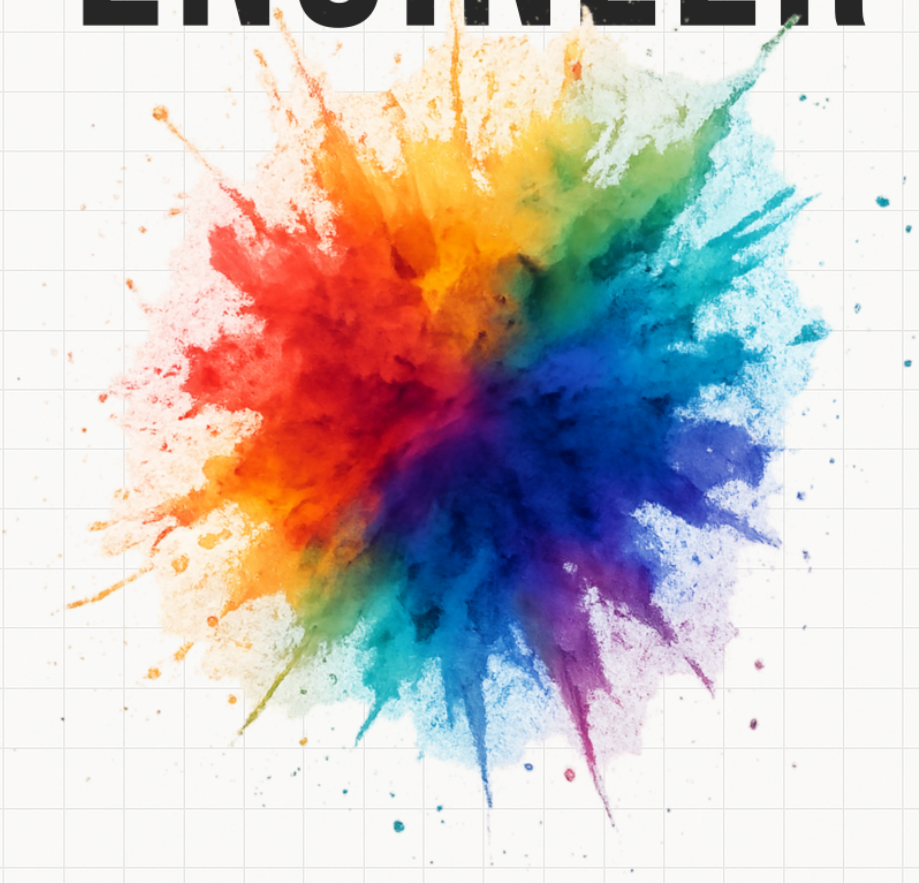
THE IMPACTFUL ENGINEER



MARCO PRACUCCI

Contents

Introduction

You care about your work. You want to build things that make a difference — not just ship features and close tickets. But in the daily rush of deadlines, meetings, and shifting priorities, it's easy to lose sight of what really matters.

This book is here to help you get that focus back. It's about thinking smarter, choosing what's worth your time, and turning ideas into results that actually move the needle.

The Impactful Engineer is built from real stories, lessons learned, and simple habits that make great engineers stand out — not because they work harder, but because they work with purpose.

You won't find buzzwords or big promises here. Just clear thinking, honest stories, and practical tools you can start using right away.

If you've ever felt there's more you could do — more impact you could have — this book may help you get there.

WHY THIS BOOK?

Over the years, I've collected thoughts and stories on software engineering, time management, prioritization, and the daily struggles of being an engineer. At first, it was just a private document — a place I used to reflect, run retrospectives, and sharpen my thinking. It helped me become a better engineer and, more importantly, a more impactful one.

During annual performance reviews, I often revisited that document. Sometimes I'd share snippets with peers, tailoring them to the feedback I wanted to give. But my reach in those moments was limited. Over the course of my career, I may mentor a few dozen engineers.

So I asked myself, why not share these notes more broadly? Why not take the lessons, the stories, the ideas, and put them in front of a - hopefully - bigger audience? Why not amplify the impact?

Well, believe it or not, that's how this book started.

WHAT IS THIS BOOK?

Let's start with what it's not. This book isn't about hacking your way to a four-hour work week. It's not about playing politics, gaming the system, or faking impact at someone else's expense.

This book is about using your time and energy wisely. It's about working hard — and working smart. It's about discipline, structure, and methods that raise the ceiling on what you can achieve.

But there's a catch: no book, no course, no mentor can hand you the exact steps. Ultimately, only you can discover your path to greater impact.

What this book can do is expand your horizons. It can challenge you, sometimes with a contrarian view, sometimes with a simple reminder of what matters. You don't need to agree with everything here. That's not the point. The point is to spark thought, to help you see things differently, and to push you to carve your own path as an exceptional - and impactful - engineer.

Take what's useful, challenge the rest.

HOW IS THIS BOOK WRITTEN?

Most books are written as a business. And in publishing, books are often paid by weight — the more pages, the higher the price, the "better" the business.

From the reader's perspective, that's nonsense. Most technical or self-improvement books could be written in half the pages — or less. Personally, I'd gladly pay double for half the content if it meant saving my time.

I value time — mine and yours. That's why this book is short, direct, and to the point. I avoided stuffing in ten examples when one does the job. I avoided repeating the same idea three times just to pad the page count. You're smart enough to connect the dots and map the concepts to your own experience.

