# THE SOFTWARE DEVELOPER'S COMMON SENSE


## PRINCIPLES, JUDGMENT, AND THE CRAFT OF BUILDING SOFTWARE THAT LASTS

EDWIN R. CALDWELL

# The Software Developer's Common Sense

## Principles, Judgment, and the Craft of Building Software That Lasts

Steve Publications

This book is available at
https://leanpub.com/thesoftwaredeveloperscommonsense

This version was published on 2026-09-04


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Contents

# Principles, Judgment, and the Craft of Building Software That Lasts

**About this book.** Software is full of rules that are right in one context and wrong in another. A feature that is brilliant in a weekend prototype can be a disaster in a decade old banking system, and the same code that thrives in a two person team can rot in a two hundred person organization. The difference between the two is rarely talent. It is judgment. This book teaches the reasoning that experienced developers use when they make those calls: how to trade off simplicity against power, how to think in cost over time, how to recognize when a principle stops applying, and how to build systems that are reliable, maintainable, secure, and easy to change. It is not a list of best practices and it does not tell you which tool to pick. It gives you the durable principles, the mental models, and the practical habits that let you decide for yourself, and the confidence to know why your decision was the right one in that particular situation.

# Introduction: The Case for Common Sense

## A Bug That Cost Millions

On the morning of August 1, 2012, a financial trading firm called Knight Capital pushed a new piece of software to production. The deployment was supposed to activate a small set of automated trading strategies. Instead, it left eight older strategies running that were meant to have been switched off. For forty five minutes, the firm's systems executed more than four million unintended trades, moving over three hundred ninety seven million shares and building up billions of dollars of positions it never meant to hold. By the time anyone could hit the brakes, the firm had lost more than $460 million, a sum that threatened to wipe out a company that had taken seventeen years to build.[1]

The bug was not subtle. No one had to be a genius to miss it. What made it catastrophic was a chain of common sense failures that each experienced developer would recognize. A deployment made weeks earlier had left a dormant defect in the trading router, and a bad switch during the release turned it on. A safeguard that should have caught an unreasonable volume of orders had been allowed to erode over the years. And the firm's own monitoring had actually raised the alarm: ninety seven automated internal alerts flagged the problem before the market even opened, and none of them was acted on. The U.S. Securities and Exchange Commission later found that Knight Capital had failed to maintain adequate controls over its automated trading, and that its pre submission, risk, and code deployment procedures were all inadequate.[1]

This book is, in one sense, about how to avoid that morning. But it is not a book about any one kind of bug. It is about the reasoning that keeps software from becoming a disaster. The developers who avoid the Knight Capital morning are not the ones who memorized the most acronyms. They are the ones who have learned to ask a small set of questions before they commit to a solution, and who trust those questions more than they trust any single rule.

## What Common Sense Is (and Is Not)

Common sense in software is not a gift you are born with, and it is not a personality trait. It is a skill, and like every skill it is built from a mix of principles you can name and a large body of experience that shows you how those principles bend. A junior developer can learn the principles in a year. It takes many more years to learn when the principles are the right ones for the situation in front of you, because the same principle that saves you in one project can sink you in another.

A useful way to separate common sense from its impostors is this. A rule is a statement that tells you what to do in a situation. Best practices are rules that are usually right. Common sense is the ability to notice the situation, to see the constraints and the risks and the hidden costs, and to choose the rule that fits, or to set a rule aside when the circumstances are exceptional. Most of the advice developers absorb is phrased as rules, and most of it is genuinely useful. The danger is not the rules. The danger is applying a rule mechanically, without checking whether the conditions that make it true are actually present in front of you.

Consider the rule "keep your code simple." It is almost always right. But there is a class of problems, such as parsing a hostile network protocol or scheduling thousands of jobs on a shared cluster, where the honest, working solution is complex and where forcing simplicity produces code that looks simple but is secretly full of traps. The common sense move is not to reject complexity. It is to decide, deliberately, where the complexity should live so that it is hidden from the people who have to understand the rest of the system. The book keeps coming back to this pattern, and Chapter 3 is essentially a long argument about it.

The second impostor of common sense is confidence. Inexperienced developers tend to be overconfident: they assume their code is fine, their design is safe, and their requirements are stable. Experienced developers tend to be appropriately uncertain: they know their assumptions are fragile, they expect things to break, and they build in a way that lets them find out early. Both camps can be right in specific cases, but the experienced developer is better calibrated. Part of what this book is about is learning to calibrate.

