

Translating engineering decisions into numbers
driving leadership decisions

Nobody speaks the same language

Different teams, different priorities

Engineers keep leaving

Deadlines slip

Costs rise, output stalls

Big investment, weak returns

Every release breaks something

Product and engineering clash

On-call burnout

Untouchable legacy code

ANDREAS CRETEN

**Lannoo
Campus**

CONTENTS

INTRODUCTION: Translating Engineering Into Economics	12
Why this matters now	13
What this book is (and isn't)	16
A few things that might surprise you	17
Key terms you'll encounter	19
How to use this book	20
A note on numbers	21
Two companies we'll follow	21
The promise	22
 CHAPTER 1: How Money Flows Through Engineering	 24
The cycle that creates or destroys value	25
Where engineering money actually goes	27
The hidden economics that shape decisions	30
What this changes about how you think	33
KEY TAKEAWAYS	34
 CHAPTER 2: Measuring Throughput and Cost per Unit of Value	 36
The measurement problem in practice	37
A framework for measuring engineering productivity	43
Efficiency is not throughput	44
The frameworks that matter	45
What changes when you measure value	46
KEY TAKEAWAYS	48

CHAPTER 3: The Cost of a Bug	50
The explicit cost: fixing the bug	52
The hidden cost: interruption and context switching	52
The timing multiplier	53
Trust as a cost factor	55
The coordination cost	56
Bugs as signals, not accidents	57
Why teams underestimate bug cost	58
The complete picture	59
What this implies for prevention	61
What this implies for architecture	61
What this implies for culture	62
The pattern that emerges	62
The fundamental insight	63
KEY TAKEAWAYS	64
 CHAPTER 4: The Cost of a Bad Hire	 66
The decision everyone delays	67
Why the standard calculation is wrong	69
Employee Lifetime Value: the missing perspective	70
The first cost: negative productivity	71
The system drag	73
The features that never shipped	74
The real killer: destroying your best people	74
The math of losing a high performer	75
The recruitment reset	76
The complete picture	77
What this implies for prevention	79
What this implies for speed	81
The pattern that emerges	81
The fundamental insight	82
KEY TAKEAWAYS	83

CHAPTER 5: The Cost of Technical Debt 86

Why the debt metaphor actually works	88
The invisible interest payment	89
The multiplication effect	89
The learning curve becomes a learning cliff	90
The compounding begins	91
The cost you can't calculate	93
The complete picture	94
The strategic debt decision	95
The rewrite temptation	96
The economics of payback	96
What this implies for measurement	98
What this implies for prevention	100
The pattern that emerges	100
The fundamental insight	101
KEY TAKEAWAYS	102

CHAPTER 6: The Cost of Changing the Roadmap 104

The visible cost	107
The sunk cost multiplier	107
The context switching tax	108
The morale drag	108
The opportunity cost of thrashing	109
When changes multiply	110
The rebuild and learning tax	110
When to persist and when to pivot	111
The cost of “just one more thing”	112
The strategic thrashing pattern	113
The complete picture	114
What this implies for decision-making	114
What this implies for roadmap stability	115
The pattern that emerges	116
The fundamental insight	116
KEY TAKEAWAYS	118

CHAPTER 7: The Cost of More Engineers	120
The coordination tax	122
The infrastructure multiplier	123
The management overhead	124
The quality drift	125
What you gain from scale	126
The break-even calculation	127
When the economics break	128
The complete picture	129
The three strategies for scale	130
What this implies for decision-making	133
What this implies for architecture	135
The pattern that emerges	136
The fundamental insight	136
KEY TAKEAWAYS	138
CHAPTER 8: The Cost of Manual Work	140
The paradox of what's easy	142
The visible cost	143
The interruption multiplier	143
The error tax	144
The coordination drag	145
The silent cost that dominates	145
The bottleneck premium	146
The complete picture	146
When automation fails	149
The strategic value beyond time	150
What this implies for decision-making	151
The patterns that work	152
The pattern that emerges	153
The fundamental insight	154
KEY TAKEAWAYS	156

CHAPTER 9: The Economics of Build vs. Buy 158

The question teams ask	160
The visible cost	160
The maintenance multiplier	161
The feature treadmill	162
The invisible cost that dominates	162
The complexity tax	163
The complete picture	164
The cost of buying	165
The classification that matters	167
The decision framework	169
A real decision	170
When building wins	172
The spectrum, not the binary	172
The vendor lock-in fear	173
The timing dimension	173
The pattern that emerges	174
The fundamental insight	175
KEY TAKEAWAYS	176