The chapters follow a simple structure: first the *why*, then the *what*, and finally the *how*. Because unless you understand why something matters and what the goal is, the how doesn't stick.

So, let's get to it — and happy reading!

Start with the why

> *Speed is irrelevant if you are going in the wrong direction.*
>
> — MAHATMA GANDHI

Most engineers start with *how*. How do we build this? How do we scale it? How do we optimize it? That's comfortable. It feels technical. It feels like progress.

The real leverage, the real impact, comes from asking why. Why are we building this at all? Why does this problem matter? Why is this the best way forward?

Don't get me wrong — the *how* matters. Without it, nothing ships. But starting with *how* before *why* is like picking tools before knowing what you're building. And yet, many engineers do exactly that every day.

The *why* shapes the *what*, and the *how* comes after. *What* you do is way more important than *how* you do it. You can execute flawlessly, optimize every step, and work at lightning speed — but if you're focused on the wrong tasks, your effort doesn't move the needle. The choice of what to tackle first, what to prioritize, and what to ignore defines the real impact of your work.

Doing something unimportant well does not make it important. You can spend hours, days, even weeks perfecting a task — but no matter how much effort you pour in, it won't move the needle if the task itself doesn't matter.

This doesn't mean craftsmanship doesn't matter — it does. Excellence compounds. But excellence applied to the wrong thing compounds waste. True impact comes from identifying the work that actually drives results and focusing your time and energy there. Excellence is valuable only when applied to the right things.

History is full of people who made the greatest impact by asking *why* when everyone else was busy with *how*. Grace Hopper asked why we should keep programming in machine code when we could use words. Elon Musk

asked why rockets had to be so damn expensive and couldn't be reused. Steve Jobs asked why computers couldn't be beautiful.

Each of them flipped the question. They didn't start by asking *how* to build something new — they asked *why* the old way was dumb. Of course, none of them stopped at *why*. They followed it with relentless execution. That's the point — the *why* gave their execution direction.

To know what to do, you first need to understand why you're doing it. Clarity on the *what* comes only after clarity on the *why*. That's why the starting point is always the same: why?

KEEP ASKING WHY

In the 1950s, on a noisy factory floor in Japan, a machine came to a halt. It was a critical piece of Toyota's production line — the kind of stoppage that could ripple through the entire plant. Workers rushed over. Managers frowned. The pressure was on: every minute lost meant money burned.

Most factories would have reacted like any sane human: patch it, restart, and pray it doesn't break again.

But at Toyota, something different happened. An engineer didn't just ask, "How do we fix this machine?". He asked why it had failed in the first place.

1. "Why did the machine stop?" Because the circuit overloaded, and the fuse blew.
2. "Why did the fuse blow?" Because the bearing was not sufficiently lubricated.
3. "Why was the bearing not lubricated?" Because the lubrication pump wasn't circulating oil.
4. "Why was the pump not circulating oil?" Because metal shavings clogged the pump.
5. "Why were metal shavings in the pump?" Because the screen that filtered them out had worn away.

Five questions. That's all it took. Not five fixes. Not five frantic guesses. Just five whys.

By the time they reached the fifth why, the real problem wasn't the fuse at all. It was a worn-out filter, neglected because maintenance routines weren't properly followed. If they had stopped at the first answer — a

blown fuse — they would have swapped it out, restarted the line, and waited for the next fuse to blow. And the cycle would repeat. But because someone kept asking why, they got to the root cause. They fixed the process, not just the symptom.

This became known inside Toyota as “The Five Whys”. It wasn’t just a technique. It was a mindset — a refusal to accept surface answers. A belief that real progress comes not from patching problems, but from digging until you understand the truth beneath them.

The results speak for themselves. Toyota transformed from a small, post-war carmaker into the most efficient auto manufacturer in the world. Their obsession with why turned into the Toyota Production System, which later evolved into Lean Manufacturing — a framework that reshaped industries.

The difference with other car makers wasn’t technical brilliance. Other car companies had smart engineers, too. The difference was that Toyota engineers were trained to flip the question. While most of the world defaulted to "how do we fix it fast?", Toyota asked "why did this happen at all?".

That single shift unlocked an entirely new way of working.

Better *how* makes good teams great — but it never turns the wrong problem into the right one.

Impactful engineers don’t stop at *how*. They push deeper. They ask why. And then they ask it again. And again. Until the problem is clear. Until the customer’s real need comes into focus.

UNDERSTAND THE BUSINESS

To answer the *why*, you have to understand the business. Not superficially. Not by reading a mission statement. You have to understand the customers, the market, and the field you’re playing in.

Not every engineer dreams of reading balance sheets. Fair. But if you want your work to matter beyond code elegance, you can’t ignore the business behind it.

Consider the difference between a startup and an established enterprise. A startup has almost nothing but potential. No customers. No processes. No history to defend. In that context, asking “why” often leads to radically different answers than in a mature company.

The mantra “move fast and break things” exists for a reason: when you have nothing to lose, the biggest risk is in standing still. Fail fast, learn fast, iterate. That’s the startup way.

Now take an enterprise. They have products in the market. Thousands of customers. Brand reputation. Complex systems. Processes. Bureaucracy.

That “move fast and break things” mentality suddenly becomes reckless. You can’t iterate in the same way, because every break can erode trust or revenue. An enterprise must optimize differently. Their why is tied not just to growth, but to preservation. Their constraints are not just technical — they are business constraints.

Neither approach is smarter — they’re just optimized for a different game. The trick is knowing which game you’re playing.

