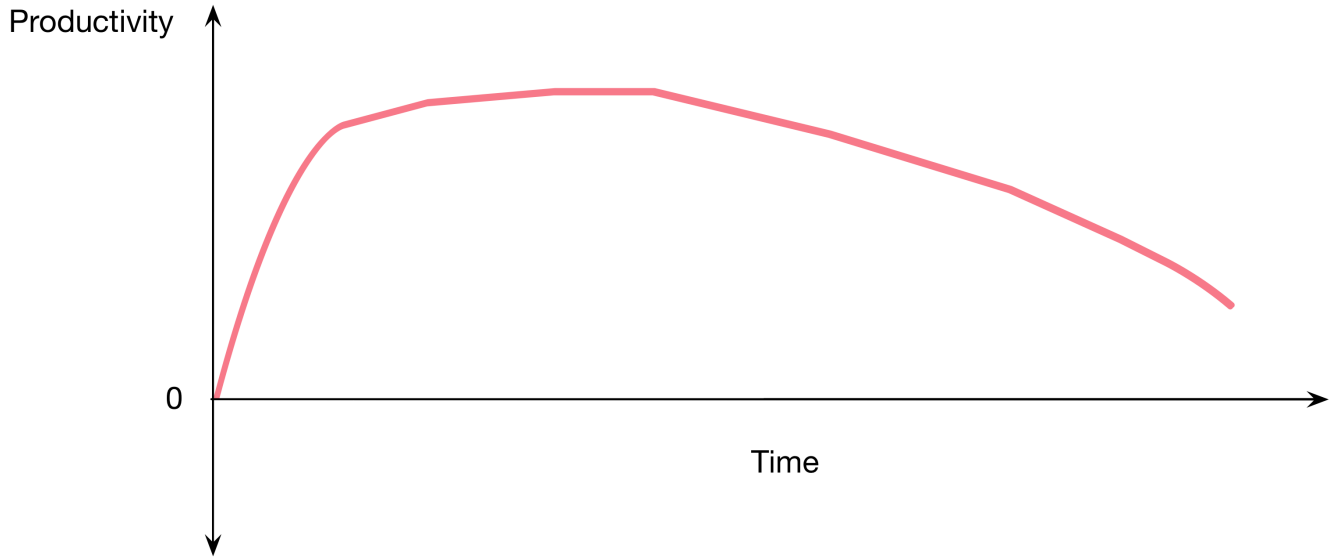
# Observations and habits of software teams

Sustainability is not just a tough thing to achieve, it's also hard to measure, predict and maintain. At a minimum, it requires forward thinking, care and discipline, which are extremely rare attributes for a software developer to master.

When a product idea is first brought up and developers try to implement it, they have no preconception of what they need to worry about. It's like starting with a blank canvas. Once they assemble the first parts of the product, the work they put in creates the impression of surging work rate, simply because they started with nothing and, after a small period of time such as a two-week sprint, they have *something*. Thus, if we were to graph their productivity over time we could end up with the following diagram.




As new features are being implemented, productivity continues to fly high. However, we've seen countless times a shift and a decline in the production value once the software needs new features that either require developers to touch parts of existing code or alter the behavior of the existing features. We can graph this decline as such.




In these cases, it is not uncommon for management to initiate a code freeze aiming for developers to magically improve the codebase and make it more maintainable for development to continue.

There are two significant problems with this approach.

## The lousy shortcuts strike back

First, the agents initiating the code freeze expect the development team to fix the issue, looking at it from a binary point of view, i.e., the problem right now can either exist or it can be fixed and cease to exist. It is much easier to think like this when we don't account for time. If we do account for time, it becomes evident that the problem did not appear in one single instance, rather through a series of decisions that occurred over a set period. Code freezes are one example of the negative outcomes that arise from a lack of sustainability and usually result in a frustrated team with low morale. The developers need to dig into theirs—and others'—code to try to make sense of what the problems are when they would much rather be working on new features without being dependent on old code. Moreover, it is a sign of things going in the wrong direction for the codebase, which leaks its negative outcome to the product and can then damage both sales and the business' future goals.

## The productivity fallacy

The second obvious problem is that, from the product's point of view, productivity has stopped, even though developers might be working much harder than they were at the beginning.

That's the problem of measuring and admiring "working hard" alone. Imagine the following scenarios:

- Dev team one works 10-12 hours/day Monday to Saturday and delivers a new product in three months.
- Dev team two works seven hours/day Monday to Friday and delivers the same product in three months.

Who worked *harder*? Dev team one worked many more hours. Now, in which team would you prefer to be?

Dev team one can no longer implement the product's vision in optimal time and have to work extra, which is completely unsustainable (especially for the individuals). If the business vision can't be realized, revenues may be affected so the team have to stop production and waste time re-establishing the conditions for getting back on track.

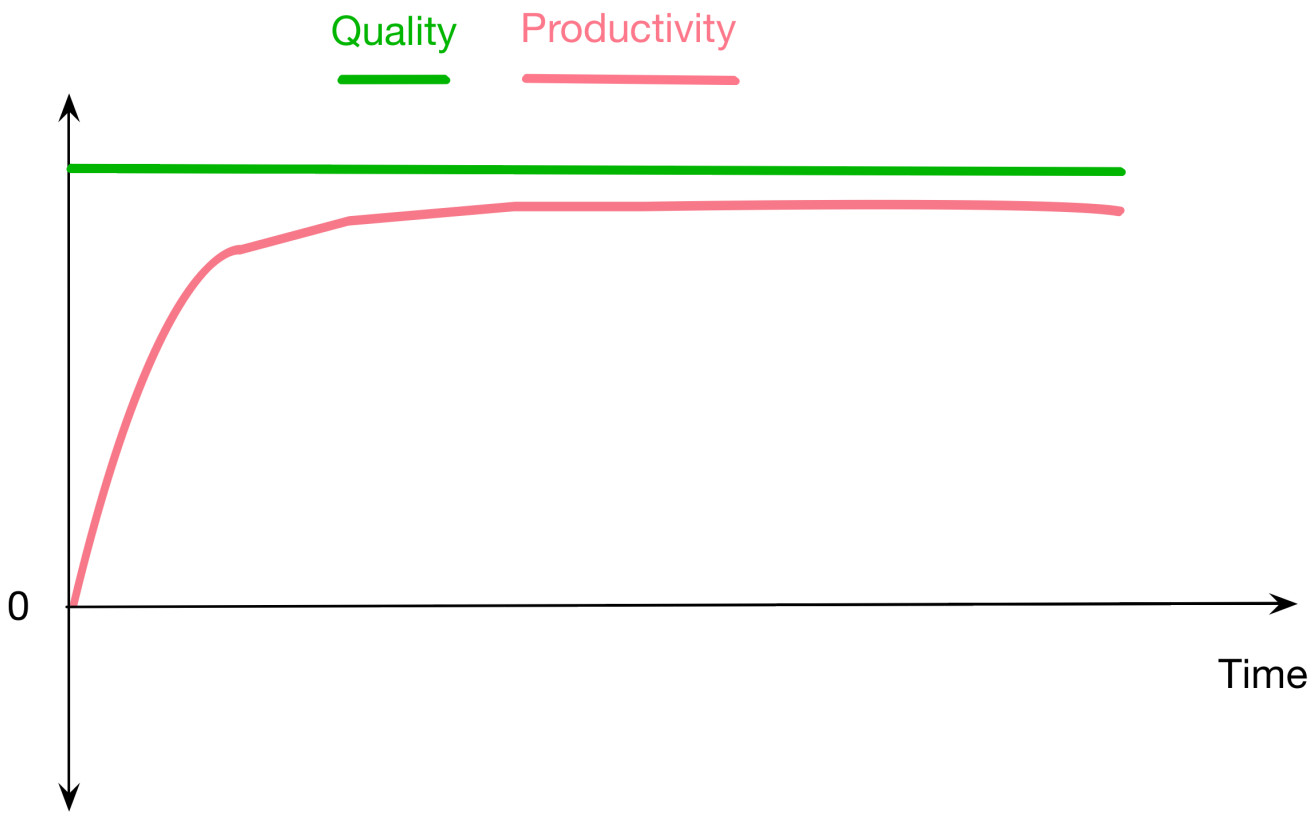
Another counterproductive result that can occur from low productivity and an unmaintainable codebase is having to rewrite it, in part or in whole. In such cases, the codebase has become a liability that isn't worth maintaining. This can be a catastrophic event for a business, especially if it financially depends on a product, as it may not be

able to generate enough revenue to keep its operations going.

As Team Lead you must always be aware of the health status of a codebase and consult the management appropriately about what it can and cannot withstand. One of the mistakes you must avoid is falling victim to the **sunk cost fallacy**. An unmaintainable codebase, from the point of view of the business, is a sunk cost, meaning **it has already occurred and can't be recovered**. Trying to rescue a sunk codebase will just generate more costs with no positive returns. As we'll see moving forward, this is the case for the team as well, because an unmaintainable codebase will only bring annoyance and low spirits, which will become a burden for the team and the rest of the company.

> "When software doesn't have a clean design, developers dread even looking at the existing mess, much less making a change that could aggravate the tangle or break something through an unforeseen dependency. In any but the smallest systems, this fragility places a ceiling on the richness of behavior it is feasible to build. It stops refactoring and iterative refinement. To have a project accelerate as development proceeds rather than get weighed down by its own legacy demands a design that is a pleasure to work with, inviting to change."
>
> —Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*

What we can learn from all this is that productivity over time, without measuring the quality of the work, is not such a good metric. It may seem tempting to set it as a leading indicator, especially when a product team is dependent on your deliverables; however, it doesn't show you the big and, most importantly, real picture. Think of it as measuring the GDP of a country or the revenues of a company without accounting for any liabilities, such as their debt.




The foundation of sustainability lies on quality. The goal shouldn't be to build software as fast as possible, but to build it in such a way that you will be able to build the firm's vision in perpetuity. When you master the art of building sustainable software with the firm's short- and long-term goals in mind, you're truly developing it as fast as possible. Any other measure of *speed* is a race to the bottom.

It can be hard to grasp the idea of always delivering top-quality software. In a professional environment, it sounds like the logical thing to do, and we often see developers and managers promise top-quality results to their superiors. However, if you've spent even a little time in a development team, you may have witnessed how such promises fade away with time or pressure from deadlines. We believe one of the main reasons for this phenomenon is that "sustainability" and "top-quality" are ambiguous terms. They paint a picture of a future that everyone wants, without giving hints of a path to get there—not even a snapshot of how a “sustainable” and “top-quality” outcome would look.

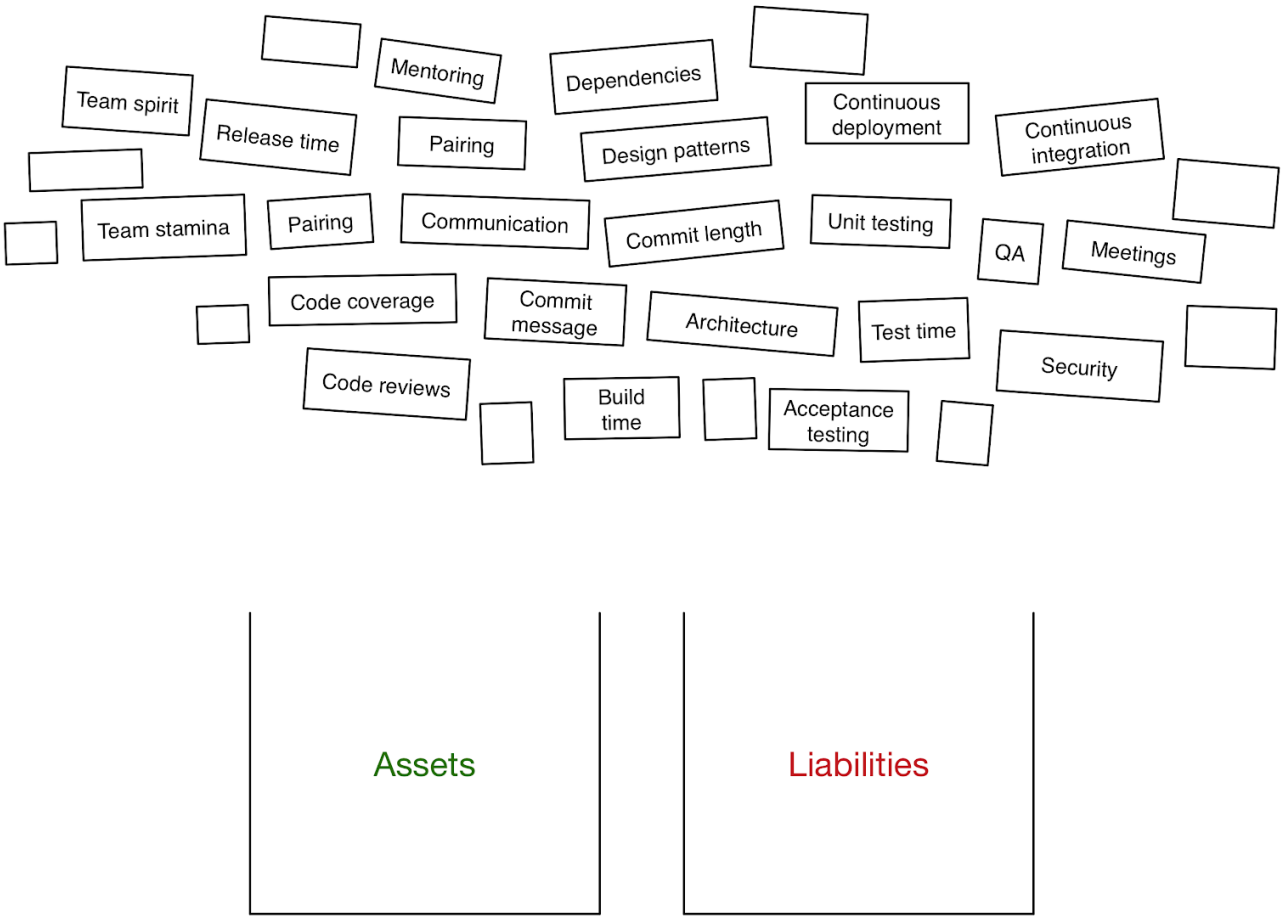
Instead of trying to grasp an unclear and way too distant future, developers can apply “sustainability“ and ”top-quality” in small contributions. Think of it this way: to create a sustainable codebase, you must first build the codebase. To build the codebase, you must write the code. To write the code, you should split your contribution into several commits. Thus, a sequence of high-quality commits produces a high-quality codebase. With this mindset, future outcomes becomes easier to manage, as we can focus on tangible actions we can take in the present.

The result is a byproduct of each choice and action we've taken to get there. So, instead of focusing on the big picture, we must get the smallest of contributions right first. It is unlikely that a team will end up with a profitable software product if the majority of its units aren't developed with profitability in mind. If quality rules apply to the commits of a codebase, then quality rules apply to all processes related to commits, such as pair programming, peer review, merge requests, branching strategy and continuous integration.

## Assets and liabilities

You can go a step further and try to frame sustainability as a game. As in every game, some rules apply and, based on your choices, you are rewarded or deprived of gains. The rules in this case are quite simple: you are called to manage a set of resources for a company. You must assess the risk of each so that they either become an *asset* (something that leads to profit) or a *liability* (something that leads to losses).

When you place a resource in the **Assets** bucket, your contribution is much more likely to meet the expectations of the organization and the customers. Unlike assets, when placing a resource in the **Liabilities** bucket, you should expect a turn for the worse either short or long term.




Because of the multivariate nature of the game (collaborators, time, company politics, the economy and others), any resource can move to either of the buckets, meaning their place is not fixed.

As you cultivate a substantial assets bucket, expect to reach closer to a sustainable future for the company and the tech team. Anticipate feeling self-fulfilled and growing professionally and financially. On the contrary, you don't want to find yourself with a bucket full of liabilities in this game. It's hard to come back from there, and you'll find your career goals harder to achieve. Your professional growth will stagnate pretty quickly, and your financial life could very well start to depend on economic trends and business cycles which, again, are out of your control. It will leave you with less autonomy and independence, and—most importantly—less freedom.

So, there you have it. That's the game. Beware though: you shouldn't trade lightly with this gamified view of things. Once you realize what is at stake, you can start accepting the responsibility for your choices and go after the rewards. Perhaps think twice before performing arbitrary actions from now on. The stakes are, and always have been, high as they affect the careers of your colleagues, other people who depend on your colleagues' output, countless customers and—of course—other people's capital. Whether or not you accept your responsibilities, the stakes are the same. So establishing a good plan to maximize assets will boost your success rate.

# The opportunity cost of ignoring Economics

Technology and software development are already overwhelming disciplines, so many developers don’t show interest in the (apparently) unrelated term "economy." Diving into a new unknown and complex subject can be intimidating. It’s natural to avoid a subject we don't understand when it’s not clear how our lives are so tightly coupled with it. The problem is: whether we understand it or not, economic developments still affect us. For example, you may not know that in a salary negotiation, you should account for future economic trends such as rising interest rates and inflation, fiscal and foreign policy changes and projected growth. For instance, $100,000 earned now may not be able to buy you the same goods and services in six months.

Few other subjects can so profoundly affect our lives as economics does, and that's why we believe it is such an important topic to understand. In fact, we deem it so highly that it became one of the three pillars that form the foundation of our principles.

# Software development through an economic lens

Another reason we have included an economics segment to this book is: economics dominate in the business world. Everything we do as professional software developers can translate into economic value for us and those who are dependent on us. When we say everything, we don't exaggerate: every single commit, every line of code, everything we touch in the codebase, any little configuration we change, they all bear risk that may affect the firm's revenue and, consequently, the team and yourself. Developers who don't understand the importance of economics in a software development workflow will most likely limit their professional growth.

## Software as the first dependency

In a business developing a product or a service backed by software, the software developers are the first dependencies. By first dependency, we don't mean they are the most important part of the business, rather they are an essential part because the product depends on their output. As the business grows, the dependencies multiply and other roles become essential, or perhaps even more important than the developers. A company can start with a single person bearing an idea for a product who has the required capital (even if this is zero). However, if that person doesn't possess the knowledge for implementing the idea, they will need to depend on someone who does. This person is the first dependency for the operation.

For larger operations, there are more resources required and a plethora of talents and

responsibilities to consider. To create these roles, the company founders rely on external capital. We can start seeing more dependencies appearing for this operation (e.g., sales and investors). However, we should also mention that the developers have a major role in the final say in the product. The developers are the ones who know **exactly** how the product works, as they are the ones who built it. The product doesn't do what the business wants, it does what the developers coded it to do. Thus, developers carry a big chunk of responsibility for the expected behavior of the product to match what the business envisioned. We, as software developers, are not only responsible for the code we write, but also for the impact our code will have on ourselves and others depending on us within the company (peers) and outside it (family, friends, customers, markets and society in general).

But if the impact of our code can affect the company's revenue, does that make us responsible for the company's finances as well? Not quite, but it certainly increases the stakes for us. As mentioned in the introduction, we can only be responsible for what we can control. In this case, we control the codebase, the build and delivery, the team and communication with other teams. Instead of managing the company's financials, we see ourselves managing risk. Simply put, risk is uncertainty for the outcome. Uncertainty that we could—accidentally or on purpose—introduce to the codebase which could then translate into a high probability of faulty behaviors. This could negatively influence the sales and revenue of the firm, increase the cost of maintenance and the marginal cost of adding new features and, of course, damage our psychological and financial states. When you understand and accept what's at stake, it's easier to find and take more assertive actions towards a profitable outcome. Profitable, measurable and reproducible actions, when made consciously, will make you an essential part of the business' success, which should grant you accelerated professional and financial growth.

## Managing risk in the codebase

We like to use a simple analogy to demonstrate how your actions as a Lead Developer can resemble those of a small business—in the short and long term—and get an idea of the mechanics and risks involved.

A small business strives for a positive cash flow to continue operations, otherwise it will start creating liabilities such as debt which, if not handled appropriately, could make the company insolvent and force it to declare bankruptcy. At the same time, when a small business produces positive cash flow and becomes profitable for a period, this does not guarantee it will remain so indefinitely. To increase the chances of its sustainability in the long term, the business must adapt efficiently based on its resources (things the business has some control over) and the environment it operates in (things the business doesn't control).

As Team Lead, every decision your team makes should be considered of high importance and should contribute to the profitability of the business. If your team meet their goals during a sprint or two, especially during the first months of a new project (as the team is not constrained by previous design choices), it doesn't necessarily mean they will continue this way forever. You must remember requirements and resources are always changing and you need to be able to adapt to every new challenge.

If you observe the impact of all the team's decisions over time, you can get a clear image of your team's performance. As you'll see in the next sections, to assess the team's performance more accurately, and understand what problems you might have now and in the future, you should establish a transparent set of measurable processes that you trust. Also, train and support the team towards adopting such processes.

## Managing credit and debt in the codebase

When we use credit, we instantly create debt with a promise to settle the transaction at a future date. As developers, we can use the concept of credit and debt in our daily coding duties. For example, in order to increase productivity we can take a shortcut now (e.g., hack a quick and dirty solution) to deliver something faster, with a promise to repay or improve the code later. The problem with recklessly using this kind of "time credit" in a software development context is that the business could easily believe we can keep "borrowing" indefinitely[1]. In fact, it might not even know we are operating through "borrowed" resources. If we allow or accept such business pressures, we might never be able to pay our debt. And then, we all lose.

We understand that thinking in financial terms can be odd at first. However, this is a fantastic way to help you thrive in the corporate world, as you will share the same language as the management and executive teams. In our experience, it is a lot easier to negotiate with non-technical people when we use terms they're familiar with, like *credit*, *debt*, *assets* and *liabilities*. Keep in mind that your professional and financial growth depend on your ability to lead the business to a sustainable and profitable path.

[1] The term *technical debt* comes from an effort to explain such a concept to non-technical folks.

# Important points before you move on

We would like you to give extra emphasis to all parts of this book that refer to credit, as it's crucial to understand for both personal and professional reasons. Credit allows us to borrow and operate beyond our means at the current time, with the obligation to repay the loan later. It takes a good amount of understanding and discipline to act responsibly when faced with a choice of fulfilling our current desires and needs, and evaluating how to "repay" in the future. We believe this holds true both in life and in professional software development. A major difference we have observed between teams that are able to develop their codebase sustainably for long periods, opposed to others that can't, is the perception of short-term gains when compared to long-term costs. In other words: accomplishing a goal in the moment without considering the effects of the actions that led to accomplishing it in the first place.

As developers, coaches and Team Leads, we have observed that efficient teams share common traits that support their success. We've also noticed common dysfunctional traits on teams whose members have low morale and who continually miss delivery milestones and often deal with regressions in the system. We believe these similarities do not exist by coincidence, rather they develop through every individual and collective choice we make.