CHAPTER 10: The Cost of a Rewrite 178

The promise that feels real	180
The comparison that misleads	181
The visible cost	182
The parallel burden	182
The silent cost that dominates	183
The feature dilemma	184
The migration reality	184
The quality tax	185
The human cost	186
The complete picture	186
When the economics might work	189
The alternative most teams ignore	190
What this implies for decision-making	192
The pattern that emerges	193
The fundamental insight	194
KEY TAKEAWAYS	196

CHAPTER 11: The Cost of Building the Wrong Thing 198

The pattern plays out repeatedly	200
The visible cost	201
The opportunity multiplier	202
The burden that never ends	202
The complexity cost	203
The removal cost	204
The complete picture	205
The certainty illusion	206
When borrowed metrics mislead	207
When one-size-fits-none	208
What this implies for validation	208
The power of no	209
The pattern that emerges	210
The question that changes everything	211
The fundamental insight	212
KEY TAKEAWAYS	214

CHAPTER 12: The Cost of Premature Optimization 216

The visible cost	218
The complexity burden	219
The opportunity multiplier	220
The learning delay	220
The optimization paradox	221
When optimization makes economic sense	222
The hidden cost of complex code	223
The complete picture	225
What this implies for decision-making	226
The three stages of optimization	226
The rule of thumb	227
The pattern that emerges	228
The fundamental insight	228
KEY TAKEAWAYS	230

CHAPTER 13: The Cost of Cutting Corners 232

What corners actually cost	234
The mathematics of compound interest	236
The hierarchy of regret	238
The compounding spiral	239
When shortcuts become strategy	241
When shortcuts become disasters	241
The complete picture	242
What this implies for decision-making	243
The payback discipline	244
The pattern that emerges	245
The fundamental insight	246
KEY TAKEAWAYS	248

CHAPTER 14: The Cost of Externalizing Development 250

The calculation that seems obvious	254
The accounting illusion	254
The coordination tax	255
The quality multiplier	256
The knowledge drain	257
The velocity penalty	258
The opportunity cost that dominates	259
The complete picture	260
When externalization makes economic sense	261
The spectrum of externalization	263
The hidden cost of lost learning	264
The quality of coordination matters more than the cost	265
What this implies for decision-making	266
The pattern that emerges	266
The fundamental insight	267
KEY TAKEAWAYS	268

CHAPTER 15: Putting It All Together	270
The trap of trying everything	271
Choosing where to start	272
A worked example	273
Building credibility systematically	275
What this changes	275
Pace and Surge, five years later	276
Where most implementations fail	278
The practice, not the event	279
 CHAPTER 16: Having the Conversation	 280
The credibility ladder	282
Speaking to engineers	283
Speaking to product managers	284
Speaking to executives	285
Speaking to finance	286
Talk tracks for difficult conversations	287
The pattern: Lead with values, support with economics	294
The conversation sequence	294
What changes when you speak economics	295
The failure modes	296
The long game	297
 CONCLUSION: What Changes	 298
 BONUS: Finding Your Starting Point	 306
 REFERENCES	 318

INTRODUCTION

Translating Engineering Into Economics

*Nobody speaks the same language
Different teams, different priorities
Engineers keep leaving
Deadlines slip
Costs rise, output stalls
Big investment, weak returns
Every release breaks something
Product and engineering clash
On-call burnout
Untouchable legacy code*

I've spent nearly two decades watching the same pattern repeat. An engineering team ships a feature three months late. The CFO asks why burn rate is up but velocity is down. The CTO struggles to explain why the team needs to spend a quarter paying down technical debt instead of building new features.

Nobody's speaking the same language. So nobody makes good decisions. Engineers talk about velocity, code quality, and technical debt. Executives talk about burn rate, runway, and revenue growth. Boards talk about capital efficiency, unit economics, and path to profitability. These conversations happen in parallel universes that occasionally collide when something goes wrong. The gap isn't about intelligence or effort. It's about translation. Engineers understand the technical implications of their decisions but struggle to articulate the economic impact. Executives understand the financial constraints but can't see how engineering choices ripple through the P&L. Investors see the metrics but miss the engineering reality that drives them.

This book exists to bridge that gap. Not by teaching executives to code or teaching engineers to read balance sheets, but by creating a shared language: the economics of software engineering.