The same logic applies inside a single company. A brand-new product that barely generates revenue will operate very differently from an established one that pays hundreds - or thousands - of salaries. The balance between risk and opportunity is fundamentally different.

Deploying a new feature on Friday afternoon might be fine — even desirable — for that new product you’re rapidly iterating on to win customers. But doing the same for a product with millions of users, a few hours before leaving it unattended for the weekend? That’s not bold. That’s reckless.

Understanding the business isn’t just about knowing the KPIs or quarterly targets. It’s about knowing the forces that shape the decisions your company makes, consciously or unconsciously.

Why do we prioritize this feature? Because it retains customers in a saturated market. Why do we invest in this process? Because a single outage could cost millions. Why do we favor stability over quick experimentation? Because the cost of breaking things is higher than the potential upside.

Your engineering decisions only have leverage when they are aligned with the business. Building faster code doesn’t matter if it doesn’t help the company survive or grow. Optimizing a service for scale doesn’t matter if the business is still figuring out what the product is. Shipping faster is meaningless if you risk losing millions in churned customers. You cannot separate the technical from the business, because they are intertwined in the real world.

Of course, sometimes the business’s *why* is wrong or outdated – markets

shift, strategies drift. That's when understanding the business helps you challenge it intelligently, not just follow orders faster.

Understanding the business doesn't make engineering easier — it makes it smarter. It gives your *why* purpose and your *how* direction. Without it, you're just a coder executing tickets, a solver of arbitrary puzzles. With it, you're an engineer who shapes outcomes, who drives impact. You're no longer asking how to build — you're asking how to make a difference.

Because you can be the most skilled engineer on earth, you can build the fastest, most elegant systems in the world. You can ship perfect code, deploy with zero downtime, and optimize endlessly. But if your engineering isn't aligned with the business, it's invisible. It's wasted energy. And nothing kills an engineer's potential faster than brilliant work that nobody needs.

UNDERSTAND THE PRODUCT

Have you ever seen a new engineer join the team and immediately start declaring everything broken? "The architecture is a mess". "The API is inconsistent". "This module looks like spaghetti — I could rewrite it in a week".

It's almost a ritual at this point. Fresh eyes, sharp mind, full of confidence — ready to fix the world in five business days. And honestly? I love that energy. That curiosity, that refusal to accept "because it's always been that way" is fuel for progress.

However, the reality underneath that "mess" is far more complex. That code wasn't written by idiots. It was written by dozens of smart engineers — some probably better than you — over years of iteration, customer requests, bug fixes, late nights, and product pivots.

That weirdly shaped method signature? It exists because of a nasty production bug three years ago, fixed on a Saturday night at 3 a.m. That "inconsistent" API? It's a deliberate compromise to avoid breaking thousands of integrations.

The product you see today isn't a clean implementation of a perfect design. It's a living organism that evolved under pressure — through survival, not symmetry. Every part of it reflects trade-offs between speed and stability, deadlines and debt, customer happiness and engineering elegance. Or, as Grafana's old-timer Carl once told me, "Survivors are winners".

Understanding the business is a good start. But to truly make an impact, you need to understand the product as well — its architecture, its history, and its trade-offs.

Before you challenge a decision, make sure you understand why that decision was made. And once you do, then challenge it. That's how evolution happens. Because sometimes, the "mess" is right. Sometimes, the ugly solution is the one that keeps the business alive — not the one that wins architecture awards.

Understanding the product doesn't mean accepting it as is. It means seeing the full picture before changing it — so that your improvements are real improvements, not well-intentioned regressions.

So, like asking *why* in business, asking *why* in the product is just as important.

SEE, LISTEN, AND ASK

At this point, you may be thinking: "Cool — but what does that mean in practice? How do I actually learn about the business and the product?" Here are a few strategies that I've found incredibly effective over the years.

Talk to the people closest to the problem

Not just your manager. Talk to support engineers, solutions engineers, sales, customer success, and operations. They see what customers actually complain about, what breaks under pressure, and what truly costs money.

You'll learn more about the real business from one honest conversation with a support engineer than from ten sprint reviews.

Use the damn product

You'd be shocked at how many engineers don't use what they build. Sign up. Break it. Go through onboarding. Try to experience the UX like a real customer.

You'll spot friction points and missing context that no ticket description will ever tell you.

Follow the money

Find out what actually drives revenue, retention, or cost. Which features bring customers in? Which ones keep them? Which ones are just there because “we’ve always had them”?

Once you understand how the company makes — or loses — money, your technical priorities will shift overnight.

Dig into the history

Look through old pull requests, RFCs, design docs, and postmortems. They’re the archaeological record of the product. They tell you not just what decisions were made, but why — and sometimes why they were wrong.

Seek out veteran engineers and capture their historical knowledge. You’ll begin to see patterns — the principles that guided the design and development, the trade-offs picked, and how constraints shaped choices.

Ask annoying questions — but good ones

Don’t be the person who complains loudly; be the one who investigates deeply. “Why did we choose this approach?” “What trade-off were we optimizing for?” “Is that still true today?”

Ask like an engineer, not like a rebel. The goal isn’t to prove you’re smarter — it’s to uncover the context that makes you smarter.

Shadow the customer, not just the code

Watch how people actually use your product. You’ll see them misuse it, ignore features you thought were genius, and depend heavily on the ones you almost deprecated. That experience rewires your sense of what “impact” really means.