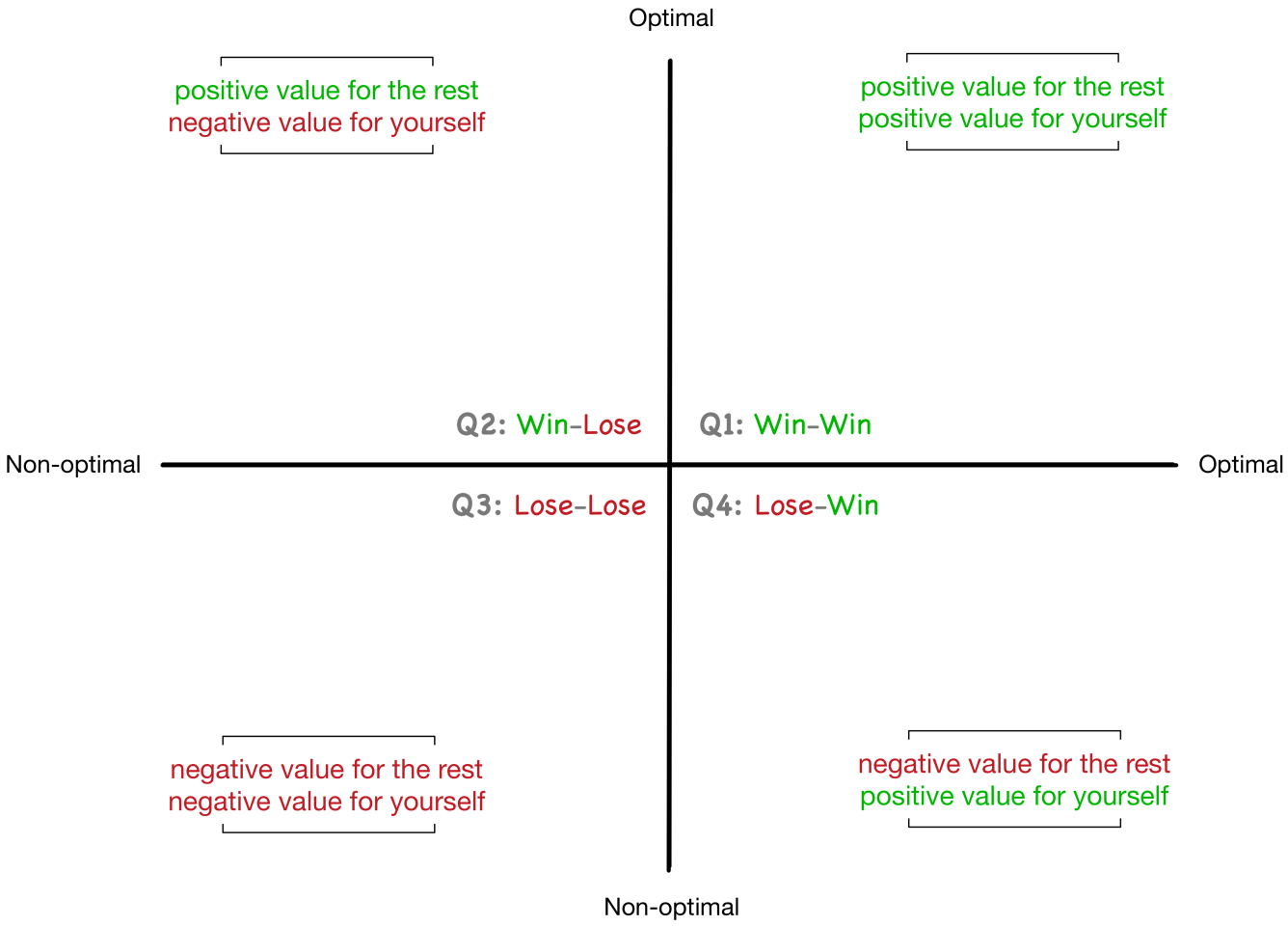
In the following sections, you'll learn proactive processes that can help you maximize positive outcomes by foreseeing, planning and acting upon current and future needs of the team and business. When applied effectively over a long period, they allow you to maintain a very low risk-to-reward ratio. If your actions have a low probability of negatively impacting the team's output and a high probability of affecting it positively, these actions must be taken often.

Part Two

# The Team

# Introduction

If we were to visualize the result of our collaboration in a simple form, including team members, managers, investors and customers, we could end up with the following axes.




Over time, all actions you make will fall under one of these quadrants. It's pretty apparent that your goal should be to end up with most actions falling under Q1 (win-win). This is where the result of your behaviors will yield positive value for both you and your collaborators.

There is one caveat, though. The results of the team's choices are only going to be visible in the future. How do you know you're making optimal decisions? How can you track asynchronous results?

## What's preventing a harmonious environment?

One of the obstacles preventing a harmonious environment where development, collaboration, productivity and rewards are high (win-win) is the gap in the perception of code between the *business* and the *tech team*. For the business, the codebase is usually

a means to an end. The business folks don't think about code. They don't see code. To them, the code is just what the tech team does to materialize the business' vision. They might not even know there is code sitting somewhere. The running software is seen by the business as the real output, and non-developers cannot correlate the codebase health with the product they see and interact with. Talking or thinking about code is irrelevant to them. As it should be.

The firm's management often doesn't possess the knowledge to understand exactly how the code affects the product, what design patterns were used or how brilliant the architecture of the system is. It also doesn't have to. The firm expects and trusts us, the tech team, to make all the technical decisions to build a codebase that can keep delivering a product that accomplishes each version of its road map.

On the tech side though, the programmers value their craft. Every day, professional programmers go to work and spend a big chunk of their time thinking about code, making designs that will later translate into code, reviewing other people's code and writing code. Needless to say, code is a big part of their lives.

This **double perception** of the work undertaken to make a product can be contradicting. Assuming a firm makes all rulings about the future of the product, they might feel like they are in control. The firm creates the vision. They set the goals. They establish the road maps. They hire the experts. They feel in control. This false perception of control makes the firm blindly expect excellence from the tech team. They expect the team to realize the firm's vision without understanding their work and how dependent the firm's future is on the programmers' decisions.

In this standard work dynamic, the programmer is perceived as not having the autonomy to make critical choices about the vision and future of the product.

However, this is a big mistake for the firm. Programmers' decisions affect the goals and road map directly. Excellent programming decisions keep the future of the project under control. Lousy programming decisions can take that control away from both the firm and the programmers (lose-lose).

Lousy programming decisions might not affect the product instantly. It may take some time, but inevitably, by accumulating reckless outcomes, the team will hit a point where they're completely out of control. When this happens, it's common to see the company blaming the tech team, and the tech team blaming the company.

## Who's to blame?

From our experience, all parties. The blame lies in the false perception of work dynamics. Both parties need to engage with a win-win mentality to make their collaboration work.

We have seen this phenomenon repeatedly happen, which made us try to find answers to the following questions.

## Why are companies blind to this phenomenon?

We've already shown one of the reasons: the companies believe they are in absolute control. For example, they might be the deciders of the processes and tools, the goals, the hiring, etc., but after a while their plans slow down because the codebase—which is out of their control—is a mess. This reminds us of Martin Fowler in "Flaccid Scrum" and the reinforcement of one of the Agile principles: "Individuals and Interactions over Processes and Tools."

## Why are developers blind to this phenomenon?

One of the reasons we've concluded through our research is because lose-lose situation do not severely affect the livelihood of developers. The current dynamics of the job market, as of this writing, allow many software developers to jump through jobs on demand. However, this attitude do tend to prevent developers from getting out of the median salary caps, so they end up missing the big opportunities. Also, we don't know how long this positive market trend for developers will stay the way it is, which is quite alarming.

## Such behaviors, from both sides, create a conflict of interest

In a system where two or more entities depend on each other, the entities are bound to be affected by the choices of all involved. In this case, if developers act suboptimally for themselves or the rest of the organization, then they are damaging themselves (including their family plans), the product, the customers, the rest of their colleagues and anyone who is included in the system in which the firm operates. The same goes for the company's behavior. It may not be immediately visible to them, but the damage is there. The result of suboptimal behavior will come back to the participants in various forms, such as miscommunication, poor techniques, stagnation, pay cuts and termination of employment.

The first symptom of a poor collaboration is usually frustration in the workplace. Frustration comes in various flavors. The common ones are: low morale, burnout, depression, missed deadlines, pointless meetings, blaming, code freezes, rewrites and

recurrent regressions.

To maximize the contributions of the tech team, the Tech Lead must identify, nurture and support a set of actions and guidelines that prevent team members from performing tasks arbitrarily. Instead, team members can learn how to account for various factors based on a ”cause and effect” paradigm. Although this is hidden to most professionals, it can have significant weight on the outcome of their actions. It gives the whole team a rightful sense of purpose, as they can easily track the results of their actions.

In this section, you'll learn essential elements when it comes to the resources and processes we use in everyday operations in software teams, including recruitment, onboarding, communication, skill growth, mentoring and dysfunctions. We will analyze what parties are involved in each one, what their relationships are and how you, as Team Lead, can improve them by planning for maximum short- and long-term yield of value.

# Recruitment

The recruitment and employment of new developers is one of the most vital processes for a tech team and, as an extension, for a company. Although many of the roles and responsibilities in a software-based company can be delegated and implemented by non-programmers, building the codebase and the running application—the actual product—isn't one of them. To create such a product, a company needs to recruit and collaborate with us, the professional software developers.

Recruiting the right people for your team has become perhaps the most challenging task in the software industry. The need for several types of expertise simply cannot be met by the supply side, that is, the developers in the job market. Because of the high demand for experts and the unregulated nature of our industry, the barrier for someone to enter and provide services to potential employers is extremely low, which has both a good and a bad side.

To establish an edge against competitors, firms with excessive resources may offer striking compensation packages for both permanent and contract roles in order to attract highly skilled candidates.

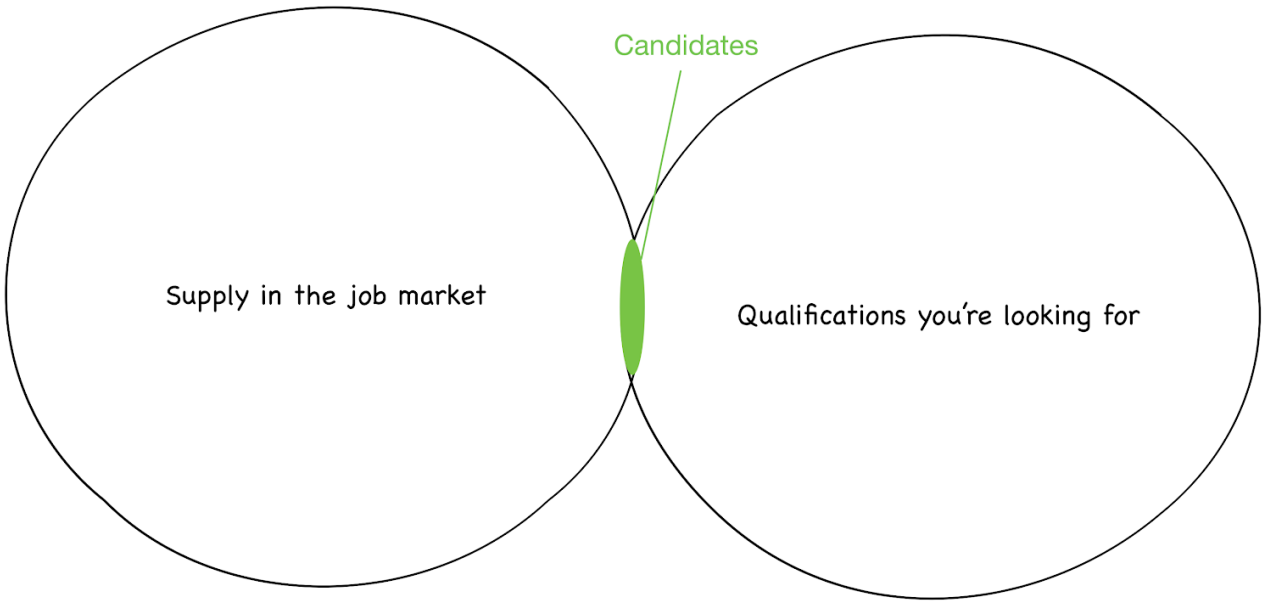
The primary filter and metric to classify candidates is the seniority model that usually depends on their experience. As time goes by and developers gain more experience, they rise up the seniority ranks. This model has two significant flaws. Firstly, time spent in an office doesn't guarantee "gained expertise." Secondly, there's no consensus of what a senior developer is, which makes the title an irrelevant metric. It's up to the developers to decide which title they own, and the impetus is to artificially inflate or promote to a more senior level in order to benefit financially from more "senior" roles, which typically pay more than "mid-level" or "junior" ones.

It's not about good or bad. It's all about incentives. If the incentive is money, and money comes with the title, professionals will do what's necessary to achieve that title. "What's necessary" is the key element we need to understand. With the current low barrier, not much is "necessary" to achieve this money. So, "Why should we bother going the extra mile to be better?" (A prime example of poor mentality).

Although companies need highly skilled developers, the motivations imposed by the industry say otherwise. All the above reasons make it very hard for the person in charge of recruiting to distinguish and filter out the right people for the team.

Due to having to deal with such dynamics in a job market, the recruitment process

resembles the old saying: "looking for a needle in a haystack."




## Exposure to risk

Recruiting and bringing a new team member on board can be seen as a risky proposition for both sides of the transaction.

On the one hand, the firm is betting that a new person who hasn't worked in its context before will, in time, increase the team's productivity and contribute to the product's future sustainability.

On the other, the candidate is willing to commit to a new employer, new colleagues, new product, new processes and overall new environment, which all bear risk.

The reason we used the word "betting" for the firm's behavior and "willing" for the candidate is because neither of the two can be sure about the outcome of the collaboration.

The firm understands that without the right people, who possess the appropriate level of expertise, there can't be a prosperous future for its product—perhaps even no future at all.

On the candidate's side, an ambitious future—employment-wise and financially—is dependent on joining a functional and thriving business. Even though the job market may allow developers to switch between jobs on demand to run away from a bad employment experience, going back to a job hunt might jeopardize the candidate's short-term career plans. Even worse, we've seen and experienced the downsides of

being part of a dysfunctional tech team, and we can confirm that it can affect developers at a harmful psychological level. The joyful and enriching experience of working in a functional tech team is a much better place to be.

## Middle ground

For these reasons, the foundation of an efficient collaboration must rely on mutual incentives for both the firm and the candidate. This foundation consists of an array of resources that both sides must be aligned with, including vision, team members, processes used for developing software, communication within and across teams and compensation.

All of the above are connected with each other and, when put into practice over time, they must yield value for both the firm and the candidate. If none or just one of these resources are aligned, there is a good chance the collaboration will turn into a liability instead of an asset.

## Candidate skills

To reduce the exposure to risk, the Lead and all other agents responsible for recruitment should analyze the fundamentals of a candidate's attitude, behavior and skill set. Ultimately, the goal for the Lead should be to understand how candidates interact both on their own and in the context of the tech team.

When hiring contractor developers, knowledge and skills to fulfill the task are mandatory and must be assessed accordingly before and during the contract. Since businesses won't train contractors, they must be able to optimally fulfill their duties, so a continuous analysis of their performance is essential to a harmonious collaboration.

When hiring permanent developers though, prior employment or coding expertise is not the most important factor. With proper guidance and practice, programmers can improve their coding and interpersonal skills. However, to do so, they must be open and eager to follow and learn from more knowledgeable programmers. Thus, one of the first skills that the recruiter or interviewer should consider is the candidate's attitude towards the unknown. The recruiter or interviewer should be able to identify how the candidate reacts to something unfamiliar and what kind of means the candidate will invoke to solve a problem.

Additionally, it is imperative to get an idea of the level of responsibility candidates bring to the table and what their incentives are for applying for that position. Are they applying *solely* because of financial reasons and the prospect of enriching their CVs, or

are they looking to join a team that is on the way to creating something great and are willing to commit to the cause?

Moreover, you must seek out the level of transparency and effectiveness candidates have towards communication. This metric is not equivalent to the number of words a candidate will speak to other team members during everyday operations and meetings. It is about how well they can describe a specific situation, help others, ask for help, document tasks, clarify requirements, structure code, commit size and message, tests, naming conventions, code review comments, diagrams visualizing workflows or architecture, user stories and tickets.

Finally, you should find out the candidate's perspective towards ownership: *does the candidate take ownership or try to push responsibility to others?* For example, when asked about testing strategies, does the candidate ask if the company have a QA department to do this job or do they show interest in testing their own work?

All the previously described behaviors encapsulate the levels of empathy that someone brings to the team. Empathy is not about *agreeing* with others, but *understanding* their side, which is vital for a fruitful team cooperation. Empathy should be considered one of the most significant assets a team can possess as it extends to sharing and caring—traits that are required for the sustainability of any product.


## Skill assessment

There are many different skill assessments that companies employ depending on their values and the type of position they are aiming to fill. These skill assessments typically consist of multiple interview stages in which the company seeks to validate that the candidate has the appropriate expertise to be part of the team. Skill assessments are one of those processes that can quickly become a liability for a company instead of an asset. On one side, it can be wasteful to spend time assessing *every* job application, on the other you do want to allocate time to find the candidates who fit the standards and culture of the team. The Tech Lead should establish various filters that can minimize time spent on skill assessment tasks and interviews, and allocate this time when it looks highly likely that a candidate conforms to the team's standards.

One of the most basic filters the Tech Lead can apply is a simple coding exercise. Regardless of the activity's output, the Tech Lead should focus on traits such as responsibility, communication, empathy and programming expertise, all of which can be identified by the testing strategy, lines of code in classes and functions, architecture, naming convention, separation of concerns, dependency management, git history to

understand how tasks were broken down and self-documenting code techniques. To the trained eyes of the reviewer, this should take a lot less time than scheduling meetings and conference calls without knowing whether the candidate conforms to the team's standards.

Going the extra mile, assessing the candidate's skills through pair programming is a great way to learn more not just about what the candidate can produce, but also the process they use to get there. Aim to understand their communication and collaborations skills. Make the challenge not too hard so they can effectively go through it in a time-efficient way, but not easy so that they have to ask for help and show their communication skills.

## Organic recruitment

Firms that care about, and are looking to invest in, their future are always on the hunt for the right people. This translates to a fiercely competitive talent market where it can take months to find a great candidate for your operation. Because of this dynamic, a Team Lead can rely on "organic recruitment" by looking at graduate or junior level programmers who can, with the proper training and guidance, evolve into employees who will be able to facilitate the firm's needs over time.

As we mentioned before, coding skills can be acquired with proper mentoring and practice. It's the attitude and ownership that are scarce attributes to find. The Team Lead should be looking for aspiring and empathetic individuals who are eager to learn (and eventually teach!) and evolve professionally as well as financially. The candidates should have an incentive and a mindset for mastery and long-term fulfillment, instead of just looking for an easy way to reach the median salary of their industry in order to profit in the short term. They must be able to understand that their training is continuous and consists of studying multiple disciplines and fields instead of just coding.

A common objection from a business is: "What if they leave the job after we spend money and time training them?" To which we normally reply: "What if you don't educate them and they stay?" Remember, the business that is not setting high standards for its team is racing to the bottom.

## Behaviors table

| High-performance professional | Behaviors to avoid |
| --- | --- |

| | | |
|---|---|---|
| **Mindset** | • Long-term fulfillment (maximizing credit/assets) | • Short-term gains with a long-term loss (maximizing debt/ liabilities) |
| **Approach** | • Practises continuous learning<br>• Seeks mastery<br>• Creates and maintains opportunities for sustainability in codebase and team<br>• Identifies problems and acts proactively to fix them<br>• Is empathetic towards people and code | • Not willing to continuously learn, because the gained knowledge is sufficient<br>• Sacrifices sustainability for a working, perhaps "hacked," solution<br>• Identifies problems and patches them instead of fixing them<br>• Has a lack of empathy towards people and code |
| **Risk** | • Understanding and good assessment of risk | • Inability to perceive hidden risk and costs in actions |
| **Possible outcomes** | • Constant evolution inside and outside of the business<br>• Constant improvement of resources and conditions in the team and firm<br>• Systematic increase in financial gains | • Frustration<br>• Missed deadlines<br>• Time spent on defects<br>• Miscommunication with own team and others<br>• Financial stagnation |

# Onboarding

As the first process new members of your team will witness, onboarding is a fantastic opportunity to show how the team operate and demonstrate what kind of behavior is expected from them in the future.

Firms that are looking to maximize efficiency understand how vital this first impression can be, and they want to turn any possible discomfiture into a smooth transition for the newcomer and the rest of the team.

When a new member enters the team, regardless of their level of expertise, there will be a transition period for them and the rest of the team. The goal of the Team Lead should be to help the team establish a set of methods that will transform the onboarding process into an asset, by maximizing its Empathy, Integrity and Economics to create a win-win situation for the new member, the team and the firm.

## Empathy

The transition for the new member should be swift and short. By swift and short we don't mean that there must be little time spent with the newcomer. The goal is to invite the new member to interact with the team's processes and start contributing as soon as possible.

Failing to smoothly allow new members to interact with the system is a symptom of a bad onboarding method. A new joiner might feel afraid to interact with the codebase, for example, if the process to get access to it is too complicated or restrictive: "Don't do this or that, or consequences will occur!"

There should be a clear path as to how to onboard people, and permissions should be set in the system in a way where new joiners shouldn't be afraid of interacting with it.

For example, a new joiner shouldn't be able to access the production database, but they should be able to interact with a safe clone of the production environment to get a sense of how to work in the product. It seems obvious, but countless times we see companies losing data and blaming a new developer for "doing something wrong" with the production database. It should never be the new member's fault.

Smart teams eliminate this fear and welcome new members to collaborate as soon as possible. Also, by reducing the onboarding time, the team minimizes the time needed for the newcomer to start producing value for the team and the firm. It's a win-win

situation since the new member will feel welcomed and valuable from day one.

It is up to the Team Lead to make sure the newcomer will be able to overcome any adjustment difficulties. The team should always try to behave with empathy towards the new member regardless of their seniority level. For example, being a senior or junior doesn't matter if the new member has just relocated to a new city to pursue the next step in their career. This may signify that they have zero acquaintances in this new environment.

The team should make the newcomer feel welcome, especially when the new member is a beginner developer with little to no experience. If the new member has no prior real-life experience in a work environment, they might feel overwhelmed by the new responsibilities, the city, the office, all the names of colleagues they must remember, etc. Such change could result in high stress levels which, in turn, may lead to a suboptimal performance and a more extended transition period for both the team and the newcomer. Make your expectations very clear to the new member to prevent them from unnecessarily increasing their stress levels. Pairing is a great exercise to welcome newcomers and get them contributing to the team from the beginning.

## Integrity and Economics

The onboarding process should indicate the team's level of professionalism through structured processes that eliminate arbitrary choices. Onboarding serves as the second phase after recruitment; thus it gives the team a chance to assess the new member's attitude and behavior in a cost-effective, organized and efficient environment. The onboarding process is also an investment phase where the new member doesn't yet provide value to the team in a way that is useful and profitable for the firm; however, it should facilitate a smooth transition from investment to a profitable collaboration.

Moreover, onboarding should set the tone for how things work on your team. It should not just serve as a grace period for the newcomer, as they still aren't responsible for what went into the codebase, but you should utilize it as an educational tool to teach the newcomer how the team thinks and operates. By doing so, you increase the chances of the new member settling and adapting faster to the team's methods, which can save you valuable time and avoid problematic or counterproductive behaviors in the future.