Why this matters now

I wrote the bulk of this book in early autumn 2025. A few weeks later, our industry started changing dramatically. AI coding assistants that had been curiosities became ubiquitous. What took a team days now took hours. Code generation that seemed futuristic became table stakes. My proofreaders asked me: "Do you need to rewrite the book? Has AI invalidated everything?" I gave it a lot of thought.

Now, in late spring 2026, while doing my final review, I still believe every calculation in this book remains valid. Here's why: the economic framework hasn't changed. What's changed is where the costs land. Some costs have dropped dramatically. Writing the code to ship a feature? Much cheaper with AI assistance. Building a prototype? Faster than ever. Generating boilerplate? Nearly free. But other costs increased. Time to find a bug written by AI? Longer, because you're debugging code you didn't write and might not fully understand. Reviewing AI-generated pull requests? More expensive, because you're essentially doing a detailed audit rather than reviewing a teammate's thought process. Technical debt from hastily accepting AI suggestions? Compounds faster. The fundamental truth remains: engineering is about economic trade-offs. AI didn't eliminate the trade-offs. It just shifted where they appear.

If anything, this makes the book more valuable, not less. When the next technological shift comes (quantum computing, brain-computer interfaces, whatever disruption awaits), the economic frameworks in this book will still apply because they're built on how value gets created and destroyed in software organizations, not on the specific technologies used to create it. The tools change. The economics don't. The companies that thrived through the current shift weren't the ones with the best technology. They were the ones where engineering leaders could speak economics and business leaders understood engineering reality. I watched CTOs who could translate technical decisions into economic outcomes get bigger budgets, more autonomy, and board-level influence. Those who couldn't got their teams cut, their roadmaps dictated, and their technical judgment overridden.

The difference wasn't technical competence. It was economic fluency. When a CTO can explain that paying down technical debt will improve deployment frequency by 40%, reduce incident response time by 60%, and ultimately increase feature velocity by 25%, that's a language CFOs understand. When they

can show that investing €50,000 in better hiring processes prevents €1 million in bad hire costs, that's a conversation boards want to have. When they can demonstrate that the "expensive" senior engineer creates 3x the value of two junior engineers, the budget discussion shifts entirely.

But here's the uncomfortable truth: most technical leaders can't have these conversations. Not because they're not smart enough, but because nobody taught them how. Engineering education focuses on algorithms and architecture, not on how those decisions impact burn rate and runway. Management training teaches people processes and methodologies, not how to calculate the opportunity cost of a delayed feature or the true price of technical debt.

My life goal is simple: bridge the gap between software engineers and their non-technical managers.

Early in my career, I realized there's a massive disconnect between economic realities and technical complexity. I've worked with hundreds of technical leaders: as a coach, as a mentor, as an individual contributor. The ones who excelled weren't necessarily the best programmers. They were the ones who understood the economic impact of technological decisions and could translate technical challenges into economic realities for their stakeholders. I'm often amazed by how little technical leaders know about the economic realities of their companies. I've been in companies where they just raised significant funding and the CTO had literally no clue how this would impact the team. No growth plan. No idea how to support business growth from a product and engineering angle. Just heads down building features while burning through runway without a coherent strategy. A CTO's job is to contribute to the bottom line. Yet most can't connect the dots between their technical decisions and the business outcomes. On the flip side, I've watched investors and management teams demand speed without understanding the consequences. They want to go fast (which is valid)

but they don't grasp what that decision costs. Meanwhile, engineers want to build things perfectly when there's no time or budget for perfection. The best companies understand the stage they're in and how that translates into engineering practices: quality, speed, resilience, and the trade-offs between them.

This book is for both sides. I want non-technical managers to understand the consequences of their economic decisions on software engineering teams. I want software engineers to understand the economic impact of their technical decisions. And I want technical leaders to become fluent in both languages so they can lead effectively.

What this book is (and isn't)

This book does not give you solutions. It does not tell you what to do in your specific situation. It does not provide a playbook for every scenario. What this book does is give you a framework for thinking about engineering decisions economically. It helps you see the hidden costs that most teams ignore. It gives you the vocabulary to have better conversations with stakeholders who speak in financial terms.

This isn't an academic textbook. It's not a management manual full of frameworks you should follow. It's not a technical deep-dive into specific technologies. And it's definitely not about promoting management by numbers or reducing engineering to spreadsheets. This is a translation guide. It translates engineering concepts into economic terms that executives, CFOs, and boards understand. It translates financial constraints into technical implications that engineers can work with. The goal is to create a shared language so different parts of your organization can actually communicate about trade-offs, priorities, and investments.