Learn the market and your competitors

Don’t just focus inward. The product you’re building exists in a competitive landscape, and understanding that context is crucial. Talk to sales teams about lost deals — why did prospects choose someone else? What features or experiences did competitors offer that we don’t?

Look at customer feedback on alternative products, read reviews, and track emerging trends in your space. The goal isn’t to copy competitors —

it's to understand the gaps, unmet needs, and opportunities that your team can uniquely solve. When you combine this market knowledge with deep insight into your own product and business, your engineering decisions gain strategic leverage.

The breadth and depth of your whys will change over time. Early in your career, the "hard part" feels technical: learning a language, understanding network protocols, wrestling with design patterns, shaving milliseconds off a function. Your whys tend to be scoped to the task in front of you — more about technology than business. That's fine. You're building the toolset.

Later, technology stops being the bottleneck. Most things look doable — not trivial, but doable — and very few technical problems scare you anymore. The hard part shifts outward: understanding real customer needs, aligning a team, influencing stakeholders, optimizing not just code paths but organizational ones. Your whys get broader — and more impactful.

Don't stress if you don't see the whole board yet. Just keep pulling the thread: keep asking why, and keep pushing to understand how the business actually works.

But don't drown in it. Asking why is a compass, not a hammock. Asking why isn't an excuse to overthink; it's the start of smart action. Once you see the direction, move.

Choose what matters

> *Nothing is a priority if everything is a priority.*

You've asked *why*. You understand the mission, the product, and the business. You see where the team is heading. And now you're ready to move.

But where do you start?

Ideas are everywhere. The backlog never shrinks. New requests flow in constantly. Emergencies, shifting priorities, unplanned work — they keep piling up. Every day, a new Jira ticket pops up promising to change the world — or at least to fix that dropdown.

Every moment of the workday feels like an opportunity to slice time thinner. And engineers? We're excellent at saying yes. Curiosity, a love of challenges, and a sense of responsibility push us to take on almost every request. More work. More busyness.

If you try to chase all of them — if you say yes to everything — you'll end up very busy but not very effective.

Effectiveness isn't measured by how busy you are. It's measured by the results you create. Busyness is quantity: effort without focus, speed without direction. Effectiveness is quality: effort with leverage, speed with direction — velocity.

Busyness is comforting. It feels like progress. It signals that we're working, that we're doing something. But effectiveness is rare. And that rarity is precisely what makes it valuable — the key to having a greater impact.

Now, sure — sometimes you don't control the backlog. You don't choose the roadmap. But even then, you do control how you spend your attention: where to go deep, what to simplify, what to challenge, what to ignore. Even in the worst-case scenario — when your manager is breathing down your neck and micro-managing every move — you still have choices. You still control how you organize your day, what to focus on, and what to

skip. At the end of the day, you're the one with your hands on the keyboard. Impactful engineers play the long game — they pick their battles.

The uncomfortable truth is that most ideas don't matter. Some are noise, some are distractions, and a few — just a few — are worth your full attention.

As engineers, we're drawn to shiny things. We love complex puzzles, technical challenges, a new technology to try. Our bias is to think that the more complex a project is, the more impactful it will be. Except that… it may not.

The most impactful work isn't necessarily the hardest technically — it's the one that moves the needle for the business, the product, or the users. Resisting our biases, figuring out what's truly worth doing, and channeling our time and energy into that work — that's what really matters.

THINK BEYOND THE CODE

The days when coding alone made you valuable are over.

Code is now abundant, and writing it has never been easier. Decades of abstractions, open-source libraries and frameworks, endless technical documentation, pre-cooked solutions, and — last but not least — the rise of generative AI have lowered the barrier to entry dramatically.

It might feel like everything's getting more complex — and at a micro level, that's true. But zoom out, and you'll see the opposite: there's never been a time in history when building software was easier. There's never been a time when creating a working, useful, end-to-end product was faster than today.

Today, nearly anyone can build an application. You might argue that AI-generated software isn't as good as yours — and sometimes you'd be right — but it's improving fast. By the time you read this, it'll already be better.

In a world where code is abundant and nearly everyone can write it, the value of pure coding is in decline. It's not new; it's just accelerating. Our job hasn't been just "writing code" for decades — but this, right here, marks the final death of coding as a unique source of leverage.

The value isn't in the code itself. It's in the problems you solve and the customer needs you address. Code is a tool — not the goal. Sometimes the smartest move isn't writing code, but deleting it, preventing it from being

written, or changing a single overlooked configuration option.

So, the question isn't whether you can build something — of course you can. AI can too, at least to some extent. The real question is whether building it makes sense. Just because you can write code doesn't mean you should. Paradoxically, generative AI made this even more true: the easier it is to build, the more valuable it becomes to choose what to build.

Is this feature worth building, and why? Does it make the product better or worse? Does the user experience improve, or does it add another layer of confusion? Is that tiny but nasty bug worth fixing? Should this business process even exist? The real value lies in your ability to discern what's worth doing — and what's not.

In today's world, if you want to make a real impact, you need to shift your mindset. Don't just think like a coder — think like a builder. Sometimes that means wearing the hat of a product manager, a project manager, or the customer. Because the engineers who shape the future aren't the ones who code the fastest — they're the ones who choose best what to build.

THE TWO LAWS GOVERNING EVERY PROJECT

There are two laws governing nearly every project. Even if the authors never met in person, these laws become powerful when combined, and more than a hundred years later, they're still as relevant as ever. I'm talking about the Pareto Principle and Parkinson's Law.