As Team Lead, a metric you can use during the onboarding process is to set expectations for new members. This serves as weekly goals for them and assesses their progress. These weekly objectives should derive from a template set up by the team's expectations for newcomers. If possible and cost-effective, adjust the template for each new member. The onboarding plan will depend on their experience, the level of

challenges they will be exposed to and the resources (e.g., time or personnel) you can use to help them get up to speed with the product, domain, processes, codebase, road map and communication with other teams. To maximize its effectiveness, new members should be encouraged to ask for help when they're feeling stuck and discuss their concerns openly.

Ideally, the weekly goals should progressively increase in value proposition as shown in the image below.


For example, week one is dedicated to meeting everyone in the team and pairing with a senior member in order to get used to the development and deployment process. In week two, the newcomer should deploy their first contribution to the live product.

To turn onboarding into an asset instead of a liability, the Team Lead can take advantage of steps that revolve around transparency and accessibility.

## Transparency

As your team welcomes more new members, you can ask them to document any steps they take so the next new member can benefit from them and save valuable time. By

doing this, new members will have the opportunity to practise empathy towards their future colleagues and, with your guidance, learn how to initiate and establish a useful process. Keep in mind that this practice doesn't require a new member to join the team —at any given time you or a member of your team can start documenting anything newcomers will need when they enter the organization.

You should keep in mind when documenting processes, workflows or code, you are essentially documenting a specific version of that process, workflow or code. If someone makes a change in the future, the recorded version will be out of date and will not reflect the truth to the reader. Documentation that requires frequent changes, like the codebase's design, can be very costly. By having the documentation diverge from each new version of the codebase, it creates the need to a) remember to update it and b) allocate time to update it. This practice can quickly turn into a liability for the team, as members will need to refer to it daily to reflect changes in the codebase. Instead, this is an excellent opportunity to explain and show the team (especially junior members) how you can avoid doing this if the code is self-documenting. This means the components (classes, functions, boundaries), the tests and the project structure reflect the user stories and features required by the product.

Another way of keeping onboarding up to date is to ask the new joiner to find flaws in the method and document them. "What wasn't clear enough?" "Where do you feel we can improve?" The team should strive to get an outsider look at the process, as great improvements can be introduced thanks to the new joiner's point of view.

On the other hand, documenting processes that change less frequently is less costly. This means they will reflect the truth more often, which can be a lot more beneficial for anyone who wants to, for example, find out how to perform certain tasks or what they should do next.

## Accessibility

Another aspect of daily operations that must conform to the Empathy, Integrity and Economics pillars is the accessibility of resources to the newcomer. Making equipment and supplies available to the newcomer as quickly as possible will most likely result in a more welcoming and smoother transition period. It will also increase the chances of enabling the new member to be productive, leading to more value for the team and, as an extension, the firm.

For example, if the newcomer asks a question about how to access resources—such as project management tools, codebase repository setup instructions, access levels, architecture diagrams, code and naming conventions, wiki links, educational resources

such as books, videos and articles or human resources-related content—it would be much more efficient if they could find the answers without relying on asking a colleague who may be absent or busy with a different kind of work. ("We can't onboard the new developer accordingly because John is on holiday.")

Instead, a "starter kit" full of useful links and descriptions of rules can eliminate the risk of the newcomer feeling stuck and with no support. By having access to this, the new member can also take advantage of practising their independence and achieving something on their own. This can be especially crucial for more junior members, as it can boost their self-fulfillment and accomplishment levels.

## Behaviors table

| | High-performance professional | Behaviors to avoid |
|---|---|---|
| **Mindset** | • Economics driven, trying to maximize value for everyone | • Lacks critical thinking (e.g., blindly following processes without measuring their worth) |
| **Approach** | • Makes expectations clear from day one<br>• Sets and assesses weekly goals for new members<br>• Sees onboarding as an educational transaction to expedite the transition to efficiency<br>• Empathetic, understanding, always ready to help | • Doesn't seem to have control over the transitional period<br>• Lack of guidance towards the new member<br>• Expects new members to figure things out on their own<br>• Not empathetic and understanding, especially towards junior members<br>• Gets anxious when there are new members because doesn't know what to do |
| **Risk** | • Minimizes the costs of the transition period until the new member is productive and profitable for the team and firm | • Arbitrary actions which may result in bigger costs |
| **Possible outcomes** | • More independent team members<br>• More valuable processes<br>• Autonomous thinking processes by team members | • Sole individuals who possess knowledge that others might depend on |

# Mentoring new talent

Mentoring new talent is an essential part of the tech ecosystem. As new developers enter the tech labor market, they need to get the right training to be able to visualize the goals of various organizations. Shrewd entrepreneurs and managers realize that the worth of a developer isn't equivalent to their price or salary, so they can't hire two lower-paid junior developers to replace a high-paid senior developer. It's their skill set, discipline and creativity that contribute to the sustainability of a product.

Junior developers entering the tech labor market are quite fortunate, as they are primarily being compensated to learn and improve. The firm should see new talent as investments, since they may not be able to contribute to the product at the time they are hired. It is your responsibility, as Team Lead, to understand this dynamic and expose the junior members of your team to a sustainable learning path as well as to adjust your behavior to facilitate their growth.

Over the years we've seen many approaches as to how juniors are treated in teams. Some companies have a policy of not training their junior members, as they believe they are doing a favor to their competitors when the junior employees leave, so they seek a purely transactional relationship. Some managers are fond of a Darwinian approach where they let the junior members try to solve their problems on their own; if the juniors don't perform well, managers perceive them as not qualified to be on their team. On another note, some teams believe significantly in empathy: experienced developers realize the junior members are potential future talent and are going through the same challenges they did in the past, so seek to help out as much as they can. There are countless ways to manage junior members, and they all have their merits according to the vision and culture of the organization. We recommend the empathy path, but, in our experience, it's not the easiest path to follow. There's a dichotomy when balancing the short-term cost with the potential long-term return of training new members. Not every company can afford to do so. As Team Lead, ask yourself: "*What's the best plan we can devise with the resources we have?*" Successful leaders are continually asking themselves how they can optimally allocate the resources they manage in order to meet their expectations and, subsequently, the expectations of the organization. Managing resources well should facilitate access to more resources in the future, allowing you to better prepare your team. That, of course, includes training for your teammates.

As Team Lead, you are most probably responsible for the hiring of new members. Remember what made you pick them from the sea of developers. What were the traits you saw and thought they were a good fit and a good bet to offer them the role? Try to utilize and maximize the growth of these traits as well as enriching whatever attributes

the junior is missing. We are fond of the quote: "There are no bad students, only bad teachers," and the ball is in your court to make it work. Of course, when working in a professional environment, you need to rely on reciprocity and understand when a relationship is not working or when the incentives of a team member are not aligned with the rest of the team. In the case of an endless struggle, consider taking the initiative and spending time with the novice developer to understand why they can't seem to fit in or adapt.

## Mentoring team members with a tailored plan

Mentoring is not just for juniors, but all team members. If the company you work for doesn't have a means for evaluating your team, we highly recommend you create one, even if only for you to use. The evaluation in this context isn't there only to motivate the progress of each of your team members and report it to superiors, instead use it also as an assessment tool to discover the strengths and weaknesses of your team.

You can create different areas such as communication skills, technical expertise, understanding of the domain and dealing with stress. When starting this process, you can use your opinion as a qualitative metric for measuring their performance. As you advance, you can shift to a quantitative approach. For example, for each mentee, evaluate estimation accuracy and blocked tickets slowing down the team. Then, discuss improvements in one-to-one sessions. Again, the goal is to gather as much information as you can for individual team members and use this information to implement a plan for them to improve.

Perhaps a junior developer isn't doing well in tasks where UI work is involved, or can't seem to grasp a specific design pattern, or they continuously get stressed because they underestimate the time needed to complete a work item. On the other hand, more senior developers might be struggling to reduce high-coupling between their components as they have never been exposed to modular design before. All the above scenarios are acceptable if appropriately addressed.

After the first evaluation, you can start creating tailored expectations for each individual. It is imperative to communicate expectations as clearly as possible and guarantee that the team member is precisely on the same page as you.

Start by setting up weekly or bi-weekly goals for your mentees and monitoring their progress. You can assign specific team members to particular tasks that pose a challenge to them and, by providing the necessary assistance, you can help overcome them. Ideally, you can appoint junior team members to more senior developers and delegate mentoring to them. Such practices can come in handy when, for example, a junior

developer isn't familiar with a method that requires consistent training to grasp and put into use, like test-driven development (TDD).

We often see job descriptions that require teams to be familiar with TDD, but when the developer gets hired they might not be able to meet the standards of everyday operations. This is understandable and shouldn't demoralize you as Team Lead. Instead, you ought to understand that these practices can take a lot of time to develop productively and it is up to you to facilitate the learning curve for all members. In other words, you should understand your and the firm's incentives and don't fight back if a team member can't deliver what you have in mind. Create and share with them a sustainable path to reach this level of competence by providing quality educational content such as courses, videos and books, but also a chance for developers to apply what they've learned in small-size features or projects under mentor supervision. Moreover, you can organize weekly learning events, where your team members can share insights from their training and can enlighten or bring their teammates up to speed.

You need to make sure you understand that mentoring can be a very challenging task and few people develop the attributes to make it beneficial for their mentees. You need empathy to understand their specific needs, as well as providing a detailed and clear path for them to improve in a time-effective way.

Finally, **as Team Lead, you're probably the one who needs to learn the most**, so we highly encourage you to seek your own mentors.

## Pairing

One of the methods that development teams are often seeking to utilize is pair programming. Pair programming can frequently be considered a golden method for working within teams, usually advocating that two sets of eyes are better than one. This technique, when used correctly, can certainly provide a lot of value by accelerating the development process and filling the codebase with integrity. However, it can also prove to be a liability for your team if you blindly follow the term, assign your team members into pairs and just let them go. As always, there are some guidelines you need to follow to end up with an efficient routine. In this section we'll examine some examples that will help you get the best out of pair programming.

As we have advocated throughout this book, the team's actions need to conform to the three pillars of Empathy, Integrity and Economics in order for them to maximize productivity and quality, and successfully contribute to the rest of the organization. Pair programming is no different. There are various reasons for the members of your team to

pair program, such as training team members, solving problems in collaboration, sharing knowledge, nourishing team spirit, increasing confidence or experimenting with different techniques (shared learning). However, you should keep in mind at all times that the result should always be the same: a contribution to the sustainability of the product you are building.

When pairing, one is the driver (owns the keyboard and types the code) and the other is a navigator (guides the solution). Pair programming has some guidelines to make this collaboration effective, including when to switch roles. It is the ability of team members to communicate their opinion—by using good arguments—that can make pair programming a valuable process for a team.

Let's consider the following scenarios:

## Scenario One

- The pair has agreed to switch roles every 20 minutes.
- After two sets of 20 minutes, the driver feels they can be more productive if they continue to drive, even though it’s the navigator’s turn in the driving seat.

Can you see the conflict? Although it might be true that the driver can produce more value in the short-term by continuing to drive, according to this policy—in which roles are alternated every 20 minutes—the agreement is broken. So what should happen in this case? Discard the agreement and let the driver continue coding? How does the navigator feel (most probably demoralized or inferior)? Is this a one-time thing which won’t happen again, or is it a consistent behavior among pairs in your team?

If the driver is dictating the work and doesn’t want to be a navigator at the same time, their pair could become merely an observer. In this case, the driver is actually working alone since the navigator has no role or say in the pairing session. This behavior creates unnecessary grudges within the team and results in a waste of time and resources if the navigator is just observing when they could be working on something else.

Such a symptom is common when the team is not yet comfortable with pairing. To avoid conflicts like these, it’s better to stick strictly to the agreed policy.

## Scenario Two

- The pair has agreed to switch roles every 20 minutes.
- After an iteration or two, the navigator claims to be more comfortable in their current role and wants the driver to continue, as they believe the driver can code

better or faster.

This time, we see the behavior of the navigator deviating from the agreed policy. It's important to understand that it doesn't really matter why this is happening, as there can be all sorts of reasons. Perhaps the navigator doesn't feel confident that day, or they aren't aware of the current domain of the codebase, or they feel that it is easier for them to be just observing from the navigator's seat while someone else does the work.

How should the driver handle this case? Should they pressure their partner? What is the cost of that pressure? If the navigator isn't knowledgeable, will they improve after seeing the driver explain and write the code?

It's hard to judge the reasons. In cases like this, it's better to stick strictly to the agreed policy as well.

## Scenario Three

- The pair has built a strong friendship and don’t want to pair with anyone else.

It's great that friendship is growing within the team, however, we should avoid silos. Pairing with other members enables, for example, knowledge sharing, distributed communication, shared learning, opportunity to bond with other members, effective staff allocation for fulfilling tasks and a higher sense of team ownership.

These three scenarios are negative and come with serious repercussions that can both make pair programming miserable for some developers and negatively impact the whole team's performance, as it's almost certain that frustration will make an appearance. These scenarios aren't imaginary, nor are they just edge cases. They happen every day, in multiple software development teams. We have witnessed these types of behaviors numerous times. Pairing issues indicate a lack of team spirit and fruitful collaboration.

## Scenario Four (Desired)

- The pair has agreed to switch roles every 20 minutes.
- The pair follow their agreement while analyzing their sense of productivity and quality of outcome, adapting the plan when judged necessary.

This is a desired scenario when it comes to benefiting from pair programming. The pair follows the rules but occasionally, based on their sense of productivity, they might decide to shortly adapt to the circumstances, indicating their autonomy, ability to

empathize and collaborate while maintaining aligned incentives and goals. As soon as they regain confidence in the process, they go back to the agreed switching policy.

# Communication

## Direct and indirect communication

Direct communication refers to a peer-to-peer exchange of words in various ways, including verbal (e.g., in meetings, conversations at the kitchen, etc.) and written (e.g., email, IM, or code reviews). In all these cases, the party initiating the communication knows who they are referring to. The indirect form includes body language, tone of voice, facial expressions, actions and demeanors that are not always directly addressed to another person.

As a programmer, one of the most significant examples of indirect communication is the way you structure and organize your code. Code is a communication tool, but when you write code, you do not directly address another team member, rather you leave it "open for interpretation." Thus, the recipient firstly needs to understand what you tried to communicate and then pick up where you left off. If you do a great job, the code intent can be interpreted correctly. However, it also depends on the *level of proficiency of the reader*.

Imagine for a second that a new member is joining your team. The newcomer has no preconception of the codebase, processes or the rules in place. What they'll see when they open the project is what the rest of the team have been communicating from the codebase's first commit until the moment the new member looks at it. Perhaps some people who originally committed code are no longer with the firm. The new member will inevitably create a mental image and start filling in the puzzle of how this team operates so that they can fit in and contribute to the product. If they find a rigid and fragile codebase, the message from this indirect communication would be translated to something like "bad news" for them. On the other hand, if they aren't experienced enough to recognize bad practices, they may learn and form bad habits by following them.

This example is one of many forms of indirect communication being used by developers every day. Every single action you take will be interpreted by someone else. Disciplined actions will reinforce high standards and set the tone for expected quality. Undisciplined actions will open space for bad habits, taking more shortcuts and developing hacky solutions. That is why communication is crucial to get right and to keep improving: all actions will be interpreted, even unconsciously, into something subjective by their recipient. As you can imagine, bad communication can start a whole chain of unwanted events. To stop such downward spiral, find and extinguish the source of this chain of events.

Depending on the mindset and incentives of each team member, all types and forms of communication will reflect the codebase's state of health and, subsequently, the product from a tech point of view. The healthier the codebase, the higher the chance of keeping the product moving. The higher the chance of keeping the product running, the more likely it is to contribute to the profitability of the firm. In other words, communication is the key ingredient in the sustainability recipe.

As Team Lead, you have two significant responsibilities regarding communication.

1. Be aware and set the standards for direct and indirect communication for your team and firm.
2. Monitor the direct and indirect communication of your team and maintain awareness for it remaining an asset or, in the case of poor communication, becoming a liability.

# Everyday communication practices

As we discussed previously, every member of a team communicates directly or indirectly with other members through a plethora of actions. In this section, you'll learn some of the most common ways tech teams communicate with each other and how to make communication a valuable asset.

## Code

Code is a way for developers to realize their business's vision. Although this is true, it's a very static view of how members of an organization should perceive code. At the end of the day, what's good about a codebase that has implemented the firm's vision but can't sustain future changes and additions? In our experience, many developers tend to not understand the imperative principle of code being a communication tool as well. If employees believe in the firm's vision, they should work towards building it right now while guaranteeing its sustainable nature for future extension.

Developers must write code in a way that clearly states what their intentions were at the time of writing it. They do this by following a set of guidelines and principles that will allow the next assigned person on the same piece of code to continue the development with ease. By doing so, developers can guarantee they won't be confusing their teammates as well as their future self (who might very well be the confused next person assigned to work on that part of the codebase).

## Committing and commit messages

The same principles apply to committing and authoring the commit message. Imagine a scenario where you're a newcomer to a team, or you have inherited a project, and you search the provenance of a file as you're trying to fix a bug where a date format is wrong. You then open the commit history trying to trace when the bug was introduced and discover that this particular file, along with 17 more, are part of a commit titled "Add title label." By looking at the diff, you can't seem to figure out how the changes on the file reflect the intent of the message. On the opposite extreme, describing long lists of changes in a commit message is also not helpful, as it can be hard to relate the message to the specific code edits. Both extremes make code reviews more challenging on pull/merge requests as well.

Such problems often occur because the components are highly coupled in a way which doesn't allow small or independent changes. Highly coupled components—along with the fact that the commit message and changes are not descriptive or specific enough—

will prevent developers from maintaining and extending the codebase at optimal levels of speed and execution.

Every team should set its own convention and standards for committing code in the repository; however, we believe that certain principles must apply.

In our opinion, a commit should contain the minimal amount of edits and its message must be explicit enough to give a clear idea of what was changed and why. In cases where further explanation should be provided, don't be afraid to write your thoughts down exactly.

## Behaviors table

| | High-performance professional | Behaviors to avoid |
|---|---|---|
| **Message** | • Concisely describes the purpose of the commit<br>• Includes ticket identifier if it's the convention or it could help others when they come back, to make it easier for them to understand changes<br>• In case of refactoring, considers amending the commit if the branch wasn't yet pushed | • One word commits with no context<br>• Puts just the ticket identifier in the commit |
| **File count** | • Includes only necessary files for the specific task you're working on<br>• Creates a new commit if you're starting to work on a new file where no dependencies were affected | • Commits all your changes at once because it is convenient |

## Branching, merge requests, and code reviews

Many developers agree that code reviews are helpful and should be mandatory; however, when not done with Empathy, Integrity and Economics in mind, they can become a massive liability for the team.

Since having your code reviewed can be interpreted as being "judged," developers may take comments on their pull requests personally and see it as criticism of their abilities. This can lead to hostility, discouragement and a feeling of "not being good enough." A healthy team culture is essential to make code reviews an asset for the team. Promoting transparency and continuous learning values across the group should eliminate

“bruised egos” and conflicts when reviews are negative. When you know your team members care and want the best for you, criticism becomes constructive feedback that helps you improve.

As Team Lead, you should assess the team's operations and codebase to decide if strict reviews are needed. Remember that, over time, this need might change. If you believe improvements can be achieved through code reviews, communicate that to the team and gather their opinions. They might be resistant to the idea. Explain the benefits and acknowledge the difficulty of *assessing* and *having your work assessed* by others who might even be “less senior than them.” Set clear objectives for code reviews and collectively set guidelines on how to give constructive feedback. Encourage open discussion to promote learning.

Some teams will continue using the code review process even if it doesn’t yield positive results. Often, this is because they blindly follow what the industry seems to be doing. Remember: if something gives you negative value, you should classify it as a liability and try to either fix it or find a better way to achieve your goals.

To facilitate code reviews, merge requests should be descriptive and contain the minimal amount of edits possible. Doing this makes it easy to continually review and merge changes, proving the team can work in small and concise batches.

## Behaviors table

| High-performance professional | Behaviors to avoid |
| --- | --- |

| | | |
|---|---|---|
| **PR creator** | • Concisely describes the purpose of the merge request<br>• Includes ticket identifier if it's the convention or it could help others when they come back, to make it easier for them to understand changes<br>• Creates a good-sized (short) pull request<br>• Provides documentation, comments, links, screenshots or other forms of assistance to help the reviewer's job<br>• A deep understanding and application of Empathy, Integrity and Economics towards pull request reviews | • Takes the reviewer's comments personally<br>• Tries to avoid the review process by asking friendly authorized reviewers to accept the pull request |
| **Reviewer** | • A deep understanding and application of Empathy, Integrity and Economics towards pull request reviews | • Comments without context<br>• Insults the author<br>• Accepts or merges for reasons other than the ones agreed by the team |

## Tickets

To maintain transparency and tracking of the work cycle, it is highly advised you use a project management tool. Whatever the framework for breaking down the task might be —perhaps user stories or work items—the fundamental principle these tickets must conform to is transparency.

Transparency, in this case, comes with two requirements. Firstly, the work must be visible throughout the organization (for restricted-access projects, everyone involved), meaning anyone must be able to see the past and current state of the work cycle as well as a detailed backlog indicating where the current state of development is and where it's going. The second aspect of transparency refers to how clear and concise the requirements of the work are in order to avoid creating problems in communication. It is imperative to show attention to detail, think of and address any edge cases and set up a series of steps depicting the order and subtasks that need to be performed to complete the task.

## Behaviors table

| | High-performance professional | Behaviors to avoid |
|---|---|---|
| **Ticket creator** | • Specifies in detail what is required<br>• Calculates any edge cases and integrations with the current state of the product<br>• Seeks help from other team members when requirements are not clear | • Vagueness in the requirements, as they create a conflict of interest which will lead to frustration and liabilities<br>• Assertive and superior tone<br>• Responds with ambiguous comments<br>• Forms assumptions from unclear requirements |
| **Ticket assignees** | • Provides a breakdown in a checklist or similar format of what subtasks are required to complete the feature<br>• In case of UI work, provides prototypes, screenshots or short videos demonstrating the feature<br>• In case of architectural work, provides diagrams or documentation demonstrating the architecture if needed<br>• Calculates any edge cases and integrations with the current state of the product | • Misses the illustration of a completed task<br>• Doesn't update the ticket's progress based on the work produced<br>• Doesn't respond to inquiries or comments by other assignees or team members |

## Email & IM

Email and instant messaging apps are probably the most frequent ways team members communicate with each other nowadays. To ensure effective communication within your team, and across different teams (e.g., back end or product), it is essential to educate team members on what to do, instead of each one having an arbitrary way of asking and replying to others.

Vagueness and ambiguity are the enemy in these forms of communication. Because of the instant delivery and notification nature, it is much easier to initiate and continue a confusing dialogue. This may lead to unnecessary disruptions in others' workflow. Aim for clarity, making sure you provide a clear context and relevant details. If necessary, give a code sample, an architectural diagram, a flowchart, screenshots, video or any other information that will facilitate communication.