You won't find prescriptive answers like "always write tests" or "never rewrite code". Instead, you'll find economic frameworks that help you think through questions like: "Given our stage, runway, and goals, what's the economically rational amount to invest in testing?" or "Under what conditions does a rewrite create more value than incremental improvement?" The answers depend on your context. But the framework for thinking through the economics applies universally. While many examples use subscription-based or SaaS companies (because recurring revenue models make certain economics more visible), the principles apply to any software organization: product companies, platform teams, internal technology organizations, consulting firms, or any team building software that creates value.

Each chapter follows a pattern: introduce the economic concept clearly, show you how to calculate the impact, ground it in research and real-world data, and give you practical tools you can use immediately. Sometimes I'll share examples from companies I've worked with. Sometimes I'll just cut to the concept. I'll also follow two companies, Pace and Surge, throughout the book, not to entertain you with stories but because watching the same decisions compound over time is the clearest way to see the economic reality behind them. Some chapters will feel obvious once you read them. Good. That means you already have the intuition. Now you'll have the formulas to back it up. Other chapters might challenge your assumptions about what's expensive and what's not. Even better. The hidden costs are the ones that compound silently until they're too big to ignore.

A few things that might surprise you

Throughout my career, I've encountered situations where the "right" engineering decision wasn't obvious, and where conventional wisdom was dead wrong.

I worked with a company that had 50,000 customers where you could only get the product after going through a sales call. As an engineer, I thought it was bonkers. Why add friction? But the CEO explained their churn was significantly lower because they could explain the product properly and adapt their pitch to each customer's situation. They qualified customers better upfront. The "friction" actually improved retention economics. Sometimes what seems inefficient from a technical perspective makes perfect economic sense.

I've also pushed back against VCs who wanted to optimize metrics that didn't match the product promise. One investor with a background in the media industry wanted to maximize time-in-app for a productivity tool. I disagreed. I wanted to see many touch points per day with very few clicks and minimal time in app. This meant people found what they needed quickly and got back to their actual work. High time-in-app would mean we failed at productivity. The economic reasoning was identical. The application was opposite. This is the nuance I want to explore: not rules, but frameworks for thinking about the economic implications of engineering decisions in your specific context.

Key terms you'll encounter

Throughout this book, I'll use terms from three domains: business, engineering, and economics. Here are the essential ones you'll encounter:

Business metrics

- ARR (Annual Recurring Revenue): Yearly subscription revenue from customers (primarily for subscription businesses)
 - MRR (Monthly Recurring Revenue): Monthly subscription revenue (primarily for subscription businesses)
 - Revenue: Total income from products, services, or licenses
 - CAC (Customer Acquisition Cost): Total cost to acquire one customer
 - LTV (Lifetime Value): Total revenue expected from a customer over their entire relationship with your company
 - Churn: Rate at which customers cancel or stop using your product
 - Burn Rate: Cash spent per month
 - Runway: Months of cash remaining at current burn rate
-

Engineering concepts

- Velocity: Speed at which a team delivers work
 - Technical Debt: Accumulated cost of shortcuts and deferred improvements
 - Deployment: Releasing code to production servers
 - CI/CD (Continuous Integration/Continuous Deployment): Automated testing and deployment pipelines
 - Sprint: Fixed time period (usually 2 weeks) for completing planned work
 - Pull Request: Proposed code change that must be reviewed before being merged
 - Code Review: Peer examination of code before it's merged into the main codebase
-

Framework-specific terms

These are concepts I introduce in this book to help quantify engineering economics:

- ELTV (Employee Lifetime Value): Total value an employee creates over their tenure
- Velocity Tax: Productivity loss from technical debt

I'll define each concept in detail when it first appears, but this glossary gives you a roadmap of the language we'll use.

How to use this book

You don't have to read this sequentially, though the first few chapters set up concepts that later chapters build on. If you're dealing with a specific problem (trying to decide whether to invest in automated testing, wondering if you should fire someone who's not working out, debating whether to outsource development), jump to the relevant chapter. Each one is designed to stand alone while connecting to the larger framework. I've also added a problem guide at the end of the book to help you quickly find the chapter that addresses your specific situation. For executives and investors. This book will help you understand why your engineering team makes the decisions they do and give you better questions to ask. You'll learn to spot the difference between necessary technical investment and engineering gold-plating. You'll understand why "just hire more people" often makes things slower, not faster.