The Pareto Principle, also called the 80/20 rule, says that roughly 80% of the results come from 20% of the effort. In software, this could mean that 20% of your features deliver 80% of the user value — or that 20% of your bugs cause 80% of the customer complaints. The key insight here is that not all work is created equal. Most of what we do contributes very little to meaningful outcomes.

Parkinson's Law, on the other hand, is deceptively simple: work expands to fill the time available. If a task has a week to be completed, it will somehow take a week, even if it could be done in half the time. Of course, there are cases where you can't further shrink the required time, but the key idea is that giving a project more time than it truly needs doesn't linearly increase its value. On the contrary, the extra time is often filled with doing the 20% that nobody really cares about.

Here's where it gets interesting. You can use these two laws as a high-level guideline to gain leverage on any project:

- Identify the 20% of work that produces most of the value — and focus on it.
- Force yourself to do it in less time than you think you need.

The harder question, of course, is: how do you identify where the value actually is?

CHOOSE LIKE AN OWNER

Thinking like an owner starts with perspective. You're not just completing tasks — you're responsible for outcomes. You anticipate problems before they happen, make tough calls without being told, and act as if the success of the product depends entirely on your choices. Because, in a way, it does.

How can you apply the owner mindset to identify what to focus on?

Think in outcomes, not tasks

Many engineers focus on checklists: tickets closed, PRs merged, lines of code written. Owners think differently. Instead of treating tasks as isolated units of work, ask yourself:

- What result am I trying to create?
- What need am I addressing?
- What experience am I delivering to the customer?
- Will this attract more customers and drive higher revenue?
- Will this save costs and improve our margins?
- Will this make customers happier and reduce churn?

Put yourself in the customer's shoes

Use the product you're building as much as possible. If that's not feasible, gather feedback from customers or proxies — sales, solutions engineers, or customer support. Then ask yourself:

- If I were the customer, which features would I actually use?
- What do customers complain about the most?
- What do customers find most confusing or difficult to use?
- Which deals are we losing to competitors, and why?

Act as if it's your money

Use a simple mental shortcut: think about time, money, and return. Ask:

- If I owned this company, would I still do this?
- Is this task worth my time — and the money the company is spending on it?
- What can I simplify, remove, or delay to get it done faster without hurting the product?

Then reverse-engineer the work that needs to be done to achieve it. It's not about finishing more stuff — it's about finishing what matters.

SIMPLIFY YOUR DECISION PROCESS

How do you make those choices consistently? How do you decide, day after day, what deserves your attention and what doesn't?

I love simplicity. And a simple mental framework I use every single day is the Eisenhower Matrix.

The Eisenhower Matrix is deceptively simple but incredibly powerful. It reminds you to focus on payoff, not just urgency. The core idea is to ask yourself:

- Is this task urgent?
- Is it important?

And then place each task, each project, each request, even the smallest one, in one of these four categories:

1. **Important and urgent**
   These are the fire drills. Stuff that must get done now because it has real consequences if ignored. For example, a critical bug affecting customers, an outage, or a feature blocking a major launch. Stop everything else and do these first.
2. **Important but not urgent**
   The sweet spot. Work that moves the needle but doesn't scream for attention. For example, designing a new architecture, improving user experience, or building an internal tool to automate recurring activities. Schedule time to do these deliberately. This is where you spend most of your energy and generate most of your long-term impact.

3. **Not important but urgent**
   The interruptions and distractions. Many meetings, last-minute requests, or some emails. They demand immediate attention but contribute little to meaningful outcomes. Don't let them hijack your day: timebox them, process asynchronously where possible, or delegate.
4. **Not important and not urgent**
   The noise. Probably 80% of your backlog. Low-value tasks, time-wasting requests, or features nobody really needs. Ignore them.

   Sometimes you'll look at your list and realize there's nothing in the "not important and not urgent" box. It typically means one of two things.

   The first - and worst - case is that everything is labeled as a priority. That's a symptom of noise disguised as importance. When everything's on fire, nothing really is. Your job is to spot the fake alarms — the work that feels urgent only because someone shouted loud enough. Ask yourself: If this doesn't get done today, what actually happens? Who's affected? What's the real cost of waiting? Those questions quickly separate signal from noise.

   The second - and best - case is that your leadership already filters out the noise before it reaches you. In that environment, priorities stay clear by design, and you can spend your energy on meaningful work instead of firefighting fake emergencies. Treasure that — it's rare.

The beauty of the Eisenhower Matrix is how it forces a clear separation between what matters and what doesn't, instead of letting the loudest or most immediate things steal your attention. It makes you conscious of where your time and energy go, which is half the battle in engineering work.

No, you don't have to grab a whiteboard and draw the quadrants every time. Think of it as a mental model. Once you get used to it, you'll unconsciously apply it to every request you receive — tens of times a day. Build the habit of daily triage, constantly asking: Is this urgent? Is this important? And if it is, is it really the highest-leverage thing I could be doing right now?

Applied consistently, it becomes a kind of decision autopilot:

- You stop reacting to every incoming ticket or email.
- You start spending your energy on the work that actually matters.
- You recognize that saying "no" or "later" is not shirking responsibility —

it’s choosing impact.

Make it real

> *Do it without waiting for someone to tell you to do it. Welcome to the 1%.*
>
> — @JACKBUTCHER

Asking why gives you direction. Choosing what matters keeps you focused. But don't stop there — clarity without action is just wasted potential.