When one of your team members is blocked, email and IM may not be the best medium to get unstuck. Functional teams strive to never be blocked, so encourage the team to ask for help in person or through more direct means like a telephone call.

Finally, junior level developers and new joiners will naturally be more inclined to have questions about pretty much everything, as they should. Instead of helping them via confusing and slow email or messaging threads, pairing them with other programmers is much more effective.

## Behaviors table

| | High-performance professional | Behaviors to avoid |
|---|---|---|
| **Inquiry initiator** | • Has empathy in mind<br>• Provides a detailed context based on research to facilitate the conversation and assist the responder | • Creates notifications and interrupts others too frequently<br>• Is vague and ambiguous in language<br>• Is rude or superior |
| **Responder** | • Has empathy in mind<br>• Provides a detailed response (in person, if necessary) to facilitate the conversation and assist the person who seeks help demonstrating the architecture if needed<br>• Documents the response/ solution for reuse in future inquiries | • Creates notifications and interrupts others too frequently<br>• Is vague and ambiguous in language<br>• Is rude or superior |

## Meetings

Meetings are an activity which, if not handled efficiently, can quickly turn into a liability, costing the firm valuable time and spreading confusion and frustration within the team. You should provide the context of the meeting before it even starts. Attach a description or a request for the attendees to prepare research or any other relevant work if they have to. Use a reminder or a notification of some sort to alert the attendees at least 10-15 minutes before the meeting so they can pause their work comfortably.

We suggest you establish an open question policy which will promote confidence and empathy in your organization. Attendees, regardless of their seniority, should feel free

to ask any questions they want, no matter how uninformed they might seem. Make sure everyone has understood their task and will be able to continue their work effortlessly when the meeting is over.

Furthermore, if the team ends a meeting with no decided actions, it was probably a waste of everyone's time. To avoid confusion or relying on human memory to remember the decisions, the team can keep notes or create the work needed on the go, using project management tools in case of planning or software design meetings. The right methods, such as note keeping, may very well reduce the need for calling another meeting or interrupting the workflow of your team members if the requirements or agreed solutions weren't clear enough. When using a whiteboard to brainstorm solutions, for example, you can take a photo and attach to the task ticket.

You should also make sure to understand when to call a meeting. We have seen a lot of developers attending irrelevant meetings just because they were invited. You should assess whether a session will provide value to your team and who should attend, otherwise you may be wasting valuable time and effort instead of producing useful work.

For more effective meetings, ask people not to bring cell phones or laptops and instead use notebooks for taking notes. This is important. We frequently see developers bringing their laptops to *code* during meetings. When developers code during meetings, they are saying: “Coding is more important than my contribution here.” Such behavior can distract other members during the meeting. If it's true there's something else they should be doing, maybe they shouldn't be in the meeting at all. By asking team members not to bring gadgets, they will think twice before attending a meeting they don't need to. To be clear, we're all grown-ups so don't ”forbid” people from doing anything. Instead, just remind them how their gadgets can decrease productivity in meetings.

To avoid dreadful meeting where no one participates, set up meetings with clear agendas and required members. Explain that the decisions made in the meeting will be final, and whoever doesn't attend will have to comply with and implement these decisions. This small shift can spark people's interest in being involved and contributing because the outcome will influence their work.

For example, here’s an *ineffective* meeting invite:

“Let’s gather to discuss the ‘Recover Password Flow.’ Everyone must attend.”

A *more effective* version would be:

"We have a 30 min slot to decide the first actions we will take to introduce a 'Recover Password Flow.' The dev team will act on the decisions straight away, so please attend if you are able to help. Scott and the security team are required to attend. Scott has implemented the initial Login Flow and has valuable information for the dev team."

## Behaviors table

| | High-performance professional | Behaviors to avoid |
|---|---|---|
| **In meetings** | • Has empathy in mind<br>• Provides a detailed context based on research to facilitate the conversation and assist the team<br>• Promotes asking questions when something isn't clear | • Is vague and ambiguous in language<br>• Is rude or superior<br>• Sees meetings as time off<br>• Doesn't focus on the meeting and does other work instead |

## Unofficial agreements

As the work day goes by, it is common for team members to discuss work in different places other than the office or meeting rooms, including corridors, the kitchen, during lunch-and-learn events or even outside the firm. In case there is an agreement that influences everyday operations, the same principles about visibility and transparency apply here. Any agreements should be documented and updated on the appropriate medium, for example a pull request, a work ticket, etc. Other team members will have the chance to review and get up to speed with whatever was discussed and agreed upon earlier.

# Team leadership

In the following section, we will be calling on the concepts we've learned from *Hal Moore on Leadership* as we go into detail for the specific leadership behaviors a Team Lead should reflect. These are based on the three pillars of Empathy, Integrity and Economics. We will then state some of the practices a leader should avoid.

We will later examine the various monitoring processes and key indicators a Team Lead can employ to review the team's inner and outer communication quality, and how to prevent it from becoming a liability.

## Behaviors to develop

As Team Lead, you should be able to listen to what team members have to say. As the old saying goes: "There is a difference between hearing and listening." This couldn't be truer when you are called to lead a group of people. Allocate time to truly listen to your team members. Make an effort to understand their problems, considerations and even celebrations. This will contribute to a bond of trust, openness and reliance between you and your team. Give the team member the opportunity to discuss anything that could decrease or increase their happiness, collaboration and productivity. Moreover, by listening to others you learn twice as much, as you are exposed to the thoughts of the other person as well as your own.

If the conversation is difficult to follow for any reason, try to take notes on what the person says. You can evaluate the information later, and you also have a reference in case you want to get back to your team member.

Making room for brief weekly meetings with your team member is an excellent way to keep up with their thoughts and provide a safe outlet for them to speak their minds about anything that might be problematic.

As Team Lead, you should always treat team members with respect and aim to create a family-like environment for them, where everyone feels valued and valuable. The same loyalty team members have for you should be given back to them by you.

By creating an environment where team members are valued and trusted, you will help eliminate emotions like fear and hostility which are counterproductive. Fearless and motivated teams can reach their real potential much more easily.

As Team Lead, you shouldn't expect to make requests and have them followed blindly.

Instead, explain *why* a specific outcome is essential and beneficial to the tech team and the company, thus producing long-lasting improvements more effectively. Also, with the *why* in place, it's much easier to propose and discuss solutions. All members, regardless of their level of expertise, should be on the same page about how to proceed.

Praise in public, discipline in private. If a team member performs suboptimally, opt to talk to them and go through the problem alone. By doing this, you show respect to the person, reduce the probability of hurting the team's cohesion and eliminate the chance that the team member will not trust or appreciate you in the future.

Praising in public is also a good motivational tool, especially when frequently used to give the team credit for their achievements.

As Team Lead, you should always put the welfare of team members above your own. You should strive to create an environment that exhibits a sense of team spirit. Success and failure shouldn't be appointed to individuals alone; they should succeed and fail as a team.

As Team Lead, your *primary* responsibility is not the customers or even outcomes. Your job is to take care of the people who deliver the outcomes to the customers: your team members. So continually ask yourself:

- Are my peers OK? If not, how can I help them be at their natural best?
- Do they have everything they need? If not, go out of your way to get them whatever they need.

Be proactive and provide anything you see fit to help your team deliver their work. Either upon request by the members or because you think it could facilitate their productivity.

A prime example of proactivity was the behavior of the Team Lead years ago while Mike was working for a mobile development team in London. iPhone cables seemed to be a valuable commodity around the office, as they often went missing. One day, when we needed to use cables to test the product on the devices, none could be found. The Team Lead, knowing the importance of the operation and the frustration the situation was creating in the team, promptly decided to put an end to the ongoing liability by briefly leaving the office and returning with three new sets of cables. Problem solved.

We never learned whether the Team Lead paid for the cables from his pocket or if the company covered the cost. The team had a problem, he acknowledged it and provided a quick and excellent solution for us. He eliminated frustration and all the common bureaucratic processes with the IT department, where the official (and slow) channel for

ordering new equipment was at the time.

Although this example may seem trivial, it is a primer on how to assess and deal with accessibility issues, which are a constant variable for the team and can end up costing peace of mind for team members and money to the firm—which are both classified as liabilities.

As Team Lead, you should be an excellent example of good communication. Always be honest with your team and anyone else you interact with.

Your goal should be to get the task done well. This requires an understanding of future problems that might occur, and developing a framework around prevention as well as a clear statement for objectives and goals for your team. You should use proactivity as a preventative measure, allowing you to plan, reflect confidence and optimism and inspire motivation in your team.

As Team Lead, you should strive to be informed on current events, trends and innovations in the industry. Use this information to anticipate future events. To make communication more efficient, set up steps to educate your team as well.

You must monitor the indirect communication when your team are keeping you up to date when something is wrong and also when they are giving you hints as to when something is about to go wrong.

As Team Lead, you should strive to develop a combination of a positive mental attitude and realism about the future. Any liabilities created along the way should be mitigated, analyzed and communicated clearly to your team members to prevent confusion and frustration within the team and damage to the firm.

As Team Lead, you should set high standards and be the prime example of conforming to them. People will only feel threatened by high standards if they have no support. That's where Empathy comes into play. There should be no fear, but there must be challenges, otherwise talented and high-performance team members who don't feel challenged will quickly get bored and move on.

As Team Lead, there are times you might face problematic situations full of stress and worries. In these cases, you need to remember that staying true to your disciplines is probably the most important thing you can do, as this will create the shortest path to get back on the right track. You are the one who's called to bring order when there is uncertainty and distress. You should be able to acknowledge the problematic nature of such situations and take the best action to turn things in your favor. Repeat until the risk has been allayed. Ask yourself and the rest of the team: what are you doing that you

shouldn't be doing and vice versa. Reflect together periodically. Make a habit to research why leaders, of any kind, fail, then learn from their mistakes and incorporate their processes and values into your daily operations. If you are troubled or unable to find a solution, don't hesitate to ask for help.

As Team Lead, you might be leading members more capable than you are. Bad leaders fear such situations, as they feel threatened. However, great leaders strive to work with high performance individuals, giving them support and open space for them to thrive.

Perhaps the best trait you can develop is the ability to create other leaders who surpass your own abilities.

In short . . . **be ready, so you don't have to get ready.**

## Behaviors to avoid

When you find yourself in a situation where you can't win an argument, never resort to a reply of: “because I said so.” If you can't justify the rationale behind your instructions, don't try to impose them on the rest of the team. Instead, you should reevaluate your reasons and find another method to communicate a solution. As Team Lead, you need to form solid arguments behind your decisions to ensure the conversation does not resort to "do as I say/because I said so.”

As Team Lead, you must never behave in a condescending, arrogant, spiteful or toxic manner. Never think that the standards for other team members don't apply to you. Team members will look at you as the role model, the person who has the answers and who leads the effort, so you should be the most prominent follower of the rules out of all the team members.

The shrewd Team Lead must understand that everyone deals with information differently and at different speeds. There will be times when there are gaps in communication and one or more parties will fail to understand the goal of the task. In these scenarios, you must avoid being sarcastic towards your team, and make sure you don't insult anyone's intelligence. You shouldn't even be tempted to assume someone is stupid because they haven't mastered a task yet. Intimidating other team members will not make them work faster nor will this inspire team spirit.

If a team member doesn't meet the team's standards or fails to deliver the expected result, do not take it personally. When this scenario occurs, you should always look for missing or broken processes along the way. What could you have done differently in the past to never allow the situation to escalate to this point? At the same time, make sure

you help the team member to understand the requirements of the task and, with proper guidance, allow them to continue their work.

Never think that you know everything you could ever know about your domain and role. This mentality will make you incapable of adapting to any new situations.

As Team Lead, you should avoid seeking the approval of stakeholders at the expense of other team members. All results are collective, as you succeed and fail as a team. Moreover, never take credit for another team member's accomplishment.

## The journey of continuous learning

The burden of a leader is tremendous, and not everyone is equipped with the required level of discipline to carry it in an organization. Open mindedness and continuous learning are vital in order to do this. Anything you read or experience can be used to provide wisdom and solutions to any situation, good or bad.

As a final note about team leadership, remember this quote as it can help you avoid unnecessary headaches: *When you believe everyone is wrong, you're wrong*. Trust your peers and listen closely to what they have to say.

# Culture

“People like us do things like this.” Said by Seth Godin, this is the clearest and easiest sentence to explain culture. It's simple and truthful. Culture is about the story we tell ourselves and others. Culture is about the past. It's about what we've done that defines our reliable behavior towards future interactions.

Businesses frequently try to *force* culture. For example, we often see companies pushing a ”blameless culture” agenda, but as soon as there's a mistake or a confrontation, the business lays off staff and starts blaming people.

Not long ago, Caio was working for a company where the motto was something like: “Transparency with ourselves and our clients.” Then, one day, someone made an error and pushed the wrong Privacy Policy to one of their products. This was an honest mistake that we, later on, added a procedure to guarantee it wouldn't happen again, and no blame or repercussions occurred. However, when we prepared a letter to tell the clients of the product what had happened, the business decided we shouldn't do it or it could damage our *trust* with them. Well, the culture says ”transparency” and it sounds good on paper. When it's time to *act*, they did something different. The culture was not about *transparency*, because that's not what the business did when it was time to *act*.

This scenario shows how actions form culture, not just words. Next time a mistake happens, people will automatically try to hide it. Culture is formed.

People who don't want to act like that will leave. When building a team, it's important to set up the basics correctly or you won't be able to retain the talent you want. Talented people know they can move somewhere else and are not afraid of doing so. Talented people only stay if the culture is right. Eliminating toxic behavior and filtering business noise from the team is the Tech Lead's responsibility. Make sure everyone is proud and happy with their work and collaboration. The Tech Lead cannot allow toxic behavior to destroy the culture and safety of the team.

A proactive attitude to avoid blaming and cultural damage when a big mistake happens is to have a protocol on how to deal with unexpected issues.

In our experience, a common (and bad) process to deal with big mistakes is:

1) Find the "culprit" (blame).
2) Make them fix it (pressure).
3) Punish them and hope the "lesson" will make them put in more effort and be more

careful next time (punishment/shame).

More care and effort can work, but better processes (checklists or peer reviews) are often more effective. You'll have to find what works best for your team. As a guideline, you can start with:

1) Fix the problem (team accountability—anyone can fix it).
2) Identify how it could have been avoided and how to avoid the issue in future (new process).
3) Write or present a postmortem (share knowledge).

## Delegating

If it's hard to focus and deliver your objectives, as other tasks and emergencies can't be pushed back and always get in the way, you might have a delegation problem. If you are organizing your notes properly, but they just keep growing and nothing gets done, you might have a delegation problem. If you set—but can't usually fulfill—your technical priorities (e.g., coding a feature) because of management tasks (e.g., meetings to decide road maps) it may be because you should delegate these technical priorities to other members. In a contrary context, it could be the case that you should focus more on technical priorities and delegate management tasks.

The solution will depend on the business and team structure, and varies over time—a good Lead will identify the needs and adapt accordingly.

For many developers transitioning into a lead role, it can be hard to delegate tasks—especially technical ones. Normally, Leads are the best developers, so they might not believe other people can perform as well as they can. And they are right. Other developers will probably deliver inferior solutions. However, an organized leader doesn't allow there to be an excess of responsibilities.

Good leaders understand that if they don't delegate, they can become the bottleneck. There's only so much one person can do. Good technical leaders can scale their capabilities with training and the right guidance. In other words, delegation is a skill that requires practice in order to be done well. The more you learn how to delegate well, the more you can scale the capacity of your team. Delegation will also free you to focus on more important work.

To delegate effectively, clear communication and guidance are essential. When undesired outcomes happen, try to understand how you could have been clearer and incorporate this learning, so you avoid making the same mistakes. You might have to

train your team and even micromanage a bit. Over time, you'll build trust and will be able to scale your abilities through training other members.

# Dysfunctions

Regardless of how good they are together, every team is moving towards dysfunction. It's imperative that we exercise all actions seen in this book *continually*. In this section, we'll share with you the most common dysfunctions (adapted from Patrick Lencioni's *The Five Dysfunctions of a Team*), and show you how to solve them as a leader.

## Absence of trust

The strongest base for functioning as a team comes with trust. Without trust, no team can function optimally. The absence of trust is the first and biggest problem, from which all other dysfunctions start.

> "In a high-trust relationship, you can say the wrong thing, and people will still get your meaning. In a low-trust relationship, you can be very measured, even precise, and they'll still misinterpret you."
>
> —Stephen M. R. Covey, *The SPEED of Trust*

In our opinion, the word *trust* has been misused so often that it has lost impact and meaning. To understand this, we need to dig deeper to find the real meaning of *trust*. A common belief is that *trust* is about always saying the truth. That *trust* is about doing what you said you would do. ”*Trust me, I can do it.*” These are necessary elements to build trust, but there's more to it.

Let's illustrate with a scenario.

As Simon Sinek depicts brilliantly in one of his leadership talks, imagine a couple are looking for a babysitter to take care of their child for a whole night while they go to a gathering. They have a choice of hiring a person with ten years babysitting experience who just moved into town yesterday and who no one knows. Or they can hire the inexperienced neighbor's daughter, a 15-year-old kid who they have known all their life and who is part of and well known in the community. Most parents we know would pick the inexperienced kid because she is ”one of us.” This is *trust*.

We see the same effect when building a team. Being *experienced*, *honest* and *showing up on time*, for example, just makes you reliable. To know someone is reliable is just having the ability to predict their behavior. It's an important facet when it comes to trust, but it's *not* what trust is. To earn *trust*, you need to be *one of us*. A *real part* of the team. To share the same set of values and goals.

For a team to function properly, success depends not just on each team member staying disciplined, but primarily on each member *trusting* that every other member will do so and help them when they're losing their way. Team members must have trust and confidence that their colleagues' intentions are good and empathetic towards everyone. Without trust, people will be protective and careful, holding back information and opinions—afraid of repercussions or scrutiny. One of the worst things for a team is for its members to be political about their interactions, *selfishly* and *strategically* trying to protect themselves. A functional team allows its members to be comfortable around each other so they can enjoy the environment they're in and be a productive collaborator for the team goal.

In a trusted environment, teams outperform their dysfunctional competition. Decisions can be made rapidly, and the lack of fear makes members comfortable with admitting when they have made a mistake. So, failures can be corrected quickly and they move on, *blameless*. The result is a significant increase in morale and happiness. Lack of skills, failure to achieve desired outcomes and social shortcomings are *not* problems for functional teams. These bad traits are easily identified and fixed when real trust is present.

Going further, a tech business is usually formed by multiple teams. When there's harmony and trust between teams, we have a *functional business*.

In the context of the business, as much as we want to say: ”we're all a big team,” we can't ignore the human nature of creating silos. Silos will happen, but that's *OK* when cross-team trust exists. For example, the *marketing team* doesn't need to know that the *tech team* are creating a great architecture for the app. They just need to *rely on* the tech team to create flexible solutions to fulfill the need to rapidly adapt the product to marketing changes. Now, if the tech team keep that promise, it can be seen as *reliable*. The tech team are then only *trusted* when they're seen as “one of us”—when they understand and behave as a well-known and trusted entity of the business. When their actions mimic what the business—and the culture—represents.

### The most important team

Many Tech Leads get too attached to their own team, which can be good, but they end

up forgetting the most important team: the business. Without a *functional business*, all teams will eventually collapse. The Tech Lead, who has the best for their team in mind, understands and directs their team actions with the business goals in mind. When the tech team are well-suited for business purposes, prosperity can occur. This comes from a harmonious collaboration from all parties.

#### Behaviors table

| | **Functional team (with trust)** | **Dysfunctional team (absence of trust)** |
|---|---|---|
| **Behavior** | • Efficiently uses energy and time on the work and important issues<br>• Productive meetings where decisions are made quickly<br>• Takes risks collectively<br>• Offers help and assistance<br>• Gives honest and useful feedback to one another<br>• Low turnover<br>• Looks forward to collaborating<br>• Clarifies assumptions and negative conclusions | • Wastes tremendous amount of energy and time managing their interactions within the group<br>• Dreadful team meetings<br>• Reluctant to take risks<br>• Reluctant to offer or ask for assistance<br>• Reluctant to give constructive feedback<br>• Low morale<br>• High turnover<br>• Bases decisions on unclarified assumptions |

#### Practical ways to start building trust in the team

The role of the Tech Lead is to build trust *inside* and *outside* the tech team. It takes time and effort, but it is possible and should be the number one goal when it comes to achieving prosperous collaborations within both the team and the business. Since trust can't be built in a couple of hours, we need to get the smallest collaboration right first between team members, then scale up to the organization level—"time and time again. Trust must be a constant exercise in the team and, like everything you'll learn in this book, the foundation is the most important thing.

To build trust, we can start with small and safe exercises to show some vulnerabilities to one another. The Team Lead should be the first one to do these exercises. However, over time, other members should naturally take the lead. When other members start taking the lead in these exercises, it's a great sign that members are now comfortable with one another.

For all the exercises below, we don't recommend you do them off work hours or on

lunch breaks. We understand the company might require you to do them in your own time, but, if possible, respect people's personal time and do them during work hours. Don't ever lie or stage answers in the following exercises. Nothing breaks trust more than lies. Be respectful and honest. Also, very importantly, don't use these exercises as a way to decide each other's compensation or bonuses—"they are *team building* exercises.

### 1. Share your history

It's amazing how people can work together for years and know nothing about each other. Everyone has an engaging background. Every human being is interesting. Allow your team to share some of their stories and watch powerful connections be made. This is a safe exercise that can help people see each other as human beings, rather than just "workmates" or "resources." For this exercise, you don't even need to arrange a specific meeting. During any meeting, bring some questions in before you start: "Number of siblings?" "Childhood funny moment?" "First job?" "Worst job?"

For example, Caio once found out one of his colleagues was a taekwondo black belt. Since he's a fan of martial arts and a practitioner himself (jiu-jitsu), they bonded over this shared interest. They had a couple of difficult discussions before that event, but, over time, they learned how to deal with each other and this shared interest was key to an improved collaboration.

People will relate to one another more, and build empathy.

### 2. Self-awareness: Personality and behavioral traits

Personality and behaviors are unique traits of a person. A functional team respects their individual differences and works them out to produce a productive collaboration. A dysfunctional team disrespectfully blames individuals' traits as the cause of their problems.

Understanding and respecting each others' personality and behavior traits is the first step to a great collaboration. There are plenty of tests you can run on your teams to identify those traits and bring awareness to team members. However, we recommend you use tests that don't define some traits and types as "better" than others . There are non-judgmental and scientifically valid behavioral tests like the Myers-Briggs personality test. People are often surprised by their results, but that's a good thing. Self-awareness is a great way to make people find their weak and strong points, and improve their interactions with the team.

In this exercise, people can share their results, if they want to, and have a safe discussion about them. We recommend you arrange this session with an HR specialist or a

consultant who understands the test's methodology.

#### 3. Team awareness

Usually, we see team members not valuing their peers' collaboration. In day-to-day work we can find ourselves so involved in our tasks that we forget to look around and appreciate each other. This exercise will make people become aware of each others' contributions.