For engineering managers and technical leaders. This book gives you the language to translate your instincts into arguments that resonate with CFOs and boards. You'll learn how to quantify the things you already know matter (code quality, hiring practices, technical debt) in terms that non-technical stakeholders understand.

For individual contributors. This book will help you understand the business context for technical decisions. You'll see why perfect code sometimes isn't worth the cost and why fast code sometimes is. You'll understand what your manager is balancing when they make trade-offs that frustrate you.

A note on numbers

Throughout this book, I use simplified assumptions to make calculations accessible. When I talk about “developer cost per hour”, I'm using total cost to company per month divided by working hours per month. When I give example numbers (like €120,000 salaries or €50,000 feature values), these are illustrative. Your numbers will differ. The goal isn't precision to the euro. It's to understand the relative magnitudes and relationships. Knowing that a bad hire costs eight times their annual salary is more useful than calculating whether it's 7.89x or 8.23x. Knowing that technical debt compounds like interest is more valuable than knowing the exact interest rate for your specific codebase. Use these frameworks as thinking tools, not as rigid formulas.

Two companies we'll follow

To keep the economics concrete, I'll follow two companies through this book: Pace and Surge. They're composites, stitched together from dozens of real teams I've worked with, but their numbers are realistic and their decisions are ones I've watched play out many times. Both build field-service software (the scheduling, dispatch, and invoicing tools that plumbers, electricians, and HVAC companies run their day on). Both raised a Series A the same year. Both started with twelve engineers, a €1.2M engineering budget, around €4M in recurring revenue, and the ability to ship roughly fifteen features a quarter. On paper, on day one, they were the same company. Then they made different choices.

Pace treats engineering decisions as economic decisions. It prices its technical debt, reviews architecture before committing, and spends about a fifth of its capacity keeping the codebase healthy. That discipline isn't free: Pace was slower out of the gate and lost two early deals to a competitor that simply shipped faster. Its CEO, Marta, came up through finance and asks *what does that cost us?* before she asks *is that cool?* Her CTO, Daan, is an evangelist for boring technology whose idea of bragging is reporting zero incidents.

Surge is that competitor. It ships fast, wins on momentum, and has genuinely good instincts about what customers want. Its speed was a real advantage; it bought market share that took years to contest. But Surge never put a price on the shortcuts it took, so it never paid them down. Tom, the CEO, runs on espresso and total-addressable-market slides, and believes every problem is solvable *after the next release*. Nora, his VP of Engineering, is the one who keeps the whole thing standing up, and the one who can never quite get “pay down the debt” onto the roadmap.

Neither company is the hero, and neither is the cautionary tale. Pace sometimes over-engineers and deliberates when it should just ship. Surge is sometimes right to move fast and pay later. The point of following them isn't to crown a winner. It's to watch the same euro spent two different ways, and to see where the bill eventually lands. You'll meet them again in nearly every chapter.

The promise

By the end of this book, you'll be able to walk into a budget meeting and explain why investing €50,000 in platform tooling will generate €200,000 in value over the next year. You'll sit in roadmap discussions and articulate the opportunity cost of building Feature A versus Feature B in terms of customer value

and revenue impact. You'll make the case for paying down technical debt not because "it's the right thing to do" but because the interest rate on that debt is costing you 30% of your team's velocity.

You'll understand why bad hires don't just cost their salary. They destroy the value of your best people. Why bugs in production cost 100x more than bugs caught in development. Why the "expensive" senior engineer who seems slower actually accelerates your entire team. Why manual processes are sometimes more economical than automation. Why outsourcing rarely saves money in the long run. You'll understand how AI tools shift the cost structure of development without changing the underlying economics. You'll know when AI-accelerated development creates value versus when it creates expensive-to-maintain code. You'll recognize that in a world where generating code is cheap, the economic value concentrates on making the right architectural decisions, asking the right questions, and building systems that remain understandable and maintainable.

Most importantly, you'll be able to have these conversations with credibility, using language that resonates with both engineers and executives. You'll bridge the gap that exists in most companies, where engineering is seen as a cost center to be managed rather than the strategic investment it actually is, one that compounds over time. Companies that understand this (that can see engineering decisions through an economic lens while respecting the craft) build better products, retain better people, and create more value with less capital. The ones that don't? They burn through cash, churn through people, and wonder why their competitors are moving faster despite spending less. Let's make sure you're in the first category.

Let's begin.