The value of your work isn't in brainstorming meetings or in a perfectly written design doc. Those help, sure, but they're just the baseline. The real value lies in what you ship to customers. That's what moves the needle. That's what creates impact.

And to ship, you need to act — to turn ideas into products, features, or whatever solves a customer's problem. To get your hands dirty, learn what breaks, and make it better. Engineering, at its core, is a contact sport.

The most surprising thing? Failing at execution is easier than you think. Projects don't fail because people are dumb — they fail because teams move too slowly, lose focus, overthink decisions, or move in the wrong direction.

At the same time, executing well isn't hard. It doesn't require supernatural powers. It's mostly a handful of habits — focus, short feedback loops, fast validation, a bit of common sense, clear ownership — that anyone can learn.

But before any of that, there's a first, simple step: move. The most impactful engineers I've ever worked with all shared one thing in common — a bias to action.

So, don't wait for someone to tell you to do it. Once you know the *why* and the *what*, hesitation has no excuse left.

BUILD, SHIP, MEASURE, AND REPEAT

No matter how many hours you spend in meetings, how many design docs

you write, or how many “strategic discussions” you have, one truth remains: reality always has the last word.

You can brainstorm for weeks. You can run every scenario in your head, whiteboard the perfect architecture, and write a design doc so detailed it could win a Pulitzer. But the moment your idea meets the real world, it will start behaving differently. Customers will use it in unexpected ways. Edge cases will crawl out of the shadows. Your “brilliant shortcut” will backfire. That clever abstraction will crumble under load.

Welcome to software engineering — where theory and reality have a complicated relationship.

You’ll never have perfect information. You’ll never fully predict user behavior, market response, or system failure. Every plan — no matter how good — is just a collection of educated guesses. Unless you have the crystal ball - and you don’t - you continuously navigate through uncertainty.

Don’t get me wrong. It’s not an invitation to try out random things without even thinking, just because you can’t predict the future. An idea that doesn’t make sense on paper will rarely turn into a brilliant solution in practice. Clear thinking before taking action can help you get to a solution. Experience and gut feeling can definitely be your allies. But at the end of the day, there will always be some unknown unknowns. No matter how hard you try, you will never completely eliminate them.

That’s why the most impactful engineers don’t obsess over being right — they obsess over learning fast. They know that every assumption is just a hypothesis until validated. So they build, ship, measure, and repeat. Fast.

DON’T GUESS, MEASURE

Don’t guess, but measure. Validate your assumptions in the field — early and frequently. In essence, shorten the feedback loop until reality can’t hide from you.

Each loop tightens your understanding. You build something small, ship it, measure what happens, and adjust. Then you do it again. Like tightening a spiral around the truth. It’s an iteration, but with intent:

- Do customers actually use it?
- Does it solve the problem we thought it did?
- Is our performance assumption still true at scale?

Amazon calls it working backwards. Tesla calls it rapid iteration. Startup founders call it not dying. The principle is the same: feedback beats foresight.

I know what you're thinking: "Sure, but my project will take at least a quarter before a minimum viable version exists. There's no reasonable way to validate assumptions sooner".

Maybe. But probably not. In my experience, there's always a way — if you're willing to get creative. You don't need to test the entire solution. You just need to test a slice of the risk. Instead of validating the whole product, validate the assumptions that could kill it.

Not convinced? Try these tactics:

- **Fake it before you make it**
  Before Dropbox wrote a single line of sync code, Drew Houston made a 3-minute demo video showing what the product would do — and watched signups explode. He didn't need a product to test demand. He just needed evidence.
- **Prototype the riskiest part**
  Building a distributed cache? Don't design the full architecture. Mock the API and benchmark a simplified version. You'll learn what breaks, what scales, and whether your idea even makes sense under load — in days, not months.
- **Dogfood ruthlessly**
  Before public release, use your own product internally. Real usage exposes hidden friction and silly assumptions faster than any design review ever could.
- **Shadow-launch**
  Roll out your feature to a small percentage of traffic, collect telemetry, and compare performance. Alternatively, mirror live traffic to the new version while users continue interacting with the old one. This lets you compare both solutions side by side — and see how the new system behaves under real-world load, without risking production stability.

Some of these tactics may feel risky in highly regulated or large-scale systems. In those cases, test assumptions in safe, controlled environments — the principle of early feedback still applies.

When you ship small, you make failure cheap and learning fast. You can afford to be wrong — repeatedly. Each iteration makes the next one smarter. It's like compounding interest, but for learning. You need early signals, not late surprises. Every day you wait to get real feedback is a day

you delay learning.

Learning quickly can completely change your course. In some extreme cases, the most successful products ever built began as accidental discoveries:

- Twitter started in a podcasting company - you probably never heard of - called Odeo.
- Slack was a failed game that turned into the team's internal chat tool.
- YouTube started as a dating site — "Tune In, Hook Up". Yes, really.

The goal isn't to reach the final solution faster. The goal is to discover the truth sooner — so you don't spend three months building the wrong thing beautifully. Because nothing is slower than perfecting the wrong thing.

DON'T BE AFRAID TO EXPERIMENT

Woodworkers have a saying: "measure twice, cut once". In woodworking, when you cut a piece of wood, it's gone. Forever. If you smooth, engrave, or plane a surface and make a mistake, there's no undo button. There's no way to revert the material back to its original state. You have to throw the piece away and start again from scratch. In the worst cases, a single wrong cut can mean rebuilding the entire product. Many decisions in woodworking are irreversible, which is why careful preparation and precision are not optional; they are survival. I imagine surgeons have a similar saying, too, though I've never been a surgeon.