Ask the team to define the best contribution a member brought to them. Focus on one person at a time, starting with the Lead. For example:

"Claudia is the one calling the actions and working hard to get us good results."

"Bob is an extrovert and a great communicator—always being the voice of the tech team during presentations and selling our work to other teams."

#### 4. Let the team decide an activity

Rock climbing? Day at the park? Barbecue? Karting? Let them decide and make it happen!

We once had a "Team Olympic Games" in a park. It was a day off for all engineers and everyone got to participate and have fun. Since the games were so diverse (tug-of-war, egg-and-spoon races, paper plane distance races, etc.) everyone's different set of skills were crucial to team success. Everyone got a shot at being an important part in the game.

#### Continuous improvement

The team will eventually get off track if these exercises are not performed. Make sure to incorporate exercises like the ones proposed as part of the team schedule. Over time, you'll notice that people will start engaging more in team activities, which brings us to the next dysfunction.

## Miscommunication

Have you ever been in a planning meeting where you sit down with the product owners, set the stories and assess their complexity and, at the end, you all claim to be on the same page and effectively vouch to fulfill each others' expectations? Perhaps a couple of weeks later you all sit down in the same meeting room and try to understand why the work isn't done and why expectations were not met? Why do you think this

happens over and over, worldwide, across teams?

Failing to meet expectations and delivering on time happens because of miscommunication. Not to exempt ourselves from this kind of event, we can confirm that miscommunication has happened to us on numerous occasions.

Imagine the following scenario: the front- and back-end teams, after a brief meeting, agreed on the work they have to deliver and the deadline. However, they forget to account for the dependencies between their work. For example, the client application cannot be deployed before the back-end is live. The client application developers can't even write integration tests without setting up a proposed interface (e.g., JSON response format) beforehand. This is a simple example in the chain of events found in everyday cross-team operations. When including other teams, like marketing, sales, product, QA and data analysts, you end up with a very dense network. In order to improve outcomes, everyone involved in this network must understand 100% the goals and challenges. When we say 100%, we mean everything: technology-, estimation- and dependency-wise.

As you can imagine that's a very challenging endeavor.

The real problem, as mentioned above, is miscommunication when two or more parties agree on the same ”thing” but have a different perception of what that “thing“ is. In other words, they speak the same words, but the perception of what they say and hear is different. They are not on the same page, even if it seems as though they are, because they agree on what they are discussing and planning.

This is a severe problem to solve, as it goes against a fundamental human driving force —our ego. If one of the individuals were to stop the others and ask for clarification, they might be afraid of being seen as less knowledgeable. What we have observed over the years is that many programmers are not willing to bruise their ego by asking about what they don't understand.

Miscommunication is mainly generated because of *wrongful assumptions, absence of trust and fear of conflict*.

## Fear of conflict

Passionate debate is essential for a software team to function optimally. The more we work with dysfunctional teams, the more we see enormous amounts of time and energy spent trying to prevent free and productive disagreements and debate. Or, simply put: *conflict*. Conflict can be a taboo for software developers, and, in general, is seen as a bad

thing. Conflict is positive when all parties share the same values and goals, as a team should.

In functional teams, we see free debates and conflict happening all the time. No one is afraid of speaking up when they notice an issue. For example, imagine a situation where Martha didn't want to allow other people to change one of the modules she wrote. Paul: “Hey Martha, we know you wrote the stock control module, and we're thankful for your great work. However, you can't carry on being the only one changing it. You need to pair with someone and pass the knowledge forward—and that means people will challenge some of your choices and you should be OK with it.“ Martha: “I know, sorry about that. I understand I got attached to it and it's not healthy. I'll write documentation and help anyone who wants to contribute, but I'll be very picky with changes.“ Paul: “Sure, that's fine. If it becomes a problem we'll let you know.”

That's just a common conflict example in software teams that we see over and over again.

In dysfunctional software teams, members usually avoid conflict and say things like: “only Martha works on this part of the software,” even though everyone knows it shouldn't be like that. When the person responsible for that part of the software is away (holidays, illness, left the company), nothing happens and productivity goes down. All in the name of avoiding conflict.

*Fear of Conflict* is directly related to *Absence of Trust*. Trusted teams understand all members are working on the same set of values and goals, with the team happiness and success in mind. Thus, trusted teams are not afraid of speaking up when a negative behavior occurs. Trusted team members are happy and thankful when someone calls them out—we're helping each other stay on a prosperous path!

Avoiding conflict can be seen as positive because we are “protecting people's feelings.” That's an inaccurate assumption. Avoiding conflict actually leads to more harm, as people start to resent one another. Talking behind someone's back, gossiping, shrugging and rolling eyes during meetings and hiding information from one another are some harmful side-effects of avoiding conflict.

Trusted teams engage in conflict, but never hurt each other's feelings—they always do it in a respectful manner. Even after heated debates, decisions are made and people are still friendly and happy to engage in a debate for the next issue at hand.

**Behaviors table**

| | Functional team (fearless conflict) | Dysfunctional team (fear of conflict) |
|---|---|---|
| **Behavior** | • Makes decisions and quickly solves the problem at hand<br>• Everyone contributes with ideas<br>• Everyone's opinion is heard and valued<br>• Few to zero politics<br>• Developers respectfully speak their minds<br>• Tackles and fixes controversial matters<br>• Respectfully addresses people's misbehavior | • Can't make decisions and problems tend to escalate<br>• Only a few members speak their concerns<br>• Only a few member's opinions are heard and valued<br>• Wastes time with politics<br>• Doesn't discuss controversial matters<br>• Personal attacks and gossip are common |

### Practical ways to achieve productive and fearless conflict

The exercises proposed in the Absence of Trust section are important to start making people feel confident and comfortable with one another. Naturally, healthy conflicts will happen. Also, knowing each other's personal traits will help people identify how to deal with each other's temperaments. However, when teams believe conflict is bad or unnecessary, they may need a little permission and incentive.

### 1. Conflict extraction

During a meeting, pay attention for points of conflict that people refuse to address. Bring the problem to the fore and tell people it must be resolved before the meeting carries on. Stick to the conflict until it's resolved. For this exercise, you can use a team member or an external person as a mediator.

For example, during a meeting while Roger is explaining his proposed architecture to solve a problem, Seth rolls his eyes and says something along the lines of: "Here we go again." At this point, the mediator must act and say: "Seth, looks like you don't agree. Do you mind sharing your thoughts with us?" Seth might be resistant and say: "Oh, whatever. We keep having this discussion over and over. Whatever . . . "

It's clear Seth is holding back opinions and that's not healthy. At this time, with care and respect, say: "We need to address this conflict to be able to carry on. It's healthy to discuss different opinions, but it's not healthy to hold grudges." It's important to remember and bring everyone back to the same side—the team side. "We're all on the same side and decisions won't all be 100% agreed, but we all must express our ideas and be sure all of them are heard."

Just by allowing people to express their ideas, many conflicts will go away. People might not agree on the solution, but they were heard so they will buy in and work towards the proposed resolution.

### 2. Conflict resolution

What to do when the conflict extraction goes wrong? Aim to turn a possibly harmful debate into a positive one.

There will be times where some team members—or perhaps you, the Team Lead—will strongly disagree with someone else's opinion. When in conflict, everyone must remember two things: you are a team and, as a team, you should all want the best collective outcome.

Disagreements are an excellent opportunity to study how everyone reacts when faced with conflict or with having one developer disagree with a handful of others. One of your goals should be to support and delegate conflict resolution and decision-making to the team so they can mitigate the situation on their own in the future. Use dialogue as a platform and let team members express their opinion.

It's possible that some team members will try to impose their opinion to avoid getting their egos hurt. As a mediator, it is imperative you understand all views should be heard, but also filtered by the core principles of the team and the company. To alleviate the need for judgment and arbitration from a higher order authority, such as the Team Lead, ask your team to justify their opinions according to the shared principles. Thus, an opinion suggesting a solution that does not fulfill the shared principles can be discarded as it won't contribute to reinforcing the integrity of the organization's values.

Be hard on the conflict, but soft on the person. Instead of *"I can't believe you said that!"*, aim on the conflict: *"I really appreciate your input. You're very knowledgeable and taught me a lot over the years. However, I disagree with this solution because it goes against what we're trying to achieve. I know you have our best interests in mind, so I'd like to explain why this solution goes against our collective values. This solution can bring damaging results that will be hard to recover from…"*

The conflict might go on over some time, and people might feel uncomfortable and decide to withdraw. Make sure to encourage healthy debate until everyone has voiced their concerns. Keep giving them permission to carry on and remind them it's for the good of the team. At the end of the meeting, congratulate people and tell them healthy debate is key for team success and should not be avoided.

## Lack of commitment

Team commitment is a crucial part of functional teams. Following the previous dysfunctions, we've already mentioned words like "buy in," "acceptance" and "agreements"—these are key for a strong team commitment.

**In functional teams**, trust and fearless conflict leads people to make decisions collectively. Even when disagreements occur, everyone is heard. Eventually a choice will be made and it leaves no room for doubt or double guessing. Every meeting ends with clear agreements and everyone in the team will direct their actions to achieve the goal. A complete buy in!

**In dysfunctional teams** decisions might be made, but since they were decided by only a few, members might secretly disagree. This creates doubtful and sabotaging behavior, even unintentionally. People will do the minimum possible, and will thrive on seeing mistakes just to validate their assumptions or point out: "I told you so!"

The **lack of commitment** will lead to low productivity, low buy in, lack of goal clarity, unconscious sabotage and more. So, why does it happen?

In the majority of cases, a **lack of commitment** occurs when the team tries to achieve **consensus** in their choices or tries to have **certainty** of their outcomes.

### Consensus

Complete agreement, or *consensus*, is practically impossible. If your team usually achieves consensus, check to make sure you don't have a lack of commitment issue. This dysfunction is connected with the Fear of Conflict, where the desire for consensus comes with the desire to have no conflict. The pursuit of consensus leads to analysis-paralysis.

**In functional teams**, *consensus* is not the goal. The goal is to have everyone heard and opinions considered. The team will then make sure decisions are made and everyone works towards the decided goal, even when disagreement occurred.

### How to deal with impossible agreements

Eventually, an impossible agreement will happen. Everyone was heard, but no judgments can be made. As a leader, we have a “My word is final” card we can use. The problem is: the more we use it, the less authority we have. Pulling the “Lead card” all the time is cheap and will demoralize your team, maximizing its dysfunctions.

When to use that card? On impossible agreements. And those cases should be rare!

For example, Roger and Seth explained their concerns but no one will bend or buy into their counterpart proposed architecture, leading to an unproductive discussion. As Team Lead, you can intervene and make a decision while praising and giving responsibility to both. “Roger will be leading this feature, so we should go with his approach. Roger, please keep an eye on Seth's points as they are very relevant, and consult him throughout the development. Seth, you're highly experienced in this area of the code, so give Roger support on any difficulty he might encounter.”

### Certainty

The need for **certainty** can be a huge problem for tech teams. We operate on an iterative, complex and even creative industry. Most of our actions will have an asynchronous result that we don't see until much further in the future. There's almost never perfect requirements or data. It's hard to trace back where issues started and it's hard to predict problems.

**To operate optimally in the tech industry, teams must be comfortable with uncertainty.** Just like seeking consensus, seeking certainty will lead teams to have unproductive conflicts. Being comfortable with uncertainty means the team is comfortable with its choices. It was a collective decision where everyone was heard, and even with disagreements the team are working together towards the proposed solution. If it fails, it's a team failure, not an individual one.

**In functional teams,** members support each other and are not afraid of failing. Functional teams prefer to fail than to be paralyzed. Not making a decision is unacceptable.

However, we need to be careful. A reckless effect of the “eliminate the need for certainty” mentality is the “move fast and break things!” movement. It sounds good on paper, as it might eliminate paralysis and force teams to make decisions. In reality, and on the negative side, it may allow the team to be reckless and thoughtless in their actions.

**Functional teams are not reckless. Functional teams are fearless in trying out ideas, but seek success above all.**

**In dysfunctional teams**, members might eventually feel weakened by the fear of failure. “Haha! I told you this would fail!” will be more common than: “OK, this idea didn't work, what can we do to fix it and prevent this from happening again?”

### Behaviors table

| | Functional team (committed) | Dysfunctional team (no commitment) |
|---|---|---|
| **Behavior** | • Shared goals and priorities<br>• Proud of collectively making decisions without perfect information<br>• Not afraid of trying new ideas<br>• Can adapt quickly to business changes | • Lack of shared goals and priorities<br>• Excessive analysis with no actions<br>• Paralysis due to fear of failure<br>• Blames dysfunction on the business for changing requirements |

### Practical ways to increase commitment in the team

#### 1. Decisions review

At the end of every meeting, revisit all decisions and agreed actions to make sure everyone is on the same page. You may find many resolutions sounded final and agreed during the meeting, but when revisiting it you may see that people had different perceptions of what had to be done. Quickly revisit the decision and make sure everyone is on board. To be effective, this exercise should be done in every meeting. By doing this every time, it'll expose communication issues and enforce members to commit to choices during the meeting.

#### 2. Small improvements

If your team are resistant and can't collaborate in making decisions without certainty, it might be beneficial to start with small choices first. Choose a simple or low-risk topic where the "lack of information" impeded a problem that needs to be resolved. For example, code formatting. "Should the curly brackets {} start in a new line after a function declaration or not?"—this problem might lead the code to be inconsistent but it's low risk. Gather the team for a short discussion and say: "In X minutes we must make a decision about the code formatting and stick to it. Everyone will be heard and at the end we'll commit to an agreement."

Low-risk conflict resolution and commitment might spark a curiosity in trying the technique when it comes to riskier issues.

#### 3. Decision deadline

The high cost of *not* making decisions can be extremely harmful to teams. One way to

solve this paralysis and help people collaborate and commit, is to **set a deadline for the outcome to be decided**. It may sound weird, since we're used to having deadlines for finishing tasks, but it's super effective. The sense of emergency will help the team collaborate in making decisions.

#### 4. Set up safeguards and contingency plans

If the team are paralyzed by fear of failure and unable to commit to decisions, discussing and agreeing on **safeguards** and **contingency plans** may resolve the problem.

”Arthur did not agree with Richard’s proposed solution, but by discussing a testing strategy to guarantee a better chance of success and defining an automated rollback capability, the team achieved agreement and commitment from all members.”

## Avoidance of accountability

Accountability is one of the most essential attributes of a functional team. Without accountability, quality and productivity (if ever achieved) will rapidly decrease. Functional teams establish high standards and hold everyone accountable to comply with them.

It's important that every individual takes responsibility for their actions. However, in a team context, accountability is not only an individual task. More important than individuals' accountability is team accountability, where every member will evaluate each other. Constantly.

The need to constantly analyze each others' performance is hard. The uncomfortable nature of these feedback interactions is what leads many team members to avoid the accountability feedback **at all**. It may seem acceptable to avoid accountability feedback conflict, especially among teams where people create strong friendships, but it's another counterintuitive action that will damage more than it helps. It's not uncommon to see team members damaging their relationship with other team members because of bad performance that harms the team.

We're social beings and we value team contributions. Avoiding team accountability will only produce harmful feelings among its members. When people constantly harm the team goals, it's natural for resentment to start growing. Resentment is a natural response, but not a positive outcome! Resentment is harmful for all parties and must be avoided at all costs. As team members, we should aim for healthy relationships and shared goals that benefit the group. To avoid resentment and the potential for it to arise,

constant and honest feedback (although difficult) is essential.

If we really care about our team members, we must hold them accountable just like we should strive for them to hold us accountable. It's a win-win situation where all parties can benefit and the team goals can be met. Team accountability shouldn't be blame-based, rather **help-based**. When someone is underperforming, it's not a matter of giving bad feedback and carrying on with our own problems. Members of functional teams hold each other accountable and help each other when they're in trouble or in need of help.

Some may call accountability "peer pressure," but that sounds a bit negative and even over the top. Accountability is, somehow, "peer pressure" but in a healthy functional team, "peer pressure" is just a different way of saying "following good examples." In functional teams, members don't want to let each other down. When teams set high standards, and members are constantly complying with them, no one will want to fall behind. And, just like other positive attributes, accountability and high standards are a "follow the leader" action. The leader must be the highest example of such standards so functional team members strive to comply and even overdeliver results for the team benefit.

However, a big mistake a Team Lead can make is to be the primary and only source of discipline and accountability. When the Team Lead doesn't invite the team to hold each other accountable, members might delegate the accountability actions solely to the leader. This is a huge burden on one person and will rarely generate great results. Also, if you keep micromanaging the team, you're signaling to them that *you're accountable for the outcomes*. If your team members don't feel accountable, their performance, happiness and fulfillment will decrease rapidly.

For example, developers might see someone pushing straight to master without tests, in an attempt to "avoid doing the work" and say nothing as they believe the leader will hold them accountable. A much better approach is to incentivize team's accountability as a collective exercise. "Hey, I saw you pushed some changes straight to the master branch. We don't allow this as we have a strict process. Was there any issue with your pull request? If so, let's revert that and I can help you get it merged."

### Behaviors table

| Functional team (accountable) | Dysfunctional team (no accountability) |
|---|---|

| | | |
|---|---|---|
| **Behavior** | • Sets high standards and complies with them<br>• Honest and empathetic feedback<br>• Helps underperforming members step up to team standards<br>• Every member ensures pressure to improve poor performance<br>• Outperforms and strives to overdeliver<br>• Respects and encourages high standards | • Might set high standards but makes no effort to comply with them (for the sake of "pragmatism")<br>• Unwillingness to tolerate interpersonal discomfort<br>• Resents team members who have lower standards<br>• Delegates discipline and standards compliance to the leader<br>• Mediocre results<br>• No urgency to achieve goals, key deliverables and deadlines |

### Practical ways to increase team accountability

#### 1. Set team goals and high standards

It might seem obvious, but setting clear team goals and documenting the desired high standards is the first step. Many teams forget to write down the standards and, as a result, the shared standards once agreed can quickly fade away. Without such documented agreements or commitments, ambiguity can destroy team goals.

Ambiguity can be a team's worst enemy, so be very clear with who needs to do what, when deliverables must be ready and what metrics have to be met for the team to be able to call the result a "success."

For example, just **meeting a deadline is not enough to call it a success**. Think about how many projects reach "deadlines" with a product full of bugs and features that are in bad shape. In these situations, it's not uncommon for *everyone* in the team to know the feature will become a burden to maintain in the long term, so how can we call it a success?

We can measure success from many angles. For example, by setting up metrics like: analytics to gather feature conversion, testing strategy (with a clear and low cost), no regression tolerance, few to no bugs (mistakes prevention), maintainability (modular architecture allowing new features to arise with low-to-zero undo cost), time from idea to production, rollback capabilities (bonus if you can rollback instantly and remotely) and much more that we'll cover in the Codebase section.

As an important note, don't expect or aim for *consensus* when setting up goals and

metrics. The Team Lead must set up initial goals and track compliance. With time, this task should be a shared team responsibility.

Clarity can be the functional team's best friend. During feedback or discussions, teams can avoid ambiguity by pointing back to the original agreed metrics, goals and standards.

#### 2. Structured team goals and high standards review

Just setting up goals as defined in exercise one might not be enough. To maximize the chance of achieving outstanding results, it's better to set up a formal and structured way to give people feedback.

As stated before, it's uncomfortable to give people feedback, so even functional teams might start avoiding accountability. Don't wait for people to take the initiative! To fight the urge to be comfortable, we can set up scheduled reviews where everyone must give each other feedback. This can be simply written using Google Forms. Or, for trusted teams, it can be verbal, for example, in a scheduled meeting.

As a Lead you must allocate time to this activity and enforce that everyone delivers feedback on time. To do so, a leader may need to negotiate with the business to allocate time during a sprint. A trusted team will easily get the time slot. If you need arguments, remember that maintaining a functional team is essential to achieve business results.

We recommend doing this at least every four months for **functional teams**. It might be necessary to run this more often when the team is underperforming.

#### 3. Team rewards

Most companies still evaluate and reward employees through individual performance. This can be beneficial, but it can damage team collaboration. An alternative is to have team rewards instead, where people's drive will shift to team results (which hopefully will increase team accountability!) Some may think team evaluation might not be fair to the individual, but we don't necessarily agree. Functional teams will not suffer from this shift, but dysfunctional teams will. If you don't have a functional team yet and are afraid, you can keep some individual rewards, but we highly recommend the big rewards shift to a team performance basis. To do so, setting up clear goals and metrics are essential, as described in exercises one and two.

## Inattention to team results

When members are not accountable for team outcomes, another dysfunction will arise:

Inattention to Team Results.

When members don't have ”skin in the game” or are not accountable, motivations are set for them to care only about themselves or in areas that will help them as an individual to progress. It is, at least in our experience, impossible to improve the collective outcome when incentives are set for individual gains.

An example of such behaviors in a tech team is when an individual decides to add a new third-party framework to the project just because they want to learn about it and add it to their CV. There's nothing wrong with learning something new, but it would be more suitable to do this in a spike/learn session or in their spare time. These individuals produce behavior that does not care about the tech team or the effect it will have on the product. Many times, this selfish act will be masked with excuses like: “I’m modernizing the codebase with the newest top-notch industry standard frameworks.” Even though this kind of individualistic behavior will have negative results for the team, the results will be ignored since the individual feels no pressure on **Team Results**.

Another phenomenon that can affect individuals in a dysfunctional team is the Team Status (or ego). It happens more often in big companies where members are happy enough to just be part of a ”prestigious tech team” and don't care about results since just being part of it is beneficial to them (and their CV). Every prestigious tech team can fail as well, so the Tech Lead must exercise the collective spirit to help everyone hold each other accountable.

Functional teams care first and foremost about collective goals. To do this, functional teams focus on specific objectives and clearly defined outcomes. Without these, any decision is fine. Nothing matters. However, with specific objectives and clearly defined outcomes, decisions such as third-party frameworks will be made with a clear focus in mind that can be measured. Finally, the team can track their metrics and expected outcomes and, when these are not met, they can change their minds about the decision. Collectively.

Functional teams have a strong desire to win. To do so, they set not only financial metrics (e.g., profit or cost reduction) but whatever metrics are necessary to achieve the higher goal of the most important team: The Company. Functional companies will set clear goals and metrics that are not always directly related to finance. For example, deployment frequency, mean time to recover, estimation accuracy, lead time for changes, customer acquisition and customer retention. Ultimately, these goals should drive tech excellence and profit.

**Behaviors table**

| | Functional team (focused on results) | Dysfunctional team (inattention to team results) |
|---|---|---|
| **Behavior** | • Supports short- and long-term business goals<br>• Retains high-performance, result-oriented members<br>• Has specific objectives and clearly defined expected outcomes<br>• Incentivizes team collaboration<br>• Success-driven | • Stagnates and fails to keep up with the business evolution<br>• Loses key members and retains lower performers<br>• Unclear objectives<br>• Incentivizes individuals to focus on their own goals<br>• Pragmatism excuses and laziness-driven (“We’ll do our best.”) |

### Practical ways to increase team attention to results

#### 1. Rewards based on specific team objectives and clearly defined expected outcomes

"We'll do our best" is a trap that can lead your team to epic failure.