## The Five Questions Experienced Developers Ask

If you watched an experienced developer work and wrote down what they were thinking, you would find they return again and again to roughly the same questions. I have compressed them into five. They are the load bearing structure of this book, and I will use them as a lens throughout.

The first question is about the problem: what am I actually trying to do, and what problem does this feature, module, or system solve? Most software fails at the level of the question, long before it fails at the level of the code. A feature built to answer the wrong question is a waste, no matter how clean the implementation is. Chapter 2 is devoted to this question and the discipline of expressing requirements so that they survive contact with a changing business.

The second question is about cost over time: how much will this choice cost in a year, in five years, to the people who come after me? The most important cost of any design decision is rarely the cost of writing it. It is the cost of every future change that has to go through it, and every future reader who has to understand it. This book spends a lot of time on that second order cost, because it is where the difference between software that ages well and software that rots is decided.

The third question is about reversibility: if this turns out to be wrong, how hard will it be to change my mind? The software equivalent of a business decision that can be undone is a two way door, and it should be decided fast. A decision that is expensive to undo is a one way door, and it deserves more thought and more testing before you commit. Many technology debates can be settled by asking which kind of door you are standing in front of.

The fourth question is about where the understanding lives: who has to understand this to change it, and how much do they have to hold in their head at once? Software is not done when it runs. It is done when the right people can reason about it, change it, and trust it. A system that only one person understands is a system that will stall the day that person leaves, and the book treats the distribution of understanding as a first order property of a design, not an afterthought.

The fifth question is about measured truth: how do I actually know this works, how do I know it is safe, and how would I find out if it starts to fail? This is the question that connects design to testing, and design to operations. A feature is not done because it behaves correctly in a demo. It is done because

there is evidence for it, and a way to observe it once it is in the hands of real users. Chapters 7, 11, and 12 turn this question into practice.

These five questions are not a checklist you tick off in order. They are a way of thinking that you keep returning to, at different depths, depending on how much is at stake. A one line bug fix mostly needs the fourth and fifth questions. A decision about how to split a system across a hundred services needs all five, asked many times over. The skill is knowing how deep to go.

## How to Use This Book

The book is organized as a climb. It starts with the mindset and the problems, because those are where most of the leverage is. It then moves into the craft of code: simplicity, readability, structure, patterns, and testing. From there it scales outward to systems: architecture, performance, data, reliability, and security. It closes with the work that surrounds the code: technical debt and legacy, the pipeline that ships software, and the teams and long term strategy that decide whether all of the earlier chapters pay off or quietly unravel.

You do not have to read it in order. Each chapter stands on its own, and each one ends by pointing to where its ideas lead. But if you are building intuition from scratch, the order matters, because the later chapters assume the vocabulary and the questions from the earlier ones. The chapters on code assume you are comfortable with the idea that reading code is the main activity of a project's life. The chapters on systems assume you have internalized that most cost is future cost. The chapters on teams and strategy assume you have accepted that software is a team sport played over many years, and that the most durable decisions are the organizational ones.

Throughout the book I will use a mix of durable principles, named laws and patterns, and concrete examples. The principles are the load, and the examples are the weight you can feel. When I name a source, such as Brooks, Ousterhout, Conway, or the SRE practice at Google, it is because the idea has a history and a body of evidence behind it, and because knowing the name lets you go and check it. When I give a number, I have tried to trace it to something verifiable, and I will flag it where the sources disagree. And when a judgment is genuinely mine, rather than something the field has settled, I will say so, because the goal of this book is to teach you to make the judgment for yourself, not to hand you a set of answers to memorize.

The single through line, the thing I want you to carry in your head when you put the book down, is this: good software engineering is the application of common sense under real constraints. The principles in the pages that follow are the raw material. The judgment, the ability to see the constraints and the hidden costs and the second order effects, is the craft. And the craft is built one decision at a time, by asking the right questions and trusting the answers more than you trust any single rule.

The next chapters build that craft. We begin with the mindset, because everything else flows from it.

# Chapter 1: The Common-Sense Mindset

The tools change. The languages change. The frameworks are replaced roughly on a five year clock, and the clever new database that was state of the art when you graduated is already a legacy system. What does not change, and what separates the developer who can be trusted with a real system from the developer who can only follow a tutorial, is the way they think. This chapter builds that thinking from the ground up. It is the foundation for everything that follows, and you will find the ideas in it resurfacing, in slightly different clothes, in almost every later chapter.