Software engineering, on the other hand, is a different world. If you cut a piece of code, you can bring it back by reverting a git commit. If you add, modify, or remove a feature and later regret it, you can usually roll it back with relative ease. The whole product doesn't need to be rebuilt from scratch. Of course, there are situations where changes become much harder to undo: corrupted or lost data, financial transactions gone wrong, or a bug that damages customer trust and brand reputation.

But those are exceptions. For the most part, decisions in software engineering are reversible. And this gives you a unique advantage that many other industries simply don't have: the freedom to experiment and move quickly. You can keep your organization lean, make decisions fast, test ideas with minimal risk, and iterate again and again until you get it right — all while keeping the downside limited.

Jeff Bezos once put it this way in an interview: "Most decisions are two-way doors. If you make the wrong decision, if it's a two-way door, you pick

a door, you walk out, you spend a little time there. If it turns out to be the wrong decision, you can come back in and pick another door. Some decisions are so consequential - and so important - and so hard to reverse that they really are one-way door decisions. You go in that door, you're not coming back - and those decisions have to be made very deliberately, very carefully."

The two-way door is a powerful decision-making framework. Think of reversible moves as low-cost experiments. Treat one-way moves like surgical procedures. Ask yourself a few key questions to determine whether it's easily reversible, for example:

- Will undoing this require more effort than doing it in the first place?
- Does it involve customer data in a way that's hard or impossible to restore?
- Is there compliance, contractual, or regulatory exposure if it goes wrong?
- Could it cause visible customer harm — lost money, a public outage, a privacy breach, or negative press?
- Will this change break backward compatibility for existing users or systems?

If the answer to all of these is no, you're looking at a two-way door. If even one answer is yes, then it's a one-way door — or at least risky enough to treat it like one.

Move fast on two-way doors. Keep the decision process lean and biased toward action. Build, deploy, measure, learn, and iterate. Speed is the advantage here — use it.

Move slowly on one-way doors. Take the time to gather data, consult stakeholders, and analyze risks. Discuss thoroughly with the team, and give people enough time to digest and raise concerns. Don't leave risks unaddressed before proceeding. Whenever possible, make the irreversible reversible: break the big change into smaller bets, add safeguards, and roll out gradually to reduce risk.

Don't confuse speed with recklessness. Measure twice when you're standing in front of a one-way door, and be surgical about the execution. For two-way doors, make small bets, set explicit rollback criteria, and iterate — fast. Build habits that make reversibility the default, practice to quickly spot which door you're at, and get velocity with controlled risk.

MASTER NAPKIN MATH

When designing or experimenting, napkin math is one of your best allies.

Mastering it gives you an edge — whether you're comparing alternative solutions, estimating the impact of an optimization, or evaluating how much that shiny new service will cost to run in production.

You don't need perfect numbers. You just need to be directionally correct — the right order of magnitude. If your napkin math tells you a new feature will cost $10,000 a month instead of $100, that's enough to change your decision.

The goal isn't perfection — it's clarity. Napkin math helps you see what's worth doing before you spend a week benchmarking or a quarter building.

Start with boundaries

Even if they're unrealistic, define your best-case and worst-case scenarios. Those are your fences. You can't do better than the best case. You can't do worse than the worst.

For example, say you're debugging a slow API that's breaching your latency SLO. You've identified a slow function as the culprit. Your boundaries are:

- Worst case: do nothing — keep the function as is.
- Best case: remove it entirely — no computation, no latency.

Now, if removing the function entirely would only improve latency by, say, 20%, then even in the best case, it may not be worth the effort. Maybe caching the whole response or parallelizing the workload will get you a bigger win — and this simple napkin math can point you in that direction within minutes.

Boundaries keep you honest. They tell you whether an idea is worth more thinking or is already a dead end.

Know your invariants

Every system has constraints that won't budge — at least not anytime soon. Know them.

- Network round-trip time between regions

- Disk I/O latency and throughput
- Per-node CPU or memory cost
- Database query fan-out limits
- Eight bits in a byte (yes, some engineers still get confused)

These are your non-negotiables.

For example, say you're estimating how long it'll take to move 10 TB of data between two data centers. You might start with the 100 Gbps link between them and estimate around 15 minutes. But if your source data lives on network-attached storage capped at 10 Gbps, that's your real bottleneck — in practice, it'll take over two hours. You'll be off by 10× if you miss that invariant.

Make educated guesses

Napkin math relies on intuition, but not imagination. If you have to guess, make educated guesses. Base them on real data: telemetry, customer usage, cost reports, or previous projects.

Say you're estimating how much a new logging system will cost. You don't know the exact volume yet, but you can look at current logs, estimate the average log line size, multiply by logs per day, and check what your provider charges per GB stored. Even if you're off by 20%, that's fine. You'll still know whether it's roughly $100 a month or $10,000 — and that's all you need to decide if it's worth continuing.

Be realistic, not idealistic

Napkin math is not wishful thinking. It's reality testing. Don't let optimism bias creep in just because you want the answer to look good. If you're missing data, use past experience or comparable systems to stay grounded.

For example, say you're designing a new in-memory cache. Your idealistic side might assume "cache hits will cover 90% of requests". But based on similar workloads, you know 60% is more realistic. That 30% difference could double your application load — and blow your cost model.