"We'll do our best to increase the number of tests . . . one extra test is an increase but is it enough?"

"We'll do our best to refactor bad code . . . we added an interface/protocol, but does that mean we created a good abstraction?"

"We'll do our best to facilitate newcomers to the team . . . we wrote a code style guide, but are we following it?"

It's not about *trying*, it's about *doing*. Without specific objectives and clearly defined outcomes, the team will not know what to pursue and what "win" means. No amount of goodwill can help a blindfolded team. As a leader, make objectives and expected outcomes clear and make sure the rewards (compensation included!) are based on them.

#### 2. Public commitment and declaration of results

The “rewards based on specific objectives and clearly defined outcomes" exercise is very important; however, it can be dangerous to focus only on financial motivation. Another great way to hold each member accountable and focus on Team Results is to make Public Commitments to Outcomes and Public Declaration of Results.

When the team is on the spot, the game changes. Teams that are willing to commit and

declare results publicly will have much more incentive to collectively pursue the right outcomes.

For example, in an all-hands meeting you can declare the team has worked hard in the previous quarter, releasing new features that were responsible for 20% of revenue increase. Along with this, they automated 25% of the deployment processes which reduced the cost of dev operations by 15%. But don't stop there, make sure to also publicly commit to higher goals. "Our aim for the next quarter is to deliver two more features and optimize the ones out there to increase revenue by another 25% and automate 95% of the deployment processes."

## Fragility

The combination of all the dysfunctions above make a tech team *fragile*.

Dysfunctional tech teams are *fragile* and get worse over time. The more they fail, the less morale, happiness and energy they have to get back on track.

Functional tech teams are *antifragile.* Antifragility is a concept taken from the book *Antifragile: Things That Gain From Disorder*, by Nassim Nicholas Taleb. In short, an antifragile item don't just resist damage, but gets stronger under stress.

Antifragile teams don't "lose." They either succeed or they learn and get better and readier for the next challenge.

Fragilities in the company will affect the team. Fragilities in the team will affect the codebase. Fragilities in the codebase will affect the company. It's a vicious circle that we must escape. As competent leaders, our goal is to help the team become antifragile in every aspect.

## Pursuit of Perfect

As we've stated many times, low standards and accepting mediocre results will demoralize the team spirit and push away talented members who don't feel challenged and fulfilled with meaningful work. The other extreme, chasing perfection, is a dysfunction disguised as a positive endeavor. Going to an extreme trying to achieve perfection is a costly goalless goal. The team operations will never be perfect, and setting up unachievable standards will backfire in the form of burnout and unhappiness. Instead, aim for continuous improvements in a supportive and understanding environment.

As a final note, don't forget that a functional team is always moving towards *dysfunction*, so these exercises must be performed continuously and be part of the team's day-to-day chores.

Dysfunctional teams are one of the biggest challenges in the software industry, and if you build up the skills to effectively transform them into functional teams, you'll be an immensely valuable asset to any business.

# Growing your team (scaling)

Most commonly used methodologies (e.g., Agile, Scrum, Lean) are battle tested in small teams. Small teams are often the premise to these methodologies. They're all about producing the best outcome from limited resources (which can be mandatory for startups and a great asset for established companies). The problem lies when trying to grow the team past the battle test size. Can you be agile with a team of 50-100 developers? We've seen it in practice and, in our experience, often it doesn't end well when following methodologies meant for small teams.

What we've seen work well is breaking down large teams into smaller ones, with decentralized chains of command—giving them autonomy to make decisions on their own. It's not just the team that's autonomous in its decisions, but the codebase, *when necessary*, can also be decoupled (autonomous) from other teams' codebases.

A team that works well in the small scale, might not work well in the large. So be aware when scaling up teams. Scaling up might be required, but often we can increase the output by removing overheads and bottlenecks. If scaling up (hiring more people) is necessary, strive to create many small decentralized teams and help them achieve good communication and smooth collaboration.

# Laying off/downsizing

As Team Lead, eventually you might have to face one of the most difficult tasks: laying off members of your team. You should strive to solve problems before they go too far and laying off is “the only apparent solution.”

Laying off might be necessary because of toxic behavior from a developer. If someone is abusive or inappropriate, that's an easy call to make. The team will even support you in the verdict: “Finally, you got rid of that (toxic) person!”

On the other hand, we often work with great developers who are just anti-social or apparently selfish. In functional teams, all members understand that the team comes before the individual, so there might be no space for individualists. However, many developers who don't operate well in groups can certainly learn how to do so with appropriate training and career goals (help them find collective goals that will also benefit them individually). Try your best to help them before taking the easy way out.

Performance is also a factor that might make you wonder whether you should fire someone. As Team Lead, make sure to take ownership of the problem and exercise

every possibility before firing someone for performance reasons: training, support, pairing, etc. Otherwise, you might be damaging the team culture—building up fear through a ”survival of the fittest” model—and losing potential future high-performance members.

Sometimes the need to lay people off is completely out of control. It could be, for example, budget issues unrelated to the team's performance. In a rational analysis, everyone would accept this fact, reduce the size of the team and move on. However, we're mostly driven by emotions. Always notify the team of such events and their reasons. Don't hide information or lie. We've seen shattered cultures where people have been fired and the team only found out much later when asking why a person stopped showing up. A common reaction you want to mitigate is: “Oh no, who's next?!” Also, make sure to not let the team turn against the business. There can be no “us against them.”

A proactive leader can allay most damage by taking responsibility: “Mark was a great asset to this team and we're all sorry to see him go. It was a tough call and the business made sure to give him all the support in this transition. I take the blame as I should have been more involved with the business to see this coming and found ways to help them discover better solutions and protect the team. I'm sorry to let you all down. I'll do better next time. Now it's our time to do our best to help the business get up to speed so we can prevent this from ever happening again.”

Regardless of the reason for the layoff, this part of the job is never easy. Our recommendation is to be transparent with the team and tell them the truth as soon as possible.

Part Three

# The Codebase

# Introduction

In our journey we've met many developers who believe their challenges are unique.

“Good practices, modular architecture, clean code and automation all sound good, but you don't know my product/budget/team/boss/manager/ . . . It's **impossible** here.”

Although we all face different challenges, we are all trying to solve the same problems. When we ask developers about their challenges, we always hear the same hidden issues that can be outlined as:

1. Dealing with the asynchronous nature of code quality.
2. Welcoming change.
3. Saying NO to impossible requests.
4. Estimating more accurately.
5. Being productive.
6. Maintaining a fast and constant pace.
7. Automating tasks.
8. Working effectively as a team.

While it's true that we all face different technical challenges, we all share the need to produce sustainable codebases that enable us to achieve the positive traits above.

## Dealing with the asynchronous nature of code quality

As developers, we try to write flexible code until an unexpected change breaks all our assumptions. We keep looking for shortcuts, and we often don't feel the pain of accumulating debt over time until it's too late.

How can we measure code quality? Practices like code review and linting are often biased by the reviewers' perception of ”what quality code is.” Code review and linting can be great to keep the codebase consistent with the team's style, which is important, but can't actually tell us “is this good code?” Another issue with code review is that the incentives to push code are greater than the incentives to block code (even when it's bad).

If we're biased towards our code quality, is there anything we can use to measure it? The best tool we know is that: “Good code survives the test of time.”

Why is *time* the best tool to measure quality? Because code changes over *time*, and we

*want* the code to be easily changeable over *time*.

When we write code to work just *now*, we can't know for sure if it will change gracefully over *time*. Code review and linting normally ask: ”according to our own biased code quality measures, is this code fine for *now*?” However, the dilemma we all face is: how much should we care about the future? The problem is that we don't know what will happen over *time*. And often, the future (long-term goals) can be as important as the present.

## Welcoming change

Software development is a fast-paced industry. Businesses that try to stay competitive keep trying to innovate, or at least stay up to date with the competition. Developers have a massive role in this fast-paced industry as we're the ones on the front line and our work can either enable or prevent the business from achieving its goals. By neglecting the business goals, we jeopardize our own careers.

Our job as developers is often to produce flexible software that will welcome requirement changes and survive the test of time.

And even when your business requirements don't change, you still have to deal with platforms, languages and many other tools constantly being updated.

Software should be *soft* (flexible/changeable/malleable), otherwise we would call it hardware (rigid/inflexible). However, it's not easy to produce soft software. The constant rate of change is exciting—and a reason why many become developers—as we're continually working on new challenges, but it's scary at the same time. The biggest issue is that the future is unknown.

Because the future is unknown, to be able to welcome change we need to protect the codebase, the team and the business by preparing for the unknowable and establishing good habits.

## Saying NO to impossible requests

To make matters worse, businesses keep setting up deadlines without consulting developers. Even when developers are consulted, they are often afraid to speak up.

By saying yes to impossible deadlines, we are not welcoming change in the codebase. We may only get away from dealing with the present conflict. However, the codebase has no feelings and will send a big ”NO WAY!” back to us.

In a utopian world, trust is blindly given to us. In the real world, we often have to earn it first. We know saying no is hard, but earning the trust to do so can go a long way towards a prosperous collaboration. With earned trust, you won't have to say no. You'll get your boss to say yes.

## Estimating more accurately

A friend of ours, Megan, had a wedding to attend in two weeks time and sent her dress to the dry cleaners. She asked when they would be done with it, and they replied: "We'll call you when it's done." But will it be ready before the wedding? Again, they replied: "We'll call you when it's done." When we tell this story, most people find it preposterous. "This is unacceptable!" Without an estimate, she can't know if she'll be able to attend the wedding, so she's left to make suboptimal contingency plans (e.g., if it's not ready up to two days before the wedding, she must buy a new dress). Even an imprecise estimate would be more helpful, such as: "It'll be done in up to ten days."

This is a problem many companies face when planning for releases, marketing or budget when codebases get in the way of estimating. *When will the software be ready, so we can schedule a marketing campaign?* "We'll call you when it's done."

But is it even possible to *accurately estimate* the duration of code changes? Probably not. After all, estimating is *guessing*: "I estimate (guess/hope) it'll be done in three days."

A better question is not if it's *possible,* but: "is it valuable to *estimate*?" or "is there anything we can do to *improve our guesses*?" While we don't like to estimate our work (and even create movements like #NoEstimates), it's important to be able to make some *informed guesses*.

When estimating, we often need to balance accuracy against precision. In our opinion, Robert C. Martin explains it well: "The best estimation is 'I don't know'." Then he goes to elaborate that, while "I don't know" is *accurate* in its truth, it's not very *precise* in the timeframe.

We can also be very precise, for example: "It will take two days and two hours," but not very accurate if we aren't able to meet the proposed deadline. Underestimations are common because teams usually don't accommodate for unpredictable scenarios (people get sick or go on holiday, systems break unexpectedly, bugs, a pipe breaks and floods the office . . . )

We can go to the other extreme and overestimate: "It will be done at some point from

now to eternity." This is accurate since we can meet that deadline, but not useful since it's not very precise on when the task will be finished.

People suggesting or asking for #NoEstimates happen mostly when businesses ask for estimates just for the sake of pressuring developers. If there's no emergency, estimates can be irrelevant or just "nice to have." At any other time, we must do our best to produce good estimates.

Keep in mind that, when your team can't estimate, "I don't know" is the right answer. However, don't stop there. Take time to research, experiment or prototype some ideas, and then come back to provide a more precise and accurate estimation.

Ultimate accuracy may be a futile goal, but functional teams produce codebases that will help them understand the team's level of uncertainty around the work to be done and improve the accuracy of their estimations as much as they can. This is to better accommodate planning and promote trust within the business.

## Being productive (now and then)

Productivity, just like code quality, is another factor of time. Our productivity *now* may take away from our productivity *tomorrow*. “I finished this task in record time. One singleton and three if statements. Since it's such a simple implementation, no tests are needed. It just works!” This productivity shortcut will send a signal to other developers that it's fine to take the easy route in this part of the codebase (or, indeed, anywhere in the codebase). Then the code mess grows and grows until it paralyzes the team.

A sustainable codebase goes a long way towards keeping us productive while we change it. However, a sustainable codebase does not guarantee its own sustainability. So, as we change the codebase, we might be either making it cleaner (enabling us to stay productive) or messier (preventing us from staying productive).

Our future productivity often pays a huge price for our present shortcuts (debt). In the same way, our current productivity may already be paying off huge interest for the shortcuts of the past. “Let's rewrite the app. We'll be much faster/better next time.”

What we can learn from all this is that productivity over time, without measuring the quality of work, is not a great metric. It may seem tempting to set it as a leading indicator, especially when a product team is dependent on your deliverables; however, it doesn't show you the big, and most importantly, real picture. Think of it as measuring the GDP of a country or the revenues of a company without accounting for any liabilities such as their debt.

A sustainable codebase developed with discipline at every step, regardless of task complexity, goes a long way towards keeping us productive now and in the future. It can make our productivity pace fast and constant over *time*. A constant pace can make us more reliable and help us earn trust and influence within the business.

## Maintaining a fast and constant pace (can we even track this?!)

What does it mean to maintain a fast and constant pace? In short, it means that if a feature takes two weeks to be implemented at the beginning of a project, it should take the same amount of time to be implemented two years down the line.

Can we track productivity when working on a codebase? Speed is often used to track the team's productivity. However, we often see it as a metric to punish underperformers. And what good would that do to team morale? Not much, we're afraid.

Productivity should not be used as a stand-alone metric. It must always be compared to something. “Shannon is a productive developer.” Compared to who?! To other team members? To other developers in the world? To non-programmers?

A sustainable codebase should yield analysis like: “Shannon is as productive as she was three months ago.” Messy codebases might yield: “Shannon is not as productive as she was three months ago.” After which, in a functional team, we could ask: “How can we help her get back up to speed?”

## Automating expensive tasks

As developers, a huge portion of the code we write is to automate tasks. Automation is often the product we're building in one way or another: payroll systems, stock control, one-click buy or automated email chains.

Going further, automation can be more than the product we create and sell. Automation can be used to reduce the cost or risk and enhance the team's operation. For example, automated testing, when done right, reduces the cost of manually testing an application and the risk of regressions. Reducing the cost or risk is a desired trait, but still a very shallow view of automation. Automated tests, for example, don't just reduce the cost. They often *enable* testing the software.

As codebases grow, there are many paths to be tested. They can eventually reach a point where it's humanly impossible to manually test all branches. At that point, automation

is not a matter of reducing cost. It's a matter of being able to perform the task with the time and resources we have.

By missing the opportunity to automate tasks, we might be preventing ourselves from actually performing the task at all (and reducing the value proposition to our customers). The earlier we automate tasks, the better we can safeguard against losing control.

## Working effectively as a team

What came first, the chicken or the egg?

We face a similar paradox when developing software in a team. What came first, a sustainable codebase or a functional team?

We all want our codebases to be clean, well architected and easy to build, extend and maintain. We want this because it allows us to deliver great products and continually welcome change. However, until we agree on how to get there, this might be hard to achieve.

Again, does a clean codebase with great architecture enable us to be functional and productive, or does being a productive and functional team enable us to develop a clean codebase with great architecture?

When we believe that clean codebases are what enable us to be functional and productive, what should we do when we see ourselves in the middle of a mess? That's easy to answer as we have many examples. When the perceived problem is in the codebase, it feels external as: “It's not our fault, it's a code problem.” We often see excuses and say: “The business didn't give us enough time,“ or “The old developers did a bad job, so I can't be productive.” Sometimes we use even more desperate measures such as: ”Let's rewrite the app” and ”code freezes to tackle tech debt.” Most measures aim to solve short-term problems and often the codebase won't survive the test of time.

On the other hand, if we believe that being a productive and functional team will enable us to develop sustainable codebases, we always find time to work on both sustainable and messy codebases. “The problem is with us, and we'll fix it.” “Not enough time? Let's solve that.” “Some developers are doing a bad job? Let's help them improve.” When working with sustainable codebases, functional teams will keep them as is—clean and flexible. When working in a messy codebase, functional teams will—step by step—improve it until it becomes sustainable.

So, what came first, a sustainable codebase or a functional team? We prefer to believe in, and recommend, the latter.

## Is the glass half empty or half full? Obstacles vs Opportunities

Most companies and development teams face the above challenges; however, only a few people have the means to effectively tackle them. Where some see obstacles (negative) others see great opportunities (positive). Leading the effort to create sustainable codebases gives you:

The opportunity to improve and help others improve.
The opportunity to grow and help others grow.
The opportunity to gain more experience.
The opportunity to influence the collective outcome positively.
The opportunity to learn and teach.
The opportunity to master skills.
The opportunity to become a leader.
The opportunity to achieve a prosperous and fulfilling career.
The opportunity to become an Essential Developer.

# Making the most out of opportunities

For a long time, we have been observing common traits and tracking developer patterns for both good and bad codebases, regardless of their open-source or proprietary statuses. As mentioned previously, a good codebase allows the firm to fulfill its vision perpetually, opposed to a bad one where it will prevent the firm from moving forward with its plans at the desired speed and growth.

There are extensive studies (e.g., *Accelerate*, 2018) that positively correlate automation, shift-left security, and modular software architectures with technical excellence, profitability, customer acquisition, team happiness, less burnout and more. To be proficient and successful leaders, we must understand *software development* on a whole new level.

Every import statement, every new dependency, even a single new line of code can potentially damage the codebase, which makes the whole Development Team responsible for the short- and long-term sustainability. Since we're all responsible, it's important to remember that developing software is a social activity, so it also has social challenges.

We cannot and don't even dare to suggest we can predict the future and prepare the system to be 100% resilient to change; however, there are behaviors that will most definitely impede progress. It's quite remarkable how often teams choose convenience in their decisions over sustainability. For example, it's evident that automated testing, when done first like in TDD and BDD, not only helps to mold and improve the design of the system but, at the same time, creates a way of asserting that our code is doing what it is supposed to do at least twice. Such methodologies may almost sound vital for a business to thrive in the software industry. “How would a dysfunctional business compete with a functional one?” Yet, we've seen many teams and projects that don't effectively follow their chosen methodology, and evidence from the codebase metadata will indicate this.

By analyzing the codebase metadata, we minimize opinions and human perception and rely on hard facts like build times, architecture and code itself, as well as its evolution during specific timeframes. By gathering and analyzing metadata, we can see where and when things started going wrong, what kind of principle violations exist in the codebase and get an idea, depending on the current state of the codebase, of how it will perform in the future. The metadata can provide signals through various metrics and alert us to the health of the codebase and whether it remains an asset or is slowly becoming (or has already become) a significant liability for the firm.

It's common for developers and businesses to seek a small group of metrics that will be used as the Holy Grail of predictions, control or health analysis. **In our experience, such an ultimate metric doesn't exist.**

> *"Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes."*
>
> *—Charles Goodhart*

Take code coverage, for example, a measurement that is built into IDEs which whole deployment workflows depend on in case it doesn't meet a specific criterion. Code coverage by itself will not tell the truth about how much of the system has been tested because individuals, incentivized to make the code coverage metric reach a specific percentage, can tweak it without necessarily covering all system behaviors.

Instead, we would like to propose a collection of **indicators** for specific traits that have been observed in practices used by developers who produce good (and bad) codebases. These indicators serve as the framework of guidelines **signaling** the strong and weak points of the codebase.

Imagine if you could have access to the lead voices in the industry and consult them about your weekly output. Their advice would then point out any improvements you could perform or any bad decisions that could harm the system in the future. Most importantly, they could justify their answer and let you know why such outcomes could take place. Our collection of key indicators along with the team's experienced interpretation helps with that; it consults the team on a granular level as to whether or not they're following good design and programming guidelines. Although this collection of indicators can be automated, the team should aim to make the principles behind the indicators part of their everyday operations, so they can recognize and understand the impact of each line of code and proactively produce better results (rather than a reactive approach of relying on indicators after the fact).

It's imperative to understand that the following indicators must be addressed and monitored *together* to get the most helpful picture possible for the current and future states of the codebase.

Finally, keep in mind that software quality and sustainability principles apply to any programming paradigm (e.g., functional and object-oriented). When we mention

"component," we mean functions, modules, classes, protocols, interfaces, data structures, dependencies and any language/platform or component-like types in the codebase.

# Indicators

## Overall test lines of code per production lines of code

This is a snapshot of the whole codebase revealing how much work has been done in a test suite. Similar to code coverage, it can't be accurate about how much of the actual codebase has been tested, or whether tests were written first or not. However, it shows the effort the team puts into testing.

We take as a fact that the test lines of code, in the long run, may surpass production lines of code because with every conditional expression (if, switch and guard statements) we are looking at least at two separate tests covering all cases. In the case of nested conditional expressions, the number of tests grows exponentially which results in a lot more lines of code than the production equivalent. Such *combinatorial explosion of conditional state* can and should be avoided with better design choices (e.g., polymorphism).

On the other hand, there might be parts of the codebase that your team have decided to test with another strategy and not unit testing. Perhaps you have found your app's UI is better to test with another automated strategy such as screenshot comparison, or even manually. Then we are looking at a surplus of production lines of code over test lines of code, which will influence the ratio.

#### Good for

- Showing the effort the team put into testing overall.

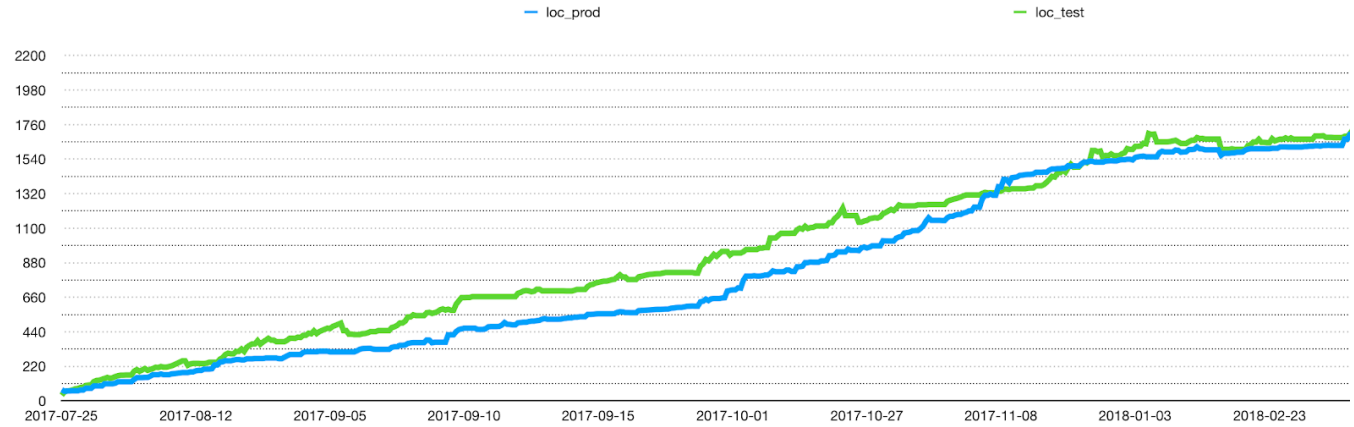
## Test lines of code per production lines of code over time

To assess the level of consistency in writing tests along with the production code, we can plot the ratio of each commit in a graph and let the graph lines tell the story, regardless of whether the tests were written first or last. Developers who do not practise TDD tend to write the production code first and come back to it, usually in the later days of a sprint, and only then add any missing tests.