## Two Kinds of Complexity

Frederick Brooks, who led the team that built the IBM OS/360 operating system and then wrote *The Mythical Man-Month*, drew a distinction that has held up for half a century. He said that the complexity in a piece of software comes in two kinds. One kind is essential. It is the difficulty that is baked into the problem itself, the difficulty of expressing and testing the idea you are actually trying to build. The other kind is accidental. It is the difficulty that has nothing to do with the problem and everything to do with the tools, the platforms, and the current state of our craft.[2]

The distinction is worth turning over carefully, because it tells you where to spend your energy and, just as importantly, where not to. Consider a program that prices airline tickets. The essential complexity is in the rules of fare construction, the logic of how a price is computed and constrained. That difficulty is real. It is there because the problem is hard. The accidental complexity is in the layers between that logic and the machine: the operating system, the network, the database driver, the serialization format, the web framework. None of those layers is essential to the problem of pricing a ticket, but they are not free, and they are where most of the bugs and most of the maintenance effort actually live.

Brooks made a famous claim that is worth taking seriously: there is no silver bullet. There is no single technology or management trick that will give an order of magnitude, a ten fold, jump in productivity or reliability within a decade.[2] The reason is that a large part of software's difficulty is essential. You can automate the accidental parts, and we have made enormous progress doing exactly that, but you cannot automate the act of figuring out what you are supposed to build and proving it is correct. That part must be done by people, and it must be done by people who understand the problem.

The practical consequence of the essential versus accidental split is a discipline. Before you reach for a tool or a pattern, ask which kind of complexity it is fighting. If a proposed layer is adding accidental complexity, ask whether you truly need it, because every layer of machinery you adopt is a place where things can break and a place where future readers have to spend attention. If a difficulty turns out to be essential, do not waste time looking for a tool that will make it disappear, because it will not. Instead, find a way to contain it, so that the essential complexity is visible and manageable in one place rather than smeared across the system. Chapter 3, on simplicity, is really an application of this discipline: it is the practice of minimizing accidental complexity and confining the essential kind.

A useful habit follows from this. When you are confused about a piece of code or a design, try to name what kind of complexity is in front of you. If you cannot, that is often a sign that you have not yet understood the problem well enough, and the right move is to stop and go understand the problem, not to start reaching for a solution. Experienced developers do this constantly, and it is one of the quietest tells that separates them from the rest.

## Judgments, Not Rules

Most of what we learn in this profession is phrased as rules. Use a repository instead of querying in the controller. Add a test for every branch. Keep your functions short. Use an ORM. Avoid global state. Each of these is a rule, and each one is usually right. The trap is to mistake the rule for the reason. The rule is a compressed summary of a judgment that someone made under specific conditions. The conditions are what matter, and the conditions are the part that is easiest to forget.

Take "add a test for every branch." The rule is right as a general policy, because it reflects the judgment that untested branches are where uncaught

bugs hide, and that the cost of a test is usually far less than the cost of the bug it would have caught. But the rule is also incomplete. It does not tell you what counts as a branch, or what kind of test is worth writing for a given branch, or that testing the wrong behavior can be worse than testing nothing, because it tells you to stop looking. Chapter 7 deals with this in detail, but the mindset is the same here: the rule is a starting point for a judgment, not the end of it.

The discipline of judgment looks like this. For any rule you reach for, you should be able to answer three questions. First, what problem is the rule solving? If you cannot name the problem, you are unlikely to know whether it applies. Second, what conditions make the rule true? Every good rule is true because of some circumstance, such as the cost of change being high, or the risk of a bug being severe, and if those circumstances are not present, the rule may not be. Third, what would make the rule wrong in this particular case? This third question is the one inexperienced developers skip, and it is the one that saves you most often. If you can name the case where a rule is wrong, you are not a rule follower. You are a reasoner.

This is why the most reliable developers in a codebase are often the ones with the most experience, not the most credentials. Experience is, in a large part, a library of conditions. It is a collection of memories of when a rule held and when it did not. No amount of reading will build that library for you. Only making decisions, living with their consequences, and paying attention to how things aged will. What this book can do is point you at the principles and the conditions, so that when you do make the decisions, you are not making them in the dark.