Napkin math isn't about confirming your hopes — it's about protecting you from them. It's reasoning under uncertainty — and the faster you can do it, the faster you can make smart calls without waiting for perfect data. It's the difference between saying "let's run a benchmark next week" and figuring out why it won't work in a matter of minutes.

DON'T LET THE PERFECT STAY IN THE WAY OF GOOD

Let's be honest: we're all perfectionists.

The architecture? It's never clean for us. The code? There's always one more refactoring to do. The tests? Never comprehensive enough. The UI? It could always be sleeker, smoother, more polished. The feature set? Well, there's always one more "must-have" thing you could add.

Perfectionism is noble — it signals care, skill, and high standards. But if you always wait for the perfect solution — that, spoiler, will never arrive — you'll never ship. Your product, feature, bug fix, or optimization will stay on your laptop or in your staging environment, while time passes, deadlines shrink, and stress rises. Nothing reaches your users, nothing delivers impact.

A couple of years ago, a software engineer from my neighborhood reached out after hearing that I'd co-founded a startup early in my career. He wanted feedback on a mobile app he was building — a local events aggregator for tourists (we live in a very touristic area). Within minutes, I pointed out a couple of obvious challenges: keeping event listings fresh and getting actual users. But he kept circling back to technical details — why his design was better, what framework he used, how his app "stood out". I told him to just ship it that day, get real feedback from the market, and iterate. He nodded politely — then ignored me.

Months passed. Every now and then, I'd bump into him around town. The app was always "almost ready". There was always one last feature, one last bug, one last improvement. Two years later, on a sunny afternoon, I saw him again and asked if he'd finally launched. He had. But it didn't go as he expected. After two years of refining every pixel, the app was flawless — in his own mind. Unfortunately, the market didn't care and none was using it. Chasing perfection hadn't brought success. It only delayed failure.

Many successful products aren't technically flawless. They're not always the fastest, most elegant, or most complete. But they shipped sooner than the alternatives, hit the market when users were ready, and solved a real problem with a good enough solution. Perfect? No. Successful? Absolutely.

This isn't an invitation to build sloppy, half-baked software. It's a call to understand the trade-off between perfect and done. It's a call to ship the good enough, even if there are ten more ways to make it better. It's a call to embrace time to market as a strategic advantage.

Shipping imperfect work — and iterating based on real feedback — is where impact lives. Done beats perfect. Every time.

STAY HUMBLE

Last, but not least: stay humble.

You have to accept that your brilliant idea might flop, that your assumptions might be wrong, and that your customers might not care. And you have to be okay with that — as long as you learn something valuable before the next build.

Sometimes, the entire premise of a project is wrong, and the best course of action is simply to cancel it. You have to be ready to let go. To stop insisting on that "brilliant" idea that doesn't work. No matter how elegant it looks on paper — if it doesn't work in the field, it doesn't work. Move on.

A few years ago, I worked with an engineer who spent over a year on a database optimization project he was deeply passionate about. The idea made perfect sense during the design phase and got the green light from the team. But when it finally reached production, the results were underwhelming. The optimization helped only a tiny subset of queries — too few to make a real impact.

Instead of recognizing that the project's premise was flawed, cutting his losses, and moving on, he doubled down. More months passed, results didn't improve, frustration grew, and eventually burnout followed. From zero to burnout, it took just over a year — a year spent defending a failing idea, not a failing implementation. A faster feedback loop — and a bit more humility — could have saved time, money, and a career setback.

The hard truth is that the longer you've been working on something, the harder it becomes to walk away.

Once you've invested quarters of work, sunk costs and pride kick in. Backing out feels like a personal failure — maybe even a hit to your reputation. And yes, it might sting in the short term. But spending even more time on something that doesn't work only multiplies the pain.

That's why validating ideas early is so powerful. If you test assumptions in days instead of quarters, even with a scrappy prototype, you de-risk both the project and yourself. It's much easier to kill an idea that's a few days old than one that's been part of your identity for a year.

People sometimes tell me they see only the successful projects I've led —

the ones that made a visible impact — and assume I just keep picking winners. What they don't see is that behind every success, there are nine other ideas that didn't take off. Some were tested for a week, others for a single day.

But that's the whole point. You don't need to be right 100% of the time. You can fail on 90% of your ideas — as long as you fail fast and redirect your energy toward the 10% that actually work.

Nobody will remember the nine days - not even nine weeks - you spent testing ideas that went nowhere. They'll remember the nine months you spent building the one that mattered. But if you keep pushing a dead idea just because you can't admit it was wrong — that's what people will remember, and that's what will stall your career.

This Is Just the Beginning

If you've made it this far — thanks. You're officially part of the early readers of *The Impactful Engineer*, a book still being built, shaped, and refined — just like software.

New chapters are coming soon. I'm writing, editing, and publishing them as fast as I can without breaking production.

If you want to get notified when new chapters drop (no spam, no marketing, no data sharing — ever), you can subscribe to the newsletter. You'll get a short email when new content is live. That's it.

And if you have thoughts, ideas, or strong opinions about what you've read — I'd love to hear them. Drop me a note at info@theimpactfulengineer.com and tell me what resonated, what didn't, or what you'd like to see next.

Thanks for reading — and for being part of this work in progress. You're helping shape it with your time and attention.

License

The Impactful Engineer · Version 2025-11-09

Written by Marco Pracucci

This is a work in progress. For updates, visit https://theimpactfulengineer.com