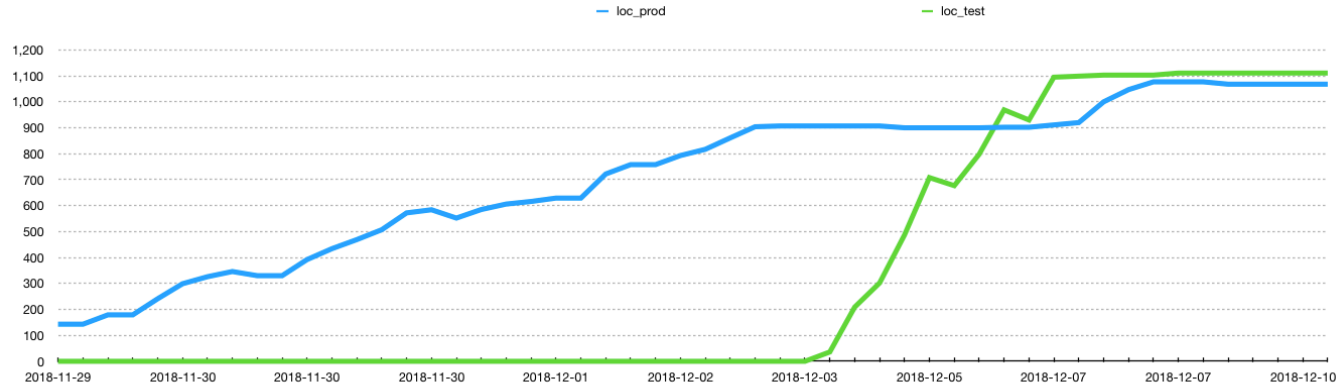
The test lines of code per production lines of code metrics should indicate how important automated testing is for the team and what the preferred way is of

performing it (first or last). If the test lines of code index is flat or with small spikes when compared to the production lines of code index, you know the codebase is not built with testing and an automated QA strategy in mind, which can be costly for the business as you move forward.

### Test first - example graph

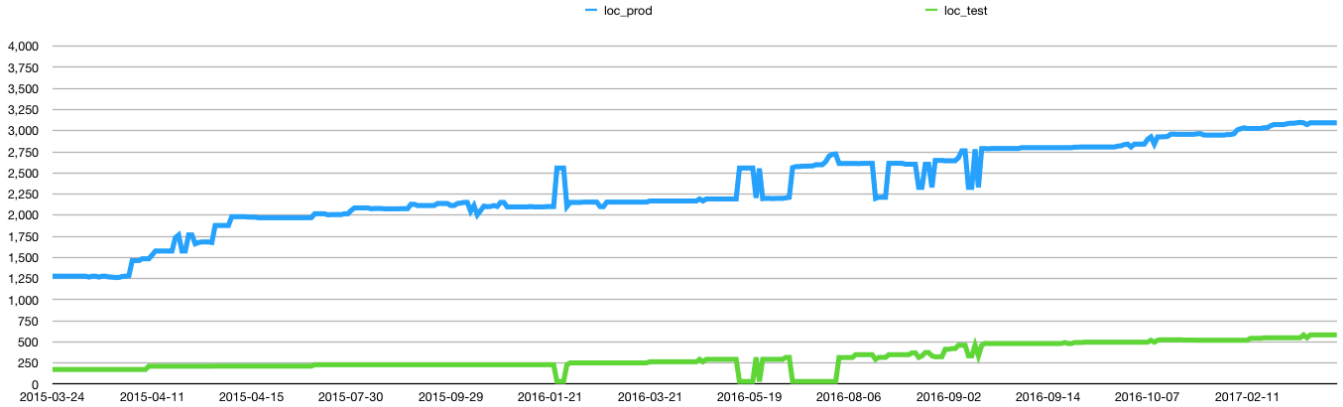



### Test after - example graph

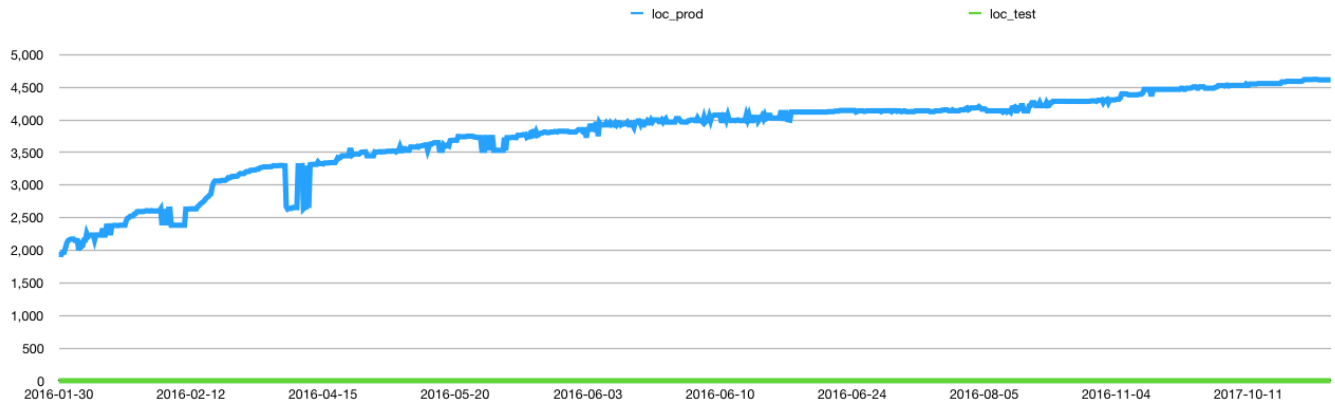



The above graph denotes the testable nature of the production code. Developers could easily add tests and reach a high ratio of production lines of code per test lines of code without making changes to the production side.

### Automated testing not being a priority - example graph



### No tests - example graph




#### Good for

- Showing the effort the team put into automated testing over time.
- Showing consistency in testing practices.
- Understanding if tests were written first (both lines moving at a similar pace) or last (tests loc stays constant as prod loc grows, then eventually they meet at the end).

## Lines of code per file

This is a measurement for code organization. A high ratio can indicate potential problems with the amount of responsibilities and dependencies various components might have.

Depending on the type of components that exist in a file, the ratio will vary. If, for instance, a file contains a single protocol with a couple of methods declared in it, then it would help the ratio move down. On the other hand, if there are files containing classes with hundreds of lines of code in them, then the ratio will move up. In any case, the higher the ratio, the higher the probability of lack of modularity and an open-close capacity for the codebase.

#### Good for

- Indicating a potential problem with modularity and open-close capacity in the codebase.
- Indicating whether the project is well organized.

## Indentations per file

Indentations represent a nested structure in an implementation. Some examples are

nested conditional statements, flow of control statements and closures. Such behavior can result in the infamous ”arrow”-shaped code, where each line is indented by tabs or spaces and would have to end each nested statement with a curly brace, resulting in the code forming an arrow-like shape (also known as ”The Pyramid of Doom.”)

```
class Example {
    function someFunction() {
        if someCondition {
            if someNestedCondition {
                invokeBehavior1()
            } else if anotherNestedCondition {
                for anItem in anEnumeration {
                    if anItemCondition {
                        invokeBehavior2()
                    } else {
                        invokeBehavior3()
                    }
                }
            } else {
                invokeBehavior4()
            }
        }
    }
}
```

Each nested statement increases the complexity of the system, which can result in more state management and a lot more difficulty in testing, or—even worse—making certain behaviors untestable.

When dealing with these components, we have observed that developers understand the problem of the arrow shape formation in their code, which makes them extract such statements in new methods in the same component. This option solves the arrow issue; however, it also increases the component's lines of code as well as hiding a logical flow of data, especially if the code isn't organized properly within the file. It would be much better to extract arrow code paths in new components where they can be tested and then composed as a whole with the previous ones. Unfortunately, this can be impossible in messy codebases because of all the dependencies involved.

Arrow-shaped code usually results in maintainability issues so we suggest you examine it carefully, as it can lead to fragile and immobile code.

**Good for**

- Finding excess complexity in components.
- Understanding what is hard or even impossible to test.

## Conditional statements per test file

As with the "Indentations per file" metric, the same principles apply to test files. Our tests will probably contain nested statements, especially when dealing with closure callbacks; however, this metric can give us a hint as to whether they are being used in other ways, such as conditional statements.

As with classes or functions, tests should also be responsible for a single objective. The arrange/act/assert structure helps us differentiate between flow of control and focus on just a single test case. In the event there is more than one case to test, we should opt to create a new test and explicitly test another branch of logic instead of including all the branches in a single test. Break down tests with multiple logical branching into multiple tests with only one logical assertion. The goal is to have a clearer view of what is being tested. Also, when a test fails it's much easier to understand the scope that generated such failure.

An easy way to identify tests that have more than one logical branch is to find tests with very long names (with *AND*).

1. *"It should return a valid token when the user is logged in AND return nil when the user is logged out"*

This test/spec can be better expressed as two separate tests:

1. *"It should return a valid token when the user is logged in"*
2. *"It should return nil when the user is logged out"*

**Good for**

- Understanding whether conditional logic is being separated in multiple tests instead of one.
- Showing consistency for following the single responsibility principle.

## Conditional statements per production file

Conditional statements imply the existence of state in the application. Of course, it's almost impossible, perhaps even wrong, to create systems without the option for other possibilities; however, their placement is quite important for the health of the codebase in the long run.

Utilizing conditionals to make decisions in all levels of the application complicates communication and data flows between the layers. That's why it is an excellent practice to isolate most conditional state in the highest level possible, so the lower level components don't have to make too many decisions, as they will act upon commands. For example, imagine your application checks whether there is a network connection available and alters the UI in case there isn't. If we were to check for the internet connection in the view, perhaps we would have to check multiple times for that state. Even worse, we might have to duplicate checks to the network state in many distinct views. Instead, if we were to check or observe the network connection availability in a high level implementation in the app, our views would be intact, knowing nothing about the network and rendering only what the higher level components tell it to.

Our goal is to get informed about a simple question: do a lot of files have conditional statements or do a few files have conditional statements? We aim for the latter.

#### Good for

- Understanding how conditionals are spread across our application.

## Method and property count per interface/protocol

This metric gives us an average of the protocol sizes which is a good indicator as to whether or not we are respecting good design principles. To achieve concise and reusable abstractions, we should avoid stuffing all method declarations into a single protocol. Thus, we should aim to separate the interface/protocol's methods into as many interfaces as needed by their specific clients. By doing so, we can avoid concrete implementations having to implement any unnecessary methods, the Liskov Substitution Principle (LSP), while decoupling clients from methods they do not need, the Interface-Segregation Principle (ISP).

We should always aim for a small number of methods and properties for interfaces throughout a system.

Ideally, one method per protocol. It's much easier to deal with dependencies that do

only one thing. The more methods, the more possible paths there are in the code. Breaking down methods into separate protocols allows us to decentralize implementations. However, there are times where it makes sense to add multiple method declarations to a protocol. The Single Responsibility Principle (SRP) is a great guideline. Ask yourself: **are all methods related and responsible for one and only one responsibility (e.g., cache control: add/remove)?** If it's a no, you should probably create a new protocol unless you have a better reason not to.

**Good for**

- Identifying good design principles, developer discipline and consistency.

## Method and property count per interface/protocol over time

Another important metric is a variation of "Method and property count per interface/protocol" where we can observe how protocol declarations change over time (component history).

It's not uncommon for developers, in order to accommodate current needs, to break the contract the interface offers and add additional behavior to facilitate their task. This behavior should be considered harmful as it can break not only the interface segregation between modules and components, but also the Liskov Substitution Principle, resulting in a more rigid and fragile system.

**Good for**

- Identifying good design principles, developer discipline and consistency.

## Boolean flags as parameters or properties per module

Boolean flags as parameters or properties in components are created to manage their internal state. Such practices can lead to unnecessary complexity which, in turn, can lead to unmaintainable and untestable code.

This metric can give you a hint as to whether your components are managing too much state internally and should split their responsibility into one or more components and redirect the condition to a higher level.

**Good for**

- Indicating whether the system can become potentially unmaintainable and

untestable.

## Private properties count per public and internal properties count in modules

This metric can help us identify what percentage of our component properties are visible to the rest of the system, internally or publicly. Ideally, we would want a small percentage of them, as, if our properties are exposed, they can be used by other components to query their values and create unnecessary complexity with conditional statements. They can also obstruct the dependency inversion between components, as their users will be able to cross boundaries.

In other words, very permissive access control (e.g., public properties) can increase the probability of modules depending on internal details of other modules, which may create tight coupling and make things harder to change (detail changes may break external modules). Hiding implementation details is a key ingredient for forming modular and extensible systems.

As a guideline, make things private until there's a very good reason not to.

### Good for

- Indicating whether the system **may** unnecessarily expose implementation details.

## Comments per production file

This metric encapsulates commented out code and annotations such as “TODO,” “FIX” or other instructions in our components.

A high value for this metric can signify overuse of these annotations which should be avoided in a team scope project. The reason we recommend removing them is simply because there should be a universal source of truth for work in progress or work required in the future (GitHub issues, TFS, Trello, JIRA, etc.). By using such annotations, we increase the probability of diverging from a single source of truth, like a project management tool, which results in a lack of full transparency with members of your team and others.

Ideally, our codebase should contain zero reminder-like annotations or commented out code, indicating that the work or fixes required are being held in a place visible to other team members, and no instructions are needed to describe any processes in our code, as everything should be working optimally.

**Good for**

- Indicating if there are *issues/tech-debt/work to be done* outside shared project management tools (not tracked or visible to the wider team).

## Comments per test file

The same principles that apply to the "Comments per production file" metric also apply to "Comments per test file." However, there is an important distinction in production and test files when referring to commented out code.

Any commented out code in the production code will simply not compile and will not be included in the product. Commented out tests, however, signify missing checks in the system's behavior. When deadlines are approaching and pressure increases it might be tempting for some teams to defer the implementation or rewriting of these tests to the future, after they have made the deadline. This behavior should be considered harmful, as clearly not all automated checks have been performed to the system, thus it increases the probability of deploying faulty behaviors to customers.

**Good for**

- Indicating potentially untested behavior.
- Indicating if there are *issues/tech-debt/work to be done* outside shared project management tools (not tracked or visible to the wider team).

## Setup lines of code per test file

This metric can quickly show how developers set up their systems under test. A large value should signify that there is an excess of configuration code for the system under test to start for each test. A lengthy configuration usually signifies a lot of dependencies that are required from the system under test to start and function properly.

These behaviors can help us point out design issues with our components. For example, if component A has a dependency on the concrete component B, then to test A we may have to also create and configure component B. The configuration of B (which might require other components) can create an excess of test setup code. We can fix these problems by using dependency inversion, hiding concrete collaborator implementations behind an interface/protocol and using the real implementation in the tests only when actually required, and a reusable test-double implementation everywhere else.

#### Good for

- Understanding if there is an excess of dependencies and configuration for the system under test to work.
- Monitoring consistency with good design practices, such as components having a single responsibility or components being decoupled from other concrete components using dependency inversion.

## Lines of code per test method

This metric allows you to see the consistency level of your tests' structure. To make your tests more readable and easier to maintain, you can follow the arrange/act/assert or, if you prefer, the given/when/then paradigms. This means a test method should be small, concise and test one and only one behavior at a time.

The discipline of TDD helps with exactly that as its last step is the refactoring process. As soon as we have a passing test asserting the behavior we expect, we can commit the working solution and start refactoring not just production code, but also test code by simplifying setup, removing duplication and clarifying intention. By removing statements that are being repeated more than once, organizing assertion code and creating functions that hide complexity and allow extensibility, we can end up with pretty short, easily readable and comprehensible tests.

#### Good for

- Measuring consistency for structuring tests using best practices such as test code refactoring.

## Code Coverage

This metric shows the percentage of code exercised by tests. A high code coverage may indicate developers' consistency for writing tests along with the production code; however, it doesn't guarantee that all the production code behavior has been checked through the tests. In other words, a high code coverage doesn't guarantee the absence of programmer mistakes or missing assertions.

On the other hand, a low (or zero) code coverage implies that the team doesn't care for automated testing.

#### Good for

- Finding untested code paths.

## File count per commit

This is a very important metric showing the number of files committed each time which can signify modularity, loose coupling in the codebase and developer discipline.

A small value for this ratio usually dictates that developers work in small batches, focusing only on specific components and are not worried about their place in the rest of the system. This approach comes easily by doing TDD, where developers can usually commit after a test passes or a group of tests pass, resulting in these micro-length commits. By doing this, they ensure the system will most probably be able to build any revision, as well as easily rolling back to a previous version and undoing any potential mistakes.

On the other hand, a larger file count per commit means that more files are being edited on each revision, which increases the probability of one or more components or, even worse, modules, being edited at the same time. This behavior makes the system's reversibility much harder, as developers will be working with larger sized batches of requirements and potentially increasing the system's complexity.

**Good for**

- Potentially understanding the developers' level of discipline.
- Finding out whether the system is being developed in small or big batches.
- Predicting a level of system reversibility (undo cost).

## Commit count size per author

As with the "File count per commit," this metric measures the same consistencies and behavior levels, but it is predicated to each author. This metric allows us to identify which developers may be experiencing issues with their daily operations and allows us to help them out and improve both the codebase consistency and their skills.

**Good for**

- Identifying which developers may require assistance with good development practices such as working in small batches, TDD and modular design.

## Commit count per author

This is a simple measurement for the number of commits per author. This metric can be used within a range of dates, as well as to assess each developer's involvement.

It is crucial to note the value of this metric does not directly correlate to the productivity, contribution or efficiency of the developer. For example, it can be assumed that a senior member of your team has a high value for this metric; however, their daily tasks and responsibilities may change from time to time. They might be pair programming on another developer's machine or they might be involved in the design or review of the system, which are all tasks that do not necessarily require them to commit code in a repository.

Junior members, especially newcomers, may not feel comfortable committing their code multiple times per day, as they might wait for approval or review of their contribution from more experienced programmers.

This is a dangerous metric, often used to assess “who contributed more.” We only added it here to alert that it should not be used to measure developers' productivity or efficiency. By using this metric to assess team members' contributions, we might incentivize them to work alone or against one another—and that's an easy metric to game.

#### Good for

- Counting how many commits a developer produced (it means nothing per se).

#### Do not use for

- Measuring developer productivity.
- Measuring developer contribution.
- Measuring developer efficiency.

## File count per merge

This metric indicates the average size of batches developers work with. A large file count denotes that they create big-sized branches that can cause delays in the development and release cycles, psychological fatigue and burnout for the developers and increase in costs for the firm.

For further explanation, see the **Average Branch Lifetime** metric.

#### Good for

- Indicating when operation speed is about to fall.
- Assessing if operation speed has changed over a period of time.
- Assessing if developers are struggling over a period of time.

## Merge count per day

Merge count per day is a metric that indicates whether your team can sustain an automated release process such as Continuous Deployment. A small ratio, typically less than 0.5 (one merge every other day), denotes a lag between merges that, in turn, may suggest problems with the development cycle (see Average Branch Lifetime).

A large ratio, typically more than 1.5 (more than one merge a day), denotes a healthier development cycle, allowing developers to continue to work independently and in small batches for long periods of time.

### Good for

- Assessing whether the current development workflow could support Continuous Deployment and faster release cycles.
- Indicating when operation speed is about to fall.
- Assessing if operation speed has changed over a period of time.
- Assessing if developers are struggling over a period of time.
- Understanding if developers work consistently in small batches.

## Average branch lifetime

This metric can give us an understanding as to whether the team work in small batches and merge their contributions often. This metric should not be used for measuring the team's productivity, as the lifetime of a branch can depend on many factors other than the developer's speed of execution.

These factors include the integrity of the requirements provided by the product team, cross-team communication in case there are dependencies on other teams and timing of the calendar year—branches created on a Friday or before a holiday can have a greater lifetime than those created at the beginning of a cycle.

Depending on the version control workflow you use, you can exclude from this metric fixed branches representing specific environments (staging, develop, master) or branches that are never deleted, such as released-version branches. We should consult this metric for ”feature” branches that should be short-lived.

We should mention that this is a good metric to assess team morale and spirit. Larger branch lifetimes imply that work produced is not included in a centralized type of branch, such as master. By not merging to master often, the work the developers focused on will not be deployed, so will not get into customers' hands. This phenomenon can significantly impact the psychological state of programmers and, in many cases, they can experience burnout.

Moreover, work that is not merged often, meaning the end of a feature branch lifetime, can signify that developers have difficulty completing their tasks. With endless ongoing deadlines and demands, this kind of behavior can severely impact the psychological state of developers (e.g., impostor syndrome) and make them not only less productive but counterproductive, especially in the case of burnout.

The Average Branch Lifetime metric can signal potential bottlenecks in the development process. As Team Lead, you should know about and address them before they escalate and become a liability for the whole team.

#### Good for

- Indicating when operation speed is about to fall.
- Assessing if operation speed has changed over a period of time.
- Assessing if developers are struggling over a period of time.

## Build time per commit

This is a metric showing the accumulating time of builds per commit while plotted in a line graph. By visually displaying the build time data, we can better understand patterns in our development practices through the potential volatility depicted in the graph.

As a project grows and more components and dependencies are added to it, we can expect the build times to grow as well. A linear growth for the build times should be expected. When the graph shows an exponential growth, or even a break of the linear pattern that sees it entering an exponential one, we should consider this behavior harmful and something that needs to be addressed as soon as possible.

Each team should consult this metric and assess when it's time to try to bring this number down or at least find a way that doesn't impact productivity and team morale. Such a strategy may consist of breaking down the project into independent and decoupled modules and developing independently on them, breaking free from delivery mechanisms. By doing this, we would significantly decrease, but not eliminate

altogether, the time we have to build the project as a whole, which can give us a fresh burst of speed and break free from the previous daily slog. Often, build systems offer a comprehensive log where you can easily find such bottlenecks.

**Good for**

- Indicating unsustainable build times that will decrease productivity and team morale.
- Identifying patterns for introducing overhead from dependencies.

## Test time per commit

Very similar to "Build time per commit," this is a metric showing the accumulating test times per commit while plotted in a line graph. By visually displaying the test time data, we can better understand patterns in our development practices through the potential volatility depicted in the graph.

Unlike the building of the project, testing might become a "second-class citizen" when it takes too long to run. When this happens, developers have deemed testing a bottleneck because it reduces their productivity, so they stop doing it altogether. Of course, such a practice should be considered harmful as it exposes the team and the business to potentially huge risks (e.g., bugs, regressions, fear of change, accumulation of tech debt . . . ), showing complete disrespect for the Integrity and Economics of the team and the firm.

There are various factors that can affect a project's test times, so there are a lot of options for us to check and try to improve.

Following a lightweight implementation of the arrange/act/assert test structure can eliminate excess work and lighten our test times. Try removing unnecessary operations and refactoring your test cases to induce reusability with the help of properties, functions and assertions.

Another common cause of high test time values is expensive operations that integrate two or more components instead of testing them in isolation. For example, a component needing a real database to assert its behavior will result in higher test times than a component that asks for an abstraction of the database, which can be passed as a mock or fake implementation. The same holds for other delivery mechanisms like network calls and UI.

A metric that can help us identify whether our codebase potentially suffers from this

kind of problem is the “Setup lines of code per test file.” Fast test cases will typically have minimal or no setup code in them, meaning they won't require real configurations and implementations of components to load and contribute to the test times overhead.

These types of integration tests can be helpful; however, you can consider optimizing or moving them to a separate test suite. Of course, moving tests to a separate test suite doesn't limit us only to integration tests. As the codebase grows and the build times grow as well, we can start separating our code into independent modules. By doing so, we can separate the test suites as well, resulting in blazing fast times as the number of components will be significantly lower.

To avoid premature optimization, be sure to first measure and check to see if you have a speed problem.

#### Good for

- Indicating unsustainable test times that will decrease productivity and team morale.

## External dependencies count

A very large value for this metric should be considered alarming and, in some cases, harmful. There's a balance between reinventing the wheel and managing risk introduced by third-party dependencies. External dependencies might enforce suboptimal design choices, cause unrecoverable and hard to trace crashes in runtime, security vulnerabilities, licensing and legal complications and the evolution of the dependency might not match the pace of your project which can negatively influence the product road map.

When choosing a third-party dependency, first understand their quality assurance practices, how in sync they are with platform updates and how compatible they are with your project road map. A good starting point is to monitor the frequency of issues, issue resolutions, releases, support SLA and—in open source project—commits and merges to their repository.

Another good metric is the number and quality of tests that exist in the dependency repository. As we discussed before, a high code coverage shouldn't imply that the project is fully tested. Instead, we should look for other metrics like the ones discussed in this section, such as:

- Lines of code per file

- Test time per commit
- Setup lines of code per test file
- Comments per test file
- Conditional statements per test file

By taking the time to choose dependencies accordingly, you'll notice a decrease in dependencies count, as most common open source dependencies are unfitted to ambitious projects and will cost more in time and money in the long run. Many *convenience* dependencies can be easily recreated with your project needs in mind.

Although we want to avoid many dependencies, realistically speaking you will probably use various ones in your projects. The key idea to understand is that these libraries or frameworks can become a liability instead of the asset we initially thought they were. To protect ourselves, the team and the company from the downsides of such dependencies, we can decouple them from our systems by using dependency inversion. To do so, create an abstraction between third-party libraries and your application. This will keep the system modular, where components are agnostic of external dependency details, making them very easy and cost-effective to replace in case of future complications.

#### Good for

- Indicating the exposure to risk in the codebase and product road map to external actors.

## Recurrent file changes committed or merged together

This metric indicates a potentially hidden coupling between components. By identifying files that often change together, you may expose weak abstractions between components. Such a fact might not be harmful within a module, but it's alarming when crossing module boundaries that, on the surface, seem decoupled.

For example, if every time you add a new UI action you have to update the analytics module to track such action, even if there's an abstract interface between the modules, they're tightly coupled. A better abstraction should allow new actions to be tracked without requiring cross-boundary changes.

Weak abstractions will force developers to commit files, components and cross-module changes in the same commit or merge request.

#### Good for

- Identifying weak abstractions.

# Interpreting indicators is key

It doesn't take much to make an unmaintainable codebase or decrease the quality of a good codebase. In fact, it takes a lot of effort to create and maintain an outstanding software product. In our experience, it's not time that makes codebases and products rigid and faulty, but many tiny lousy decisions (shortcuts). Once these decisions happen, they are communicated through the code to other developers, and so on. This creates a vicious circle reinforcing wrong practices and lack of attention to what's essential to a sustainable operation.

Luckily, these can be monitored through a set of indicators that can signal where things are going. The values shown in the listed indicators rely solely on hard facts, so their interpretations within the business context is key. These indicators help you examine the past and evaluate the present state of the codebase. The values you should be looking for vary from team to team, as they depend on the environment, the platform, the product, the company policies and the systems in use.

You can study the results and monitor the indicators and indexes manually; however, we'd highly advise you to entertain the idea of automating the gathering and reporting of these metrics. Monitoring how specific values change over small and long periods and having the ability to be notified about these changes can be extremely valuable for you, your team and the firm, as it may be able to prevent the loss of time, money, team spirit, talent, and even emotional well-being.

Part Four

# The Software Product and the Delivery Process

# Introduction

In the software industry, it's common to look at the *codebase* as the *final product*, since *developing the product* is seen as the act of *coding*. However, the code is more like a blueprint. *Coding* can be better defined when we look at it as *describing the business requirements in a high-level machine language*.

In simpler terms, code alone does nothing, so it is not the final product of developers' work. The real product is running software that can realize the customers' and business' needs. So, in fact, there's much more to software development than just coding. *We are also responsible for shipping it to customers!*

Back in the day, we would most probably code, build and ship a software product as a physical item. Imagine Windows 95, for example. Customers would purchase the product as 21 floppy disks and install it on their machines. The product was delivered in "hardware!" Thus, it was *hard* to change it later (you would have to purchase extra floppy disks). Even though we called the products "software," they were so bound to hardware when shipping that you might as well call it hardware.

Over the years, the process of building and delivering has evolved significantly. We can now ship code over the internet and update customer facing software much more seamlessly. Even better, in many cases we can run the product on our own hardware and serve it over the web, so updates won't require clients to upgrade. Such advancements should make the act of coding, building and delivering software solutions much faster, easier and cheaper, and allow businesses to realize their ambitious visions. The dream of SOFTware should by now be possible. In reality, we see that our industry is still far behind in being able to deliver what's actually needed from businesses, and the cost of creation, maintenance, change and delivery is often even higher than before.

# SOFTware as an asset

Software products can be assets when they're running and realizing the business' needs. To realize the business' needs, the product must be accessible to its intended customers. Thus, the act of coding and building alone do not provide enough business value. Valuable products are high-quality running products converting positively to the business.

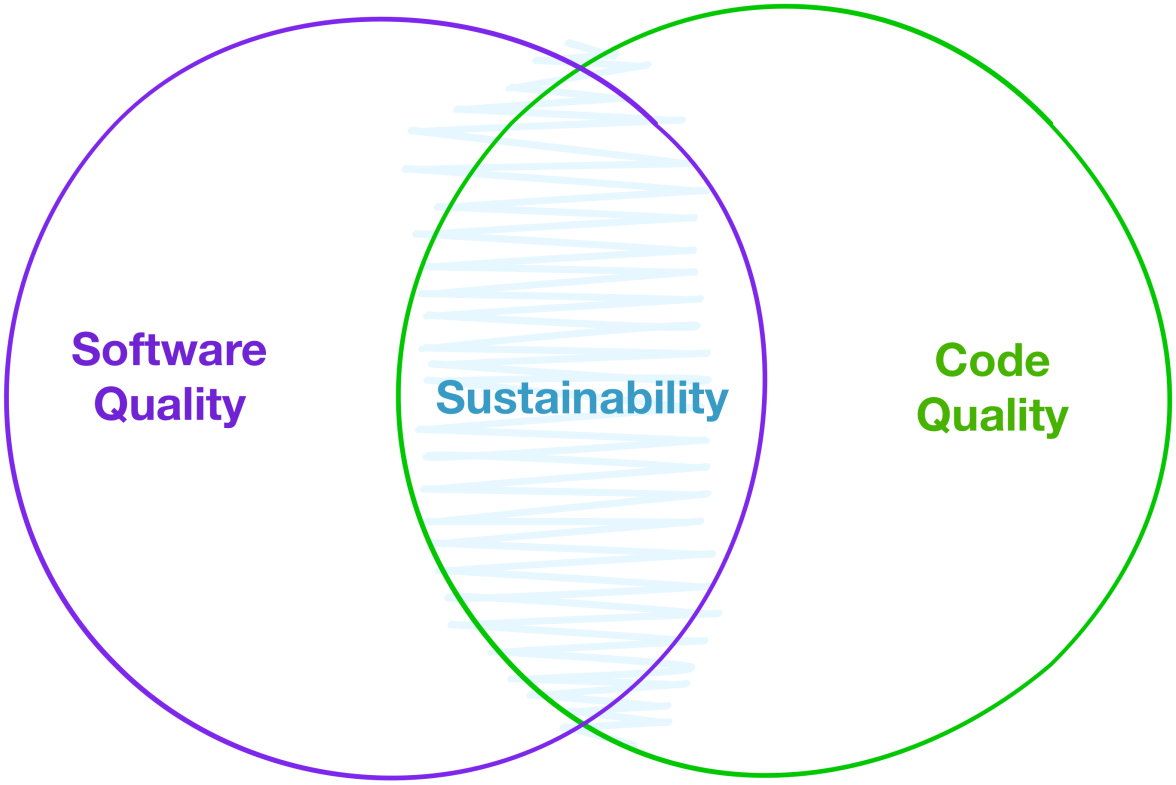
Many parties are involved in writing quality software, and only a few (developers) interact with the code. So every process around building a software product is

important to determine its quality, from onboarding to good communication and **delivery**.

In other words, the cleanest and most concise code won't matter if it doesn't fulfill its purpose (incorrect business logic) or doesn't do its job well for the customers (slow, crashes, hard to use).

On the other hand, we've seen quality software (from the business/customers' point of view) made from pretty bad code. However, bad code eventually becomes unsustainable (a liability).

> *"At the intersection between software and code quality the code sustains the business both now, and in the future."*
>
> *–Mark Seemann.*




In high-performance teams, **the act of shipping a high-quality software product happens seamlessly, several times a day**. To do this, the team should understand the economics of the software delivery process. Seamless quality assurance, continuous integration and continuous deployment can make the delivery process exceptionally fast and cost-efficient. *To do so, automation is mandatory*. Such a practice is one of the highest contributions a team can provide to the business.

In this section, we'll explain essential elements of the software deployment process along with indicators to track the health of your build and delivery system.

# Indicators

## Build and delivery time

Build and delivery time are two of the most important build metrics to pay attention to. If it takes an hour to build and deliver your application, it means it would take at least one hour to fix an issue in production. Of course, this kind of lead time is unacceptable in most businesses, so the quick and dirty answer to the problem is to disregard all of the tech team's build processes and push a ”fix build” of the application, bypassing all the automated delivery pipeline safeties. This may bring newer and bigger problems.

There are hard limits (e.g., processing time and network), but, in a healthy automated pipeline, it should take a few minutes to build and deliver the software. If possible, aim for seconds.

### Good for

- Indicating there’s a need to invest time to improve the build pipeline.

## Merges to master per day/week/month

For this metric, we assume that a team pushes stable versions of their codebase to a master branch, although the same principles apply for any branching strategy that aims to merge developers' work into a main branch or trunk. This metric tracks the count and frequency of merges to master over a specific period.

By tracking the count and frequency of the merges, you can monitor the consistency and collaboration of your team. Merging to master can be a complex and multipart operation, since merging code is an activity performed by everyone working on the codebase. The less often we merge to a master branch, the higher the probability there are too many bottlenecks (e.g., conflicts and lack of automation) in the development, build and delivery process. Ideally, a team should work in small batches and merge code several times a day.

If the delivery cycle suffers some missed deadlines, you can consult this metric to understand when things started to slow down. Utilizing the rest of the build and codebase metrics, you may get a better understanding of examining the past

contributions in the codebase and triangulate potential problems.

**Good for**

- Indicating the team might have collaboration issues that might impact the delivery plans.
- Indicating if developers work consistently in small and concise batches.

## Merges to master as the team grow per day/week/month

This is a leading metric measuring the sustainable growth of a team.

Without stellar processes in place, as more contributors are added to a codebase, the more difficult team communication will become. More developers signify more human resources for developing features; however, without the proper understanding of what is required by each developer and how their work will integrate seamlessly with others', more people can result in less productivity (e.g., fewer merges, fewer releases).

A simple way to be informed of the team's sustainable scaling is to measure throughout specific periods the number of merges in the main branch per the number of active contributors in the codebase. A healthy team should increase the number in this metric as new developers join the team (it might decrease at first during onboarding). If the ratio declines over time or fluctuates wildly, this is a good indicator of possible broken communication methods and a lot of room for improvement.

**Good for**

- Signaling if the team can scale sustainably.
- Signaling good or bad communication processes in the team.
- Signaling if a team practises continuous and independent development.
- Signaling if a team is onboarding new developers accordingly.
- Signaling long-term modularity in the system design.
- Signaling possible conflicts in merging strategies.
- Signaling if developers work consistently in small batches.

## Time from merge to master to production

Can we deliver value several times a day?

This metric targets the efficiency of the team's deployment strategy. Assuming that there's a master branch where any code contributions end up before the new version of

the product goes live, then we should aim for the least amount of time from the latest commit until the product is live.

Pressing the button for an app to go live can be a stressful proposition for developers, as the risk of blame for shipping a problematic version may loom in our heads. Instead, we can employ an automated proactive strategy for deploying the latest version of our product once it reaches the master branch. We can automatically perform all the necessary checks beforehand (e.g., running all tests) and enrich our process as we learn more about possible errors that may occur.

#### Good for

- Understanding deployment frequency and efficiency.
- Indicating potential room for improvement in the automation process.

## Time from idea to production

Regardless whether the idea is for the first version of a product or a new feature, time from idea to production measures the efficiency of your team and all other teams that might be involved. There are so many inputs to account for when building something new, including requirement analysis, communication of responsibilities, independent development, quality assurance and many more, as well as countless subprocesses for these.

Of course, all the above aspects are tied to the cross-teams' leadership and collaboration skills. If there are bottlenecks in the development or release processes, this metric will probably expose them over time.

If this metric decays, we'd advise you to go over the communication protocols between all involved teams and assess how to promote decentralized control to avoid bottlenecks. We should give more autonomy to our team members, as, if there are a lot of ”approval committees,” we can't be very agile.

A high value for this metric, especially with a low rate of regressions, indicates tremendous flexibility for the business, as it can strategically explore new trends or ideas with a low cost, both monetary-wise and technical debt-wise.

#### Good for

- Indicating the business capacity for strategic exploration.
- Exhibiting confidence towards the business and undertaking propositions with higher risk/reward.

## Number of releases over time

Assuming that what is released reflects the vision and specifications of the product, the number of releases over time is a metric for roughly assessing overall business productivity on a macro scale. Number of releases over time can give a good indication as to whether there are any overall problems with how the software product evolves.

In our experience, one of the technical leading factors that can damage this metric is when the coupling between modules of the system's architecture is high. If this metric's index starts to tank (no releases in a long period), we'd advise you to consult some of the codebase metrics and try to get a better understanding of what started to go wrong and when.

For example, you should monitor in unison:

- Overall test lines of code per production lines of code.
- Code coverage.
- Conditional statements per test and production file.
- Method and property count per interface/protocol over time.
- Boolean flags as parameters or properties per module.
- File count per commit and merge.
- Average branch lifetime.
- External dependencies count.
- Recurrent file changes committed or merged together.

We recommend you investigate the proposed technical metrics to find bottlenecks in your team, but also investigate the overall business process with other Team Leads.

### Good for

- Assessing business productivity on a macro scale.

## Number of times rework was necessary for security reasons

It's not uncommon to see teams defer or completely overlook security tasks when developing new features because they “slow down development.” Security needs to be taken extremely seriously; you should not let it become a burden in the release process. By flagging failed builds because of security issues, you can keep a log of all the instances that such a problem occurred and try to fine-tune your release process.

If security barriers become an issue, we recommend you train the team on security topics (ask for help from the security team to find out common mistakes) or, even better, make security part of the development operation. Security should be a concern in every stage of development, not just at the end. Instead of running “security checks” at the end of development, make sure to adopt a proactive security-first mindset (also called ”shift-left security”) within the team.

**Good for**

- Indicating if security has become a bottleneck in the release process.

## Number of deployment steps

Task segmentation is an essential aspect of the deployment lifecycle. As in various branches of project management and software development, fast feedback enables the team to iterate and refine the product. So instead of having one task covering all aspects of the deployment process including linting, reporting, testing and building, we can gain more information and control by creating a workflow that chains specialized units of work sequentially or in parallel. Monitoring and profiling should become easier as tracking and measuring the performance of the tasks can happen independently.

**Good for**

- Monitoring and measuring the deployment pipeline's tasks.
- Assessing modularity of the deployment process.
- Assessing the cost of individual pipeline steps.

## System health checks per day/week/month

System health checks are essential to increase confidence in the product by proactively monitoring and alerting the team to potential issues. Ideally, the team should check the system health several times a day, under low and high stress, to understand how the product behaves out in the wild. Since it can be a costly and hard task to perform manually, automation is vital.

We can learn a lot about the product when it's used by real customers. A system health check can combine log data from multiple sources (e.g., analytics, crashes, usage, events, etc.) so it's easier to preemptively alert and find the reason for problems.

When issues occur (or are potentially about to occur) in the product (e.g., a downtime) the tech team should be alerted by the automated health check system. If they aren't

alerted on time, they should learn from the issue and improve the system check rules to alert them before similar problems happen in the future.

**Good for**

- Indicating if the tech team is proactively monitoring for potential issues.

## Recurrent system health issues per day/week/month

Recurrent system health check issues that are not solved will quickly be ignored by the wider team and what once were small warnings can potentially become big problems.

If a recurrent alert is really not an issue, update the system health check rules to remove the false alerts.

**Good for**

- Indicating if the team is solving issues accordingly.

## Delivery system incidents per day/week/month

This is a simple metric that measures the reliability of the delivery system and if and when it becomes a bottleneck for the team. Teams often try to be proactive about setting up certain mechanisms to facilitate the deployment process, but might not accordingly maintain such systems over time. Take a Continuous Integration Pipeline, for example. Perhaps the pipeline's happy path builds the system, runs the tests on various environments, lints the code, logs and reports the results and updates various clients who are listening. To do so, the pipeline contains various rules, dependencies and web-hooks that might be fragile and flaky, resulting in the build system's failure. In turn, failure of the build system translates to lost resources, mainly time, money and developer peace of mind.

The build system incidents indicator can easily show you an overview of the health of your system process over time. It serves as a proactive way to easily catch the problem before it escalates in the team's daily life and blocks them from producing their best work.

**Good for**

- Measuring if build failures occur too often.
- Indicating there’s a need to invest time in improving the build pipeline.

## Number of bugs over time

No one wants bugs in their applications, but, inevitably, some mistakes will occur. Classify bugs into categories so you can find out which areas of the application are more problematic. For example, the team may realize they are shipping too many bugs around multi-threaded parts of the code. It could be wise to invest time to find a better threading strategy. If the team are unable to find better solutions, training is a good alternative.

#### Good for

- Indicating areas of the code that can be improved.
- Indicating if the team requires further training.

## Number of regressions over time

Too many regressions (repeated bugs or mistakes) indicate the lack of an automated testing strategy to prevent mistakes from reoccurring.

Ideally, every reported bug should be fixed along with an automated test to prevent it from ever happening again. As a guideline, first prove the bug exists by writing a test (it should fail). Then go on to fix the bug and run the test again (it should pass).

#### Good for

- Indicating missing automated tests.
- Indicating if the team requires further training.

# Outro

> “Success is nothing more than a few simple disciplines, practiced every day; while failure is simply a few errors in judgement, repeated every day. It is the accumulative weight of our disciplines and our judgements that leads us to either fortune or failure.”
>
> —Jim Rohn

When building a team from scratch, you might initially need to perform most of the responsibilities described in this book. However, don't forget that to become an effective leader, you must learn how to delegate accordingly. A productive way of dealing with so many responsibilities is looking at them as roles that anyone can conform to. For example, you might conform to a meeting mediator or a process automation role now, but, as the team grows, you should teach and push for other members to take ownership of those duties. By giving more responsibility to your team members, you reinforce the group values and high standards through its participants, which is a fantastic way to create a strong culture where people feel valued, trusted and accountable for actions, behaviors and outcomes.

# Epilogue

We've worked in companies with virtually infinite resources of money and time, and still we've seen codebases getting out of hand while the team burnout decreased morale and increased unhappiness. At the same time, we've worked with companies with severely limited resources of time and money, where the team were joyful and the codebase was a profitable asset in allowing the business to rapidly adapt to market trends and customers' needs. What we've observed over and over again is that the challenge of building great software is rarely a lack of resources or even technical, as most believe it is. Developing software is a social activity, so the challenges are usually social ones.

In our journey of continuous improvement as software engineers, mentors and founders of a business, we sought help from various disciplines including leadership, finance, marketing, psychology and economics. Although all disciplines helped us mold our opinions, the one discipline that stood out was ”behavioral economics.” A relatively new discipline, behavioral economics emerged from the intersection of psychology and economics during the 1970s. This newfound branch of economics examined the decision-making process of individuals and teams, how the decision-making process came to be and whether the choices were ultimately beneficial for the people making them.

This discovery came at the right time for us, as we had already started Essential Developer, with a mission to help professional software developers and teams become more valuable and achieve a rewarding and fulfilling life and career.

While studying the work of Daniel Kahneman, Amos Tversky, Richard Thaler and others, we started noticing how their findings helped us understand our own industry a lot better. For example, we've repeatedly witnessed the same predictable behaviors in dysfunctional teams producing codebases that were rigid and hard to maintain, along with the repercussions they brought to the tech teams and, ultimately, the business as a whole (high risk and, too often, low reward). By better understanding decision-making in functional teams, we can more easily distinguish the other side as well, where codebases and teams are considered valuable assets to the business, which enabled higher rewards by taking much fewer risks.

Although the words ”risk,“ ”credit,“ ”debt,“ ”assets,“ ”liabilities” and other terms derived from economics and finance are not used in our craft very often, we'd like to see that change. Software development is a multileveled endeavor consisting of many entities and actors. We can definitely acknowledge that it's not only about writing code. Every little decision we take asynchronously influences a plethora of entities, including ourselves. It is for this reason we see ourselves as risk managers. With the proper awareness and discipline, we can learn to better understand how to manage the risks our choices bear; thus, increasing the probability of achieving higher rewards.

As a final note in this book, we'd like to highlight the importance of the three pillars of *Empathy, Integrity and Economics* and how they all fit together. We need to be aware of all three when making decisions. This mindset works like a chair, and, like a chair that has only one or two legs, it won't take long until it falls. As long as we humans write code professionally then we need to treat coding as a social activity. Practicing empathy is equally as important as contributing code with integrity or making rational economic decisions for yourself, your team and your business.

We wish you a happy, productive and prosperous life!

Caio
Mike

# About the authors

**Caio Zullo**

**Twitter: @caiozullo**

**LinkedIn: /in/caiozullo**

I've been writing software since 1998, professionally since 2006, and on Apple platforms since 2009. I love building robust, well-engineered, and beautiful applications and coaching developers to achieve their best potential.

**Mike Apostolakis**

**Twitter: @mrmichael**

**LinkedIn: /in/mapostolakis**

I'm a software engineer from Athens, Greece. My goal is to help the software industry evolve by enabling developers and companies to practice valuable techniques and build powerful and durable systems.


# Connect with us

**essentialdeveloper.com**

twitter.com/EssentialDevCom
youtube.com/EssentialDeveloper
facebook.com/EssentialDeveloper
github.com/EssentialDeveloperCom