There is a corollary to this that matters in a team. A shared body of common sense is a kind of asset. When a team develops a common vocabulary and a shared set of expectations, code review gets faster, new people ramp up faster, and the number of surprising decisions goes down. This is why the later chapters on communication and review are not a distraction from the technical core. They are part of the same craft, because a team's common sense compounds in the same way an individual's does.

## One-Way Doors and Two-Way Doors

Some decisions are easy to reverse. Some are not. This distinction is so useful that it is worth making into a habit. A decision is a two way door if you can

undo it cheaply: you change your mind, you roll the change back, you throw the code away and try again. A decision is a one way door if undoing it would be expensive: you have built a data migration that will have to be reversed, you have signed a multi year vendor contract, you have chosen a database and modeled a year of business logic on top of it, you have restructured the organization around a set of services.

The practical rule is that two way doors should be decided fast, and one way doors should be decided carefully. When you are standing in front of a two way door, the cost of being wrong is low, so the right move is to pick an option, commit to it, and move on. Deliberating for a month about which of three logging libraries to use is a waste of time, because all three will do and you can switch later if one disappoints you. When you are standing in front of a one way door, the cost of being wrong is high, so the right move is to slow down, to get more information, to prototype, to get a second opinion, and to design the decision so that the most painful part is the most reversible part you can make it.

Most of the anxiety in software architecture comes from treating two way doors as one way doors, or from not noticing which kind of door you are standing in front of. A team that spends a quarter deciding the shape of an internal API that only two services will call is probably in front of a two way door, and the delay is costing more than a bad guess would have. A team that charges straight into a database choice for a system that will hold regulatory records for thirty years is in front of a one way door, and the lack of deliberation is exactly the problem.

The trick is that the kind of door is not always obvious, and it can change. A decision that is a two way door when you have one user can become a one way door when you have a million, because the cost of changing it scales with the number of things that depend on it. This is why one of the five questions in the introduction is cost over time. The same choice is a different kind of door in different contexts, and the common sense move is to ask how the door will change as the system grows, not just to classify it in its current state.

## Cost Over Time

The deepest idea in this book, and the one I want you to internalize, is that the cost of a decision is paid over time, not at the moment you make it. The cost

of writing a line of code is nearly zero. The cost of that line over the life of the system, which is every future change that touches it, every future reader who has to understand it, every future bug that traces back to a misunderstanding of it, can be enormous. Software is not a product you build and ship. It is a living thing you maintain for years, and most of the cost you will ever incur was locked in by design decisions made long before the system was in trouble.

This is the second order effect that Brooks was gesturing at when he distinguished essential from accidental complexity, and it is the reason the mindset of the experienced developer is so different from the mindset of the novice. The novice optimizes for the cost of getting the current feature done. The experienced developer optimizes for the cost of the feature over the next five years. They know that a slightly more expensive solution now, one that is easier to change and easier to understand, is usually a bargain, because it lowers the price of every future change.

You can see this in a concrete example. Imagine you need to send users an email. The cheapest possible solution is to drop an email sending call directly into your order handling code. It works, it is fast to write, and it costs nothing today. But now every change to your order logic requires someone to understand the email code, and every change to how you send email requires someone to dig through the order logic, and any failure in sending an email can corrupt your order state. The common sense solution is to separate sending email from processing orders, to put the sending behind a small interface, and to make the two independent. It costs a little more up front. It pays for itself every time someone changes either piece. The cost over time of the simple direct approach is higher, even though the cost up front was lower.

The danger with this mindset, which is worth naming, is that it can be used to justify anything. It is very easy to wrap a bad design in the language of long term thinking and to call every complication a future investment. The common sense check is that the future cost you are claiming must be real and proportional, not invented. You should be able to name the specific future changes you are protecting against, and you should be able to show that the extra cost you are paying now is less than the cost of those changes. If you cannot, you are probably paying for complexity that will not earn its keep, and the simple solution is the right one. Chapter 3 is where you learn to make that check honestly, because simplicity is the discipline that keeps cost over time from becoming an excuse for over building.

## Knowing What You Don't Know

The last ingredient in the mindset is the hardest to teach and the most valuable: knowing the edge of your own knowledge. The most dangerous person in a system is not the person who knows nothing. It is the person who is confident about a domain they have not actually learned, because they will make the one way door decisions in that domain with the calm of a person who thinks they are in front of a two way door.

The fix is a mix of humility and method. Humility means accepting that you are likely to be wrong, especially at the edges, and building decisions so that being wrong is survivable. Method means giving yourself ways to find out: small experiments, prototypes, measurements, and the habit of reading the code and the documentation of the systems you depend on rather than guessing at them. This is the mindset that connects to the fifth question in the introduction, the question of measured truth. You do not get to be confident because you feel confident. You get to be confident because you have evidence, and you know where the evidence comes from.

A good sign of a mature mindset is how a developer talks about their own uncertainty. The novice says "this should work" and means it as a prediction. The experienced developer says "this should work, and here is how I would find out in ten minutes if it doesn't." The first statement is a hope. The second is a plan. The whole book is, at heart, an argument for making your hopes into plans, because plans are the things you can test, and testing is the thing that separates a system you can trust from a system you can only hope about.

The mindset in this chapter is not a list of attitudes to adopt. It is a set of habits of thought: ask which kind of complexity you are facing, remember that every rule is a judgment with hidden conditions, classify your decisions by how reversible they are, pay attention to the cost that will be paid later, and treat your own confidence as something you have to earn with evidence. The next chapter applies this mindset to the first and most important decision you will ever make in a project, which is the decision about what you are actually building. If the mindset is the engine, the problem is the road, and the road is where this book goes next.

# Chapter 2: Problems, Requirements, and Change

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Real Problem Is Rarely the Stated Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Eliciting and Expressing Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Scope, YAGNI, and the Planning Dilemma

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Estimation and Prioritization as Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 3: Simplicity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Compounding Cost of Complexity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## What Simple Actually Means

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Techniques for Staying Simple

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## When to Complicate on Purpose

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 4: Code Quality and Readability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Code Is Read More Than It Is Written

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Naming and Local Clarity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Small Functions and Flat Control Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Same Logic, Three Ways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 5: Structure, Abstraction, Modularity, Coupling, and Cohesion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Why We Abstract, and What It Costs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Modules and Single Responsibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Deep Modules and Information Hiding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Coupling, Cohesion, and the Direction of Dependence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 6: Design Patterns, Anti-Patterns, and Judgment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Patterns as a Shared Language

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## A Practical Tour of Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Anti-Patterns That Do the Most Damage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Meta-Pattern of Restraint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 7: Testing and Confidence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## What Testing Is Actually For

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Test Pyramid and Its Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Test-Driven Development, Honestly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Flaky Tests and the Cost of Untrustworthy Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 8: APIs, Interfaces, and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Designing Interfaces People Will Depend On

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Contracts, Versioning, and Backward Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Managing the Dependencies You Take In

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Frameworks, Libraries, and the Framework Tax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 9: Architecture, System Design, and Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## What Architecture Is and Isn't

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Monoliths, Services, and the Shape of the System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Conway's Law and Organizational Reality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Fallacies of Distributed Computing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Designing for Failure and Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 10: Performance and Scalability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Measure Before You Optimize

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Bottlenecks, and How They Move

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Caching, Batching, and Layering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Throughput, Latency, and the Tail

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## When to Scale, and When to Stop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 11: Data and Databases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Data Is the Thing You Cannot Get Back

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Modeling Before You Pick a Database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## SQL, NoSQL, and Consistency, Honestly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Migrations, Replication, and Sharding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 12: Reliability, Observability, and Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Reliability Is Designed In, Not Bolted On

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Three Pillars of Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## A Debugging Method That Scales

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Post-Mortems and the Culture of Failure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 13: Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Threat Modeling Before You Build

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The OWASP Map

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Least Privilege and Defense in Depth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Human and Supply-Chain Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Cost Asymmetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 14: Technical Debt, Refactoring, and Legacy Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## What Technical Debt Really Is (and Isn't)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Refactoring as Risky, Managed Change

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## The Strangler Fig and Living with Legacy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Why "Just Rewrite It" Usually Loses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 15: Building, Shipping, and Operating

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Version Control as a Discipline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Continuous Integration and Delivery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Deployment Strategies and Reversibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Infrastructure as Code and Environment Parity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Chapter 16: Teams, Review, Communication, and the Long-Term Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Code Review Done Right

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Estimation, Planning, and Commitments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Communication and the Bus Factor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## What Actually Drives Developer Productivity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

## Long-Term Engineering Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# Conclusion: The Common Sense That Compounds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.

# References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesoftwaredeveloperscommonsense.