The Mythical Agent-Month

Abdul Kavungal (AK)

AWARD WINNING CREATIONS

The Mythical Agent-Month

© 2026 Abdul Kavungal (AK). Published by Award Winning Creations.

All rights reserved. No part of this book may be reproduced without permission, except for brief quotations in reviews and scholarly works.

This is a work of non-fiction. Case histories are drawn from published investigations, audits, regulatory findings and research, cited in the Notes on Sources. Composite and hypothetical examples are identified as such in the text.

Display type: Keyslate, from the TyAk type foundry. Text type: Crimson Text.

First edition, 2026.

Contents

Introduction The Tempting Arithmetic 4

Part One: The Law and the Loop 14

Chapter 1 Nine Women, One Month 15

Chapter 2 The Loop Behind the Law 27

Chapter 3 Cheap Hands 38

Chapter 4 Essence and Accident 47

Part Two: How Lateness Hides 58

Chapter 5 The Reassuring Story 59

Chapter 6 Preoccupation with Small Failures 69

Chapter 7 Reluctance to Simplify 79

Chapter 8 Sensitivity to the Sharp End 90

Chapter 9 Ninety Percent Done 100

Part Three: Conceptual Integrity When Workers Are Cheap 109

Chapter 10 The Cathedral and the Committee 110

Chapter 11 The Architect's Playbook 119

Chapter 12 Every System a Second System 128

Chapter 13 Documentation as a City Plan 138

Chapter 14 The Surgical Team, Revisited 149

Part Four: Build It Small, Test It Early 159

Chapter 15 The Walking Skeleton 160

Chapter 16 A Boat Full of Holes 170

Chapter 17 Limiting the Blast Radius 180

Chapter 18 Resilience in the Unfolding Crisis 189

Chapter 19 Leverage Points for the Late Project 198

Chapter 20 A Rescue in Practice 208

Conclusion What Can Rescue a Late Project 217

Notes on Sources 223

References 229

Glossary 232

INTRODUCTION

The Tempting Arithmetic

The meeting usually starts with a chart. A line that was supposed to slope gently down to zero by the end of the quarter has flattened into a plateau. The burndown has stopped burning. Someone has added a red band across the top of the slide, and in that band are the words every engineering leader learns to dread: at current velocity, the release slips by eleven weeks.

For most of the history of software, the next move in that meeting was predictable. Somebody would propose more people. Borrow two engineers from the platform group. Bring in the contractors who did good work last year. Pull the strongest developer off the mobile app for a month. The arithmetic looked irresistible. If eight people produce a certain amount of work in a week, twelve people should produce half again as much, and eleven weeks of slip should shrink to something survivable.

In 1975 Frederick P. Brooks Jr. published a short book of essays that told the industry, in plain language, that this arithmetic was wrong. Brooks had managed the development of IBM's Operating System/360 in the middle of the 1960s, one of the largest software efforts attempted up to that point, and he had watched the arithmetic fail at scale. His book, The Mythical Man-Month, gave the failure a name that has outlived almost everything else written about software management in that decade. Brooks's Law, in his own phrasing, says that adding manpower to a late software project makes it later.

Fifty years on, the meeting has changed. The chart is the same and the red band is the same, but the proposal is different. Nobody suggests borrowing two engineers from the platform group. Somebody suggests spinning up twenty agents.

It is a reasonable suggestion on its face. The AI coding agents available to engineering teams in 2026 can read a repository, propose a plan, write code across many files, run the test suite, read the failures, and try again. They do not need a desk, a laptop, a security badge or a three-week onboarding programme. They do not get tired at four in the afternoon. They can be started in parallel, each on its own branch, each chewing on its own ticket. The cost of an additional worker, which for Brooks was measured in salary, recruitment time and months of ramp-up, has collapsed toward the price of compute. If Brooks's Law was a statement about the cost of adding people, and adding people has become nearly free, surely the law has finally been repealed.

This book argues that it has not. It argues something stronger: that the arrival of cheap workers makes Brooks's insight more important, not less, because the forces that made his law true were never mainly about the price of a programmer. They were about the structure of the work. They were about how much of a software project is the shaping of a single coherent idea, and how much of that shaping can be divided among many hands. They were about the feedback loops that a late project sets in motion, the delays that hide those loops from the people running them, and the stories organisations tell themselves to avoid looking at the evidence until it is too late to act cheaply. None of those forces disappears when the new hands are made of silicon. Several of them get worse.

Three books, one argument

The argument here is built from three books written decades apart by people who, as far as I know, never thought of themselves as collaborators.

The first is Brooks's. The Mythical Man-Month is a collection of essays about building large software systems, written by a manager who had done it and been bruised by it. It is often remembered for a single law, but its deeper subject is what Brooks called conceptual integrity: the idea that a system should reflect one coherent set of design ideas rather than a pile of good but uncoordinated ones. Brooks believed that conceptual integrity was the most important consideration in system design, and he believed it

was achieved by separating the architecture of a system, which a small number of minds must hold, from its implementation, which many hands can share. In the 1986 essay No Silver Bullet, later reprinted in the anniversary edition of the book, he added a second distinction that sits at the centre of this book: the difference between the essential complexity of software, the conceptual structure of the problem itself, and its accidental complexity, the difficulties that attend producing it with the tools of the day.

The second is Donella Meadows's Thinking in Systems, published in 2008, seven years after her death, from a manuscript she had been refining for years. Meadows was a systems scientist, one of the authors of The Limits to Growth, and a gifted explainer of how stocks, flows and feedback loops produce behaviour that no individual participant intends. Her vocabulary gives us a way to see Brooks's Law as something other than a proverb. It is a reinforcing feedback loop, wrapped around a stock of unfinished work, with delays that hide the loop from the people inside it. Once you can draw the loop, you can ask the question Meadows spent her career asking: where are the places to intervene?

The third is Managing the Unexpected, by the organisational psychologist Karl E. Weick and his long-time collaborator Kathleen M. Sutcliffe. Weick and Sutcliffe studied what they called high reliability organisations: aircraft carrier flight decks, nuclear power plants, wildland firefighting crews, hospital emergency departments. These are places where small errors can compound into catastrophe, and where the people who do well have learned habits of attention that most organisations lack. They notice small anomalies and treat them as clues. They refuse to accept comfortable labels for things they do not yet understand. They stay close to the front line, where the real work is done. They improvise when the plan stops matching the world. And they move decisions toward the people with the most relevant expertise, whatever their rank.

Each book, on its own, is about something different. Brooks is about building software. Meadows is about how systems behave. Weick and Sutcliffe are about how organisations notice trouble. Put them together, and they describe a single phenomenon with unusual completeness: the

late project. Brooks tells us why adding workers makes it later. Meadows tells us why the people running it cannot see that happening until the evidence is overwhelming. Weick and Sutcliffe tell us why, even when the evidence is visible, organisations so often explain it away.

What changes when workers are cheap

The temptation to treat AI agents as a repeal of Brooks's Law rests on a specific idea: that the cost Brooks identified was mostly the cost of human beings. Humans need training. Humans need to talk to each other. Humans hold knowledge in their heads and lose it when they leave. Remove the humans, or at least remove the need to hire more of them, and the cost goes away.

The first half of this book takes that idea apart carefully, because some of it is true. Agents do remove certain costs. An agent working on a well-specified, well-tested, well-bounded task can produce in minutes what would have taken a person an afternoon. There are categories of work where the speedup is real and large: boilerplate, migrations with clear rules, test scaffolding, documentation of code that already exists, the translation of a precise specification into a first draft. These are, in Brooks's terms, mostly accidental complexity. They are the part of software that is about representing an idea in a particular language and toolchain, rather than the part that is about having the idea in the first place.

But Brooks's Law was never primarily about the accidental part. Brooks gave three reasons the law held. New people need time to become productive, and the people already on the project must spend their own time training them. Work that is not perfectly divisible requires communication among the workers, and the number of communication paths grows much faster than the number of workers. And the act of repartitioning work to accommodate the newcomers is itself work, which consumes time the project does not have. When you look closely at each of these, you find that agents change their shape without removing them. An agent's ramp-up does not happen once, at hiring, but at the start of every session, every time it is handed a task without the context it needs. The

communication burden moves from conversation to review: every line an agent writes has to be read, understood, judged and integrated by someone who holds the design in their head. And the repartitioning work grows rather than shrinks, because the smaller and more numerous the pieces, the more effort it takes to define their boundaries cleanly enough that independent workers will not collide.

There is early evidence that these costs are real and that they are underestimated by the people bearing them. In a randomised controlled trial run in the first half of 2025 by the research group METR, experienced open-source developers working in mature repositories they knew well took 19 percent longer to complete tasks when they were allowed to use AI tools than when they were not. Before the study, the same developers had forecast that AI would make them 24 percent faster. Afterward, having just been measurably slowed down, they estimated that it had made them about 20 percent faster. The researchers were careful to say that their result applied to a particular setting and should not be generalised, and they have since published updated work on newer tools. The detail that matters for this book is not the size of the slowdown. It is the gap between what the developers experienced and what they believed.

The 2024 Accelerate State of DevOps report from Google's DORA research programme found a similar pattern at the level of teams. Respondents who relied more heavily on AI reported better documentation, better code quality and faster reviews. Yet the report estimated that every 25 percent increase in AI adoption was associated with a 1.5 percent reduction in software delivery throughput and a 7.2 percent reduction in delivery stability. The authors offered a hypothesis that should be printed on the wall of every engineering leader's office: that faster code generation had tempted teams to forget the importance of small batches, and that larger changes, as DORA had found for years, are slower and more prone to instability.

These findings do not prove that agents are useless or that Brooks's Law applies to them in exactly the form he wrote down. They prove something more modest and more useful: that the intuition that more workers

produce proportionally more output is as unreliable now as it was in 1975, and that the people closest to the work are not reliable judges of whether it is helping.

The warning layer

Brooks and Meadows together explain why adding workers to a late project tends to make it later. Neither of them fully explains why intelligent, experienced people keep doing it.

That is where Weick and Sutcliffe come in. Their research begins from an observation that every incident investigator eventually makes: disasters rarely arrive without warning. They arrive after a long series of small signals that somebody noticed, discussed, and explained away. The signal was routine. It was operator error. It was a flaky test. It was a one-off. Each explanation was plausible on its own. Each one made the signal easier to ignore. And each one moved the organisation a little further from the point where correction would have been cheap.

Late software projects are among the most richly documented examples of this pattern. When the Office of Inspector General at the US Department of Health and Human Services reviewed the troubled 2013 launch of HealthCare.gov, it found that the Centers for Medicare and Medicaid Services had received 18 documented warnings about the project between July 2011 and July 2013. The warnings were detailed and formally submitted. They were not shared widely, and no single person had a comprehensive view of the project's poor progress. The inspector general's description of what happened next could serve as an epigraph for this book: as the project degraded and its problems became better known, officials and staff became desensitised to bad news about progress.

Desensitised to bad news. That phrase describes the condition in which the decision to add workers usually gets made. By the time the chart in the meeting has a red band on it, the project has typically been late for a long time. The lateness accumulated a day at a time, as Brooks put it, and at each step there was a reasonable explanation for why this particular slip did not mean the plan was broken. When the evidence finally becomes

undeniable, the organisation is primed to look for a fix that will not require it to admit what the evidence implies. Adding workers is that kind of fix. It looks like action. It requires no one to say that the design was wrong, the estimate was fantasy, or the project should be rescoped. And with agents it is cheaper and faster than ever, which means it can be reached for earlier and more often.

What can rescue a late project

The subtitle of this book promises an answer to two questions: why AI agents cannot rescue a late project, and what can. The second question deserves as much attention as the first, and most of the second half of the book is devoted to it.

The answer, in brief, is that late projects are rescued by the same things that keep projects from becoming late in the first place, applied with more discipline because the margin for error has gone. The first is conceptual integrity, held by a small number of people who understand the whole design and are empowered to defend it. Brooks believed this was the most important property of a system, and it becomes more important when the cost of implementation falls, because cheap implementation multiplies the number of ways a design can be pulled apart. When every engineer can direct a fleet of agents, every engineer is effectively an architect of something, and the system will reflect as many design philosophies as there are people directing fleets unless somebody holds the line.

The second is an end-to-end skeleton built early: a thin, working version of the whole system, with placeholders where the real components will go, that runs from input to output on the first day and gets fleshed out from there. The term walking skeleton comes from Alistair Cockburn, and the idea behind it echoes a principle the systems writer John Gall stated in the 1970s, that a complex system that works is invariably found to have evolved from a simple system that worked. A skeleton turns integration from a terrifying final phase into a continuous activity. It gives agents

something concrete to plug into. And it gives the humans running the project the one thing late projects most lack: early, honest evidence about whether the pieces fit.

The third is testing that comes before launch, and checks that cannot be modified by the change they judge. Shipping software that has not been tested end to end is like launching a boat full of holes. The image is more apt than it first seems, because modern systems are not single boats but fleets, connected to each other in ways that let one leak sink many. When the cybersecurity company CrowdStrike pushed a faulty content update to Windows machines in July 2024, the crash spread across millions of computers in hours. Its root cause analysis described a mismatch between the 21 input fields a template defined and the 20 values the sensor actually supplied, a mismatch that had evaded multiple layers of validation and testing. Agents produce more changes faster. That makes the gate between a change and the world more important, not less.

The fourth is an organisation that hears bad news while it is still cheap. This is the Weick and Sutcliffe contribution, translated into the specific setting of software teams that now include machine contributors. It means treating small anomalies as information, including the small anomalies in agent behaviour that are easy to dismiss as quirks. It means resisting the labels that end inquiry. It means staying close to the sharp end, where code meets production and production meets users. And it means building the capacity to improvise when the plan breaks, which it will.

None of these is new. All of them are harder to practise when workers are cheap, because cheap workers make it easy to substitute motion for progress. The organisations that will do well in the next decade of software are not the ones that deploy the most agents. They are the ones that understand what agents can and cannot reduce, and design their projects accordingly.

How this book is organised

The book is divided into four parts.

Part One, The Law and the Loop, starts with Brooks's original argument and the experience of OS/360 that produced it. It then redraws Brooks's Law in Meadows's language of stocks, flows and feedback, examines what genuinely changes when workers become cheap, and sets out the distinction between essential and accidental complexity that governs everything that follows.

Part Two, How Lateness Hides, turns to Weick and Sutcliffe. It looks at the reassuring stories organisations tell about early slippage, the discipline of preoccupation with small failures, the danger of comfortable simplifications, the importance of staying close to the sharp end, and the difficulty of measuring progress honestly when agents generate so much activity. Its cases include HealthCare.gov, the Mars Climate Orbiter, the Therac-25 radiation therapy machine, the Boeing 737 MAX, and England's National Programme for IT.

Part Three, Conceptual Integrity When Workers Are Cheap, returns to Brooks's central idea and asks what it means for teams in which humans direct agents. It covers the architect's role, the evolving design playbook that keeps a team of humans and machines building the same thing, the second-system temptation that cheap features revive, documentation understood as a city planner's blueprint rather than an afterthought, and new shapes for Brooks's surgical team.

Part Four, Build It Small, Test It Early, is the practical core. It covers the walking skeleton, testing before launch, limiting the blast radius of changes and of agents themselves, resilience when the crisis is already unfolding, a reworking of Meadows's famous list of leverage points for the specific problem of the late project, and a composite account of what a rescue looks like in practice.

A short conclusion draws the threads together, followed by notes on sources and a glossary.

Throughout, I have tried to stay close to the documented record. The cases in this book are drawn from published investigations, inspector general reports, regulatory filings, government audits and peer-reviewed research. Where I describe what happened, I have tried to describe it as the primary sources do. Where I quote Brooks, Meadows or Weick and

Sutcliffe, the words are theirs; elsewhere I paraphrase. Where the record is uncertain, I say so. Engineering leaders have been badly served by a literature of confident anecdotes, and the subject is important enough to deserve better.

One more note before we begin. This is a book about AI agents, but it is not a book about any particular model, vendor or tool. The products will change faster than any book can track. The structure of the problem will not. Brooks wrote about IBM mainframes and punched cards, and his book is still useful because he wrote about the structure of the work rather than the tools of the moment. I have tried to follow his example. If the specific capabilities described here look quaint in five years, I hope the loops, the stories and the remedies will not.

PART ONE

The Law and the Loop

Brooks's Law was written for human teams. These four chapters ask what it actually claims, how it looks when drawn as a system of stocks, flows and feedback loops, what really changes when workers become cheap, and why the difference between essential and accidental complexity decides whether AI agents help.

CHAPTER 1

Nine Women, One Month

On 7 April 1964, IBM announced System/360, a family of computers that was meant to replace the company's tangle of incompatible machines with a single architecture. A customer could start with a small model and move up to a large one without rewriting their programs. That promise, which sounds ordinary now, was a radical bet at the time, and it rested on software as much as hardware. The machines would need an operating system that could run across the whole family, manage jobs and devices in ways no previous IBM system had attempted, and support customers whose needs ranged from small business accounting to large scientific computing.

Frederick Brooks had led the hardware development of System/360. In 1964 he became manager of the operating system effort, OS/360. He later described the experience as very educational and very frustrating, in roughly equal measure. The system that eventually emerged had real strengths. Brooks singled out device-independent input and output and external library management as innovations that were widely copied, and he noted that OS/360 became reliable, reasonably efficient and very versatile, and achieved widespread use. But he was candid about the rest. In the preface to the original edition of The Mythical Man-Month, he wrote that the product was late, took more memory than planned, cost several times the estimate, and did not perform very well until several releases after the first. He laid most of the design flaws, which dated from 1964 and 1965, at his own door.

The book he wrote a decade later was an attempt to understand why. Brooks had left IBM in 1965 to found the computer science department at the University of North Carolina at Chapel Hill, and he had spent the intervening years thinking about the gap between the way large software projects were managed and the way they actually behaved. The Mythical Man-Month, published in 1975, is a short book of essays, and it reads less like a textbook than like a series of conversations with a colleague who has made every mistake you are about to make and would like to spare you some of them.

Its most famous essay is the second, from which the book takes its title. It begins with an observation that should be uncontroversial but somehow never is: more software projects go wrong for lack of calendar time than for all other causes combined. Brooks then asks why that should be, and his answer is a dissection of the unit that managers used, and still use, to plan software work.

The fallacy in the unit

The man-month, which we would now call the person-month, is a unit of effort. One person working for one month produces one person-month. Ten people working for a month produce ten. Effort, measured this way, is perfectly fungible: a task estimated at twelve person-months can apparently be completed by one person in a year, by four people in a quarter, or by twelve people in a month.

Brooks's point was that this fungibility holds only under a very specific condition. People and months are interchangeable when a task can be divided among many workers with no communication among them. Picking cotton works this way. So does harvesting wheat, or stuffing envelopes. Each worker can proceed independently, and the total output is the sum of what each produces. Add a worker and the work goes faster in direct proportion.

Very little software looks like that. Brooks identified a second kind of task, one that cannot be divided at all because of sequential constraints. Here his illustration became the most quoted sentence in the book: the

bearing of a child takes nine months, no matter how many women are assigned. Some work simply has a fixed duration, and adding workers does nothing to shorten it. In software, the sequential parts include debugging, which often requires one mind to follow a chain of causes from symptom to root, and the many steps where one piece of work cannot begin until another is finished and understood.

Between these extremes lies most real software work: tasks that can be partitioned, but whose pieces need to communicate with each other. This is the case that matters, and it is the case where the man-month misleads most badly. When you divide a task among several people, each of them must understand enough of the others' work to make their own piece fit. That understanding has to be built and maintained, and it costs time. Brooks broke the cost into two parts.

The first is training. Every person added to a project needs to learn the technology, the goals of the effort, the overall strategy and the plan of work. This training cannot be partitioned, so it adds linearly with the number of new workers. Worse, it usually has to be delivered by people who are already on the project, which means the existing team's output falls while the newcomers are coming up to speed.

The second is intercommunication. If every part of a task must be coordinated with every other part, the number of pairwise relationships among workers grows much faster than the number of workers. With n people, there are n(n-1)/2 possible pairs. Three people have three pairs. Four have six. Ten have forty-five. Twenty have one hundred and ninety. Not every pair needs to talk all the time, and good structure reduces how many must talk at all. But the underlying pattern holds: the coordination burden of a team grows faster than its size, and beyond some point the added coordination consumes more time than the added worker contributes.

Put these together and you get a curve that looks nothing like the one the man-month implies. Instead of time falling smoothly as you add people, it falls at first, then flattens, then begins to rise again as the

overhead of training and communication overwhelms the extra capacity. For tasks with complex interrelationships, which describes most large systems, the rise can come early.

Optimism and the schedule

Before Brooks gets to his law, he spends some pages on a quieter cause of lateness, and it is worth pausing on, because it is the soil in which the decision to add workers grows.

All programmers are optimists, Brooks wrote. He did not mean it as an insult. He meant that the medium of programming is unusually forgiving of the imagination. A sculptor working in stone, or a carpenter working in wood, meets resistance from the material at every step, and that resistance keeps their estimates honest. A programmer works in something closer to pure thought. Ideas seem to translate directly into reality, and so programmers tend to estimate as if the translation will be smooth. It is not. Every implementation reveals flaws in the idea, and the time to find and fix those flaws is the time that schedules routinely leave out.

Brooks's own rule of thumb for scheduling a software task, which he offered as the result of experience rather than theory, gave a third of the schedule to planning, a sixth to coding, a quarter to component testing and early system testing, and a quarter to system testing with all components in hand. The proportions surprised readers then and still surprise them now. Coding, the activity most people picture when they think about building software, gets the smallest share. Testing and integration get half. Brooks observed that few projects actually allowed half their schedule for testing, and that most of the projects that failed to do so ended up spending that time anyway, just later and less pleasantly.

This rule has a direct bearing on the argument of this book. If coding is a sixth of the work, then even an infinitely fast coder can shorten a project by at most a sixth. The other five sixths are planning, understanding, testing, integrating and debugging, and those are precisely the activities that require the communication and the coherent understanding that additional workers struggle to provide. An AI agent that writes code ten

times faster than a person speeds up one slice of the schedule. Whether it speeds up the others depends on things that have little to do with how fast it types.

The regenerative disaster

Brooks illustrated what happens when a project falls behind with a worked example that rewards close reading. Suppose a task is estimated at twelve person-months and assigned to three people for four months. There are four milestones, one at the end of each month. The first milestone is not reached until the end of the second month. What should the manager do?

Brooks walked through the options. If the manager assumes only the first part of the task was misestimated, then nine person-months of work remain, and two months are left. To finish on time requires four and a half people, so two more must be added. If the manager assumes the whole estimate was uniformly low, the arithmetic is worse: eighteen person-months remain, needing nine people, so six must be added. In either case, the new people must be trained, which takes time from the existing staff. The work must be repartitioned, which takes more time. And the new team must now coordinate across more pairs than the old one. Brooks's conclusion was that in his example, adding two people at month two, after training and repartitioning, would leave the project at least as late as if no one had been added. Adding more would make it worse.

He then described what happens next in real projects. The project, now with more people, falls behind again. The manager, facing the same chart with a deeper red band, adds still more people. Each addition increases the training load and the coordination load, which reduces the productivity of everyone already present, which makes the project later, which prompts another addition. Brooks called this a regenerative schedule disaster. It feeds on itself.

It was here that he stated the law. Oversimplifying outrageously, he wrote, we state Brooks's Law: adding manpower to a late software project makes it later. He then explained the reasoning compactly. The number of months of a project depends on its sequential constraints. The maximum

number of people depends on the number of independent subtasks. From these two quantities one can derive schedules using fewer people and more months. The only risk is product obsolescence. One cannot, however, get workable schedules using more people and fewer months.

The phrase "oversimplifying outrageously" is easy to skip, but it matters. Brooks knew his law was a heuristic, not a theorem. He offered it as a compressed statement of a dynamic that held in the conditions he had seen, and he trusted readers to understand that conditions vary.

Testing the law

Researchers did test it. The most careful early attempt came from Tarek Abdel-Hamid and Stuart Madnick at MIT, who built a system dynamics model of software project management in the 1980s, based on field interviews with project managers and a large body of published empirical findings. Their work, which culminated in the 1991 book Software Project Dynamics, used exactly the kind of feedback modelling that Donella Meadows would later explain for general readers, and it is the first bridge between Brooks's world and hers.

Abdel-Hamid and Madnick drew the regenerative disaster as a loop. Adding people raises communication and training overheads, which dilutes the team's productivity, which slows progress, which delays the project further, which can trigger another round of hiring. Then they ran experiments on their model, calibrated to medium-sized application projects, to see whether the law held. Their finding was more nuanced than either Brooks's admirers or his critics might have expected. Adding people to a late project of that type always made it more costly, because the training and communication overheads reduced average productivity. But it did not always make the project later. For the schedule itself to suffer, the drop in productivity had to be severe enough and come late enough in the project for each new person's net contribution to be negative. In their simulations, that happened under very aggressive staffing policies, where management kept adding people into the final stages of testing.

This is useful refinement, not refutation. It tells us that Brooks's Law describes a region of the space of possible projects rather than the whole of it. Early additions, made while the work is still being partitioned and before the design has hardened, can help. Late additions, made under pressure into a project whose structure is set and whose remaining work is mostly integration and debugging, are the dangerous ones. The danger is greatest exactly when the temptation is strongest, because the moment a project is visibly late is usually the moment it has entered its most sequential, communication-heavy phase.

Abdel-Hamid and Madnick also made a point about why managers kept falling into the trap. The mental model most project managers relied on, they wrote, was a simple single loop. Resources produce work. Work is reported. Reports are compared with the schedule. If the forecast completion date is later than the scheduled date, resources are adjusted. This loop is reasonable, simple and manageable. It is also missing the link that makes the trouble: the effect of the workforce level on productivity. A manager who carries only the simple loop in their head will add people to a late project with complete confidence, because in their model of the world, more resources always mean more work done.

Hold on to that observation. We will come back to it repeatedly, because the mental model of the manager who proposes twenty agents in the meeting is, in most cases, exactly this simple loop with the cost of resources set to nearly zero.

Three costs, revisited

The heart of Brooks's analysis is that adding a worker to a project with interdependent tasks imposes three costs: training the newcomer, the extra communication needed to keep everyone coordinated, and the effort of repartitioning the work so the newcomer has something to do. Before we can ask what agents change, we need to be precise about what each of these costs is made of.

Training is the cost of transferring context. A new engineer joining a project needs to know what the system is for, how it is structured, what conventions the team follows, which parts are fragile, which decisions have been made and why, and which are still open. Some of this can be written down. Much of it, on any real project, lives in the heads of the people already there, in old discussions and design reviews and the accumulated scar tissue of past incidents. Transferring it means those people stop doing their own work for a while and explain. The newcomer's questions are also a hidden form of review: they force the existing team to articulate assumptions they had stopped noticing, which is valuable, but which costs time.

Communication is the cost of keeping a shared understanding current as the work evolves. It is not a one-time cost. Every decision that affects more than one piece of the system needs to reach everyone whose piece it affects. Every interface needs to be agreed, and every change to an interface needs to be propagated. As the number of workers grows, the number of potential interfaces grows with it, and the chance that some decision fails to reach someone who needed it grows too. Brooks's pairwise formula is a rough upper bound on this cost. Good architecture reduces it by limiting which pieces need to know about which. But it never reaches zero, and on a project that is already late, the architecture is usually not in a state to absorb a sudden expansion in the number of independent workers.

Repartitioning is the cost of redrawing the boundaries of the work. A task that was divided among three people has to be redivided among five or ten. Each new boundary is an interface that must be defined, documented and agreed. Each new piece is a unit that must be specified well enough for someone to complete it without constant reference to the whole. Repartitioning a late project is particularly expensive because the boundaries that were drawn at the start have usually been eroded by the time the project is in trouble. Code that was meant to be separate has become entangled. Assumptions that were meant to be local have leaked. Drawing clean new boundaries through that kind of system is design work of the hardest kind.

Each of these costs is, at bottom, about understanding. Training is the cost of creating understanding in a new mind. Communication is the cost of keeping understanding synchronised across minds. Repartitioning is the cost of restructuring the work so that each mind needs to understand less. When we ask whether AI agents escape Brooks's Law, we are really asking whether they escape these costs of understanding. That question turns out to be subtler than it first appears, and the next three chapters are devoted to it.

The conditions Brooks named

It is easy to forget that Brooks did not claim adding people always fails. He described the conditions under which it does. The work must have interdependencies that require communication. The project must be late, which usually means the work is far enough along that its structure has set. And the added workers must need training and coordination to be useful.

Each of these conditions can be weakened. Work with fewer interdependencies tolerates more workers. Projects that add people early, before the structure hardens, suffer less. Workers who need less training and coordination cost less to add. Much of good software practice over the past fifty years can be read as a long effort to weaken these conditions: modular architectures that reduce interdependencies, early staffing plans that put people in place before the crunch, documentation and onboarding that cut training time, automated tests that reduce the communication needed to know whether a change broke something.

The case for AI agents as a solution to lateness is, implicitly, a claim that agents weaken the third condition so dramatically that the law no longer bites. An agent, the argument goes, needs no training because it has read more code than any human ever will. It needs no coordination because it can be given a precise task and left alone. It costs almost nothing, so even if each agent is only marginally useful, a large number of them must add up to something.

Each part of that argument contains some truth and some confusion. An agent has broad general knowledge, but it starts every task without the specific context of your system: the decisions your team made, the reasons behind them, the parts that are fragile, the conventions that are not written down. That context has to be supplied, either by humans writing it down or by the agent rediscovering it, and both cost time. An agent can be given a precise task, but precise tasks are the product of design work, and the design work is the part of the project that is late. An agent is cheap, but its output is not free to integrate. Every change it produces must be reviewed by someone who understands the design, merged without breaking anything else, and tested against the whole. If that someone is a scarce senior engineer, the cheap agent has simply moved the expensive bottleneck.

The man-month in a new costume

There is a version of the man-month fallacy that is specific to the age of agents, and it is worth naming before we go further. Call it the agent-hour. An agent-hour is a unit of compute spent on a task, and it is even more seductive than the man-month because it seems even more fungible. Agents do not have personalities, career ambitions, or bad days. One agent-hour seems exactly like another. Two agents should do twice the work of one, ten should do ten times the work, and the only limit is the budget.

The agent-hour fails for the same reason the man-month failed, which is that it measures effort rather than progress. The question that matters in a late project is not how much effort is being spent, but whether the work that remains is getting smaller. That depends on whether the effort is aimed at the right things, whether its results fit together, and whether the people responsible for the whole can absorb and verify what is being produced. None of those is measured by the number of agents running.

There is also a version of the training cost that is specific to agents, and it is more insidious than the human version because it is easy to miss. A human engineer, once trained, stays trained. They carry the context of the

project from one day to the next, and they get better at the work as the weeks go by. Most agents, as currently deployed, do not carry context in the same way. Each session begins with whatever context it is given: the task description, the files it reads, any instructions or memory the team has set up. Context that is not supplied is not known. The training cost is therefore not paid once per worker but, to some degree, once per task. Teams that take this seriously invest heavily in written context for their agents, and that investment is real and valuable. But it is design and documentation work, done by the humans who understand the system, and it competes for their time with every other demand a late project makes.

Brooks's three costs, in other words, do not disappear when the worker is an agent. They change form. Training becomes context provision. Communication becomes review and integration. Repartitioning becomes task specification. All three remain costs of understanding, borne largely by the people who already understand the system best. And all three grow as you add more workers.

Why the law survived

Most management maxims from the 1970s have aged badly. Brooks's Law has not, and it is worth asking why. Part of the answer is that Brooks was describing something structural rather than cultural. He was not saying that programmers of his era were bad at communicating, or that IBM's management was uniquely inept. He was saying that work with interdependencies requires coordination, that coordination costs grow faster than team size, and that adding coordination load to a project that is already behind tends to push it further behind. These are statements about the shape of the work, and the shape of software work has not changed nearly as much as its tools.

Another part of the answer is that Brooks wrote with unusual honesty about his own failures. The Mythical Man-Month is not a book of advice from someone who got it right. It is a book of lessons from someone who got many things wrong on one of the most consequential software projects

of his time, and who spent a decade working out why. That honesty gives the book a credibility that confident prescriptions lack. When Brooks says that adding people to OS/360 did not help, he is reporting what he saw.

The rest of this book takes Brooks's analysis as its starting point and asks what it means when the workers being added are not people. To answer that question properly, we need a better way to see the dynamics he described. Brooks gave us a law and a worked example. Abdel-Hamid and Madnick gave us a model. Donella Meadows gives us a language for reading any such model, and for asking where in it we might intervene. That is the subject of the next chapter.

CHAPTER 2

The Loop Behind the Law

Donella Meadows liked to begin with a bathtub. Water flows in from the tap and out through the drain. The level of water in the tub at any moment is the result of everything that has flowed in and out up to that moment. If the inflow is greater than the outflow, the level rises. If the outflow is greater, it falls. If they are equal, the level stays where it is, even though water is moving through the tub the whole time.

This is almost too simple to state, and Meadows knew it. Her point was that most of the behaviour of complex systems, including the behaviour that baffles the people inside them, comes from structures not much more complicated than the bathtub, linked together in ways that are hard to see from any single vantage point. In Thinking in Systems she described a stock as a store, a quantity of material or information that has built up over time, and she gave it a definition that every project manager should memorise: a stock is a physical memory of the history of the system.

A software project is full of stocks. The backlog of unfinished features is one. The body of working, tested code is another. The set of known bugs is a third. Less visibly, there is the stock of undiscovered defects, the errors that have been made but not yet found; the stock of design decisions that have been taken but not communicated to everyone who needs them; the stock of shared understanding in the team's heads; and, for a project that has begun using AI agents at scale, the stock of code that has been written but not yet reviewed or verified. Each of these stocks rises and falls according to its flows. Each of them carries the memory of the project's history. And each of them, crucially, changes slowly relative to the decisions that managers make about them.

This chapter redraws Brooks's Law in Meadows's terms. The aim is not to dress an old proverb in new vocabulary, but to make visible the structure that produces the behaviour Brooks observed, so that we can see which parts of the structure agents change and which they leave alone.

The stock everyone watches

Start with the stock every project manager watches: remaining work. It is usually represented by a backlog, a list of tickets, or a burndown chart. Its inflow is new work: features added to scope, bugs discovered, tasks that turn out to be bigger than estimated. Its outflow is completed work: tickets closed, features shipped. When the outflow exceeds the inflow, the remaining work shrinks and the project approaches completion.

The simple mental model that Abdel-Hamid and Madnick found among project managers treats this as a bathtub with one controllable drain. The rate at which work is completed depends on the number of people working, multiplied by how productive each one is. If the stock is falling too slowly to reach zero by the deadline, the manager opens the drain wider by adding people. This is a balancing loop, in Meadows's vocabulary: a structure that tries to bring a stock toward a goal. The goal is zero remaining work by the deadline. The gap between where the project is and where it should be drives a corrective action. The corrective action closes the gap.

Balancing loops are the workhorses of system design. A thermostat is a balancing loop. So is the reflex that keeps you upright when you stumble. They are also, as Meadows emphasised, only as good as the information they act on and the speed with which they can act. A thermostat with a broken sensor, or one that takes an hour to respond, will not keep a room at a steady temperature. It will oscillate, overshooting and undershooting, or it will drift away from the goal entirely.

The balancing loop in the project manager's head has both problems. Its information about remaining work is wrong, because the stock it can see is not the stock that matters. And its corrective action has side effects that the manager's model leaves out, which feed back to make the gap bigger rather than smaller.

The stock nobody sees

Here is the first problem. The burndown chart shows work the team knows about. It does not show work the team has done badly, which will have to be done again when the errors are discovered. That work sits in a separate, invisible stock: undiscovered rework.

Every time a developer writes code, some fraction of it contains errors. Some of those errors are caught immediately, in the developer's own testing or in review. Others slip through and sit in the codebase, looking like completed work. The ticket is closed. The burndown chart drops. But the stock of undiscovered rework has grown, and at some point in the future, during integration or system testing or in production, those errors will surface and flow back into the visible backlog as new work.

System dynamics practitioners who model large projects have described this structure, often called the rework cycle, since the 1980s, and it explains a pattern that every experienced engineer recognises. A project appears to make steady progress for a long time. The burndown chart descends on schedule. Then, as integration begins and the pieces are put together for the first time, a flood of problems appears. Remaining work, which had been falling, suddenly rises. The project that was ninety percent done is now, by any honest measure, perhaps sixty percent done, and the last forty percent turns out to take as long as the first sixty.

The delay between making an error and discovering it is the key to this pattern. Meadows wrote that delays in feedback loops are critical determinants of system behaviour, and that they are common causes of oscillation. If you are trying to adjust a stock toward a goal, but you only receive delayed information about the true state of the stock, you will overshoot and undershoot. A project manager watching a burndown chart

is receiving delayed information about the true state of the project, because the chart does not include undiscovered rework. They will believe the project is further along than it is, right up until the moment the rework surfaces. By then, the decisions that could have corrected course cheaply, such as reducing scope, extending the schedule, or fixing the design, have become expensive.

This is why Brooks's warning that projects slip one day at a time is so precise. The slippage is not invisible because people are lying or incompetent. It is invisible because the stock that is accumulating it is, by its nature, not shown on the charts people use to manage the project.

The loop that makes it worse

Now the second problem. When the manager acts on the gap by adding workers, the action has effects that their simple model does not include.

Adding workers increases the training load on the existing team, which reduces their productivity. It increases the number of communication paths, which reduces everyone's productivity further. It requires repartitioning the work, which takes time from the people best able to do it. All of this reduces the rate at which work is completed, at least for a while. If the reduction is large enough, the gap between the forecast and the deadline grows rather than shrinks. The manager, seeing the gap grow, adds more workers. The loop runs again.

This is a reinforcing loop. Where a balancing loop pushes a stock toward a goal, a reinforcing loop amplifies whatever change is already happening. More lateness leads to more workers leads to more overhead leads to more lateness. Brooks's regenerative schedule disaster is a reinforcing loop, and his law is a description of what happens when this loop dominates the balancing loop that the manager thinks they are operating.

There is a second reinforcing loop that runs alongside the first, and it is equally important. When a project is late, the people on it are under pressure. Pressure leads to shortcuts: less time for design, thinner testing, quicker reviews, more tolerance for code that works but is not quite right.

Shortcuts increase the error rate. A higher error rate fills the stock of undiscovered rework faster. When that rework surfaces, the project becomes later, the pressure increases, and the shortcuts multiply. This loop is quieter than the hiring loop, because it does not involve any visible decision. Nobody announces that the team will now write worse code. The quality simply erodes, a little at a time, under the weight of the schedule.

These two loops interact. New workers, still coming up to speed, make more errors than experienced ones. The existing team, distracted by training and coordination, has less time to catch them. Reviews get thinner as the volume of code to review grows. The stock of undiscovered rework fills from two directions at once, and the delay before it surfaces means the manager sees none of this until integration begins.

Meadows had a name for the general pattern in which a system's corrective action makes the original problem worse while seeming, in the short term, to help. She described a family of what she called system traps, structures that reliably produce problematic behaviour, and one of them fits the late project almost perfectly. In the trap she called shifting the burden to the intervenor, a system develops a problem, and instead of addressing its underlying cause, someone applies an intervention that relieves the symptom. The symptom eases. The underlying problem continues or worsens. The system becomes dependent on the intervention, and its own capacity to solve the problem atrophies.

Adding workers to a late project is a symptomatic intervention. The symptom is the gap between the forecast and the deadline. The underlying problem is usually something else: a design that does not fit the requirements, requirements that have not been settled, an estimate that was never realistic, an integration strategy that postponed all the hard problems to the end. Adding workers does not touch any of these. It applies force to the symptom, and the underlying problem continues. If anything, the extra workers make the underlying problem harder to address, because there are now more people building on the flawed foundation, and changing it would disrupt more work.

Agents in the diagram

Now redraw the diagram with agents in it. What changes?

The most obvious change is in the cost of adding a worker. In the human version, adding a worker takes weeks: recruiting, onboarding, getting access, learning the codebase. This delay is one of the few things that protects a late project from the worst of the hiring loop. By the time new people arrive, the manager may have recognised the problem and changed course. With agents, the delay shrinks to minutes. A manager can add twenty workers before lunch. The hiring loop, which used to turn slowly, can now spin as fast as someone can type a command.

This is not obviously good news. Meadows warned that speeding up a feedback loop can destabilise a system if the loop's information is still delayed. Her example was financial markets: the great push to reduce information and money transfer delays, she wrote, was just asking for wild gyrations. If the manager's information about the true state of the project is still delayed by the invisible stock of undiscovered rework, then making it faster and cheaper to add workers means the manager can make more decisions based on wrong information, more quickly, before the truth surfaces.

The second change is in what the new workers do to the stocks. Human workers added to a late project mostly increase the training and coordination burden on the existing team. Agents increase a different burden. They produce code quickly, and that code flows into a stock that did not used to be large enough to matter: code that has been written but not yet reviewed, verified and integrated. In a team of humans, this stock is small because writing and reviewing proceed at roughly comparable speeds. The person who wrote a change can usually explain it, and the person reviewing it can usually keep up. When agents write code at many times human speed, the stock of unreviewed code can grow faster than the humans responsible for the design can drain it.

What happens then? One of two things. Either the unreviewed stock grows without limit, and the project's visible progress stalls while a mountain of pending changes accumulates. Or the reviews get thinner, so

that more changes pass through without the scrutiny they need, and the stock of undiscovered rework grows instead. In practice, teams usually do some of each. The first is at least honest: everyone can see the pile of unreviewed changes. The second is the dangerous one, because it converts a visible stock into an invisible one, and invisible stocks are where late projects go to die.

This is the mechanism behind the DORA finding mentioned in the introduction. Teams that relied more heavily on AI reported better code quality and faster reviews, yet their delivery stability fell. The report's authors hypothesised that faster code generation had led to larger changes, and that larger changes were slower and more prone to instability. In stock-and-flow terms, the inflow to the unreviewed stock increased, the review outflow could not keep up, and the excess either piled up as larger batches or slipped through as undiscovered defects. Neither shows up as a problem on the day the code is written. Both show up later.

Delays upon delays

There is a third change, subtler than the first two. It concerns the delay between doing something and knowing whether it worked.

In a healthy software project, much of the feedback is fast. A developer writes a function, runs the tests, and knows within seconds whether it does what they expected. They push a change, and within minutes the continuous integration system tells them whether it broke anything. These fast loops are what make modern software development possible. They catch most errors close to the point where they were made, before they can compound.

But some feedback is slow by nature. Whether a design will scale to production load is not known until it meets production load, or until someone builds a realistic test of it. Whether a set of features actually solves the user's problem is not known until users try them. Whether two components built by different people will work together is not known until they are integrated. These slow loops carry the most important

information about a project, and they are the ones that a late project tends to defer. Integration gets pushed to the end. Load testing gets cut. User feedback is postponed until launch.

Agents are very good at closing fast loops. An agent can write code, run the tests, read the failures and fix them, all in a tight cycle that runs much faster than a human could manage. This is real value. But it can create an illusion. A project where agents are closing thousands of fast loops a day looks extraordinarily productive. Tests are passing. Tickets are closing. The burndown chart is falling. And the slow loops, the ones that carry information about whether the whole thing will actually work, are no shorter than they were. If anything, the speed of the fast loops may make it easier to defer the slow ones, because there is always another test to pass and another ticket to close, and the feeling of progress is intoxicating.

Meadows wrote that a system cannot respond to short-term changes when it has long-term delays. A late project with agents is a system in which the short-term loops have been made very fast and the long-term loops have stayed slow. The result is a project that responds energetically to every small signal and remains blind to the large ones.

The goal of the system

Meadows ranked the places to intervene in a system in order of increasing effectiveness, from changing numbers and parameters at the bottom to changing goals and paradigms at the top. We will return to her list in detail in the final part of this book, but one of her points belongs here, because it explains why the late project keeps choosing the intervention that does not work.

Adding workers is an intervention at the level of parameters. It changes a number, the size of the team, without changing the structure of the system, its rules, its information flows or its goals. Meadows put parameters at the bottom of her list. She did not say they never matter. She said that people pay far more attention to them than they deserve, because

parameters are easy to see, easy to argue about and easy to change, while the structural features that actually determine a system's behaviour are harder to see and harder to touch.

What would a higher-leverage intervention look like in a late project? It might change the information flows, so that the stock of undiscovered rework becomes visible earlier, through continuous integration of the whole system, realistic testing environments, or a walking skeleton that forces the pieces to fit together from the start. It might change the rules, so that no change can be merged until it passes tests that the change itself cannot modify, or so that the batch size of changes is kept small regardless of how fast they are produced. It might change the goal, from delivering a fixed scope by a fixed date to delivering the most valuable working subset by that date. Or it might change the paradigm, from thinking of software as a quantity of output to be produced to thinking of it as a coherent design to be realised.

Each of these is harder than adding workers. Each requires someone to admit that the current structure is part of the problem. And each, if done well, does more good than any number of extra hands, human or machine.

Drawing your own loop

The diagrams in this chapter are generic, and every project's version will differ in its details. It is worth drawing your own, and the exercise takes less time than most planning meetings.

Start with the stocks. On a whiteboard or a sheet of paper, write down the accumulations that matter for your project: the backlog of remaining work, the body of integrated and tested code, the queue of unreviewed changes, the list of known defects, and the stock you cannot see directly, undiscovered rework. If your team uses agents heavily, add the stock of agent output awaiting human judgment. If your design has open questions, add those too, as a stock of unresolved decisions.

Then draw the flows. What fills each stock, and what drains it? New scope and discovered defects fill the backlog; completed, verified work drains it. Agent output fills the review queue; human review drains it.

Errors that slip through review fill the undiscovered rework; integration, testing and production incidents drain it, by turning hidden problems into visible ones.

Then mark the decisions that people actually make about these flows, and trace what each decision does to the stocks over the following weeks. When the team adds an agent, which flows change first? When a deadline is brought forward, what happens to the time spent on review? When integration is deferred, how long does the undiscovered rework stock keep filling before anyone sees it?

Finally, mark the delays. How long, typically, between a change being written and its defects being found? Between a component being built and its first integration? Between a feature being finished and a real user trying it? The longest delays are usually where the project's real risk lives.

Teams that do this exercise often find that it shifts the conversation. The question stops being how to go faster and becomes which stock is filling without anyone watching it. That second question is the one that leads to the interventions described in the rest of this book.

Seeing the structure

The most useful thing Meadows offers engineering leaders is not any particular insight about software but a habit of seeing. She taught that the behaviour of a system arises from its structure, and that if you want to change the behaviour, you have to understand the structure. When a project is late, the natural instinct is to look for a cause: a person who underperformed, a requirement that changed, a technology that disappointed. Meadows would ask instead what structure produced the lateness. What stocks are accumulating? What flows are filling and draining them? What loops are driving those flows? Where are the delays that hide the true state of things from the people making decisions?

Asked this way, the late project usually reveals a familiar shape. A visible stock of remaining work that is shrinking more slowly than planned. An invisible stock of undiscovered rework that is growing faster than anyone knows. A balancing loop that the manager believes they are

operating, and two reinforcing loops, one from staffing and one from schedule pressure, that are actually running the show. Delays between action and consequence that make all of this hard to see until it is too late to fix cheaply.

Agents change some of the numbers in this structure. They make adding workers faster and cheaper. They make some kinds of work go much faster. They introduce a new stock, unreviewed code, that can fill much faster than it used to. But they do not change the shape. The reinforcing loops are still there. The invisible stock is still there. The delays are still there. And because agents make it easier and faster to pull the lever that feeds the reinforcing loops, a manager who reaches for them without understanding the structure can produce a regenerative disaster faster than Brooks ever imagined.

The next chapter looks more closely at what agents genuinely change, because the honest answer is not nothing. Some of the costs Brooks identified really do shrink when the worker is a machine. The question is which ones, by how much, and what new costs appear in their place.

CHAPTER 3

Cheap Hands

A fair argument has to begin by granting what is true in the other side's case, and there is a great deal that is true in the case for AI agents.

Consider what an engineer in 2026 can delegate. A coding agent can read an unfamiliar repository and produce a reasonable summary of its structure in the time it takes to make coffee. It can write the tedious parts of a feature, the data classes and serialisers and validation logic, from a short description. It can migrate a codebase from one version of a framework to another, following rules that would take a human days to apply by hand. It can write tests for code that has none, generate documentation for functions nobody remembers writing, and track down the call sites of a deprecated API across hundreds of files. It can run the test suite, read the failures, and try again without being asked. Teams that use these tools well report that whole categories of drudgery have disappeared from their days.

The measured capability of these systems has also moved fast. Anthropic, describing how it evaluates coding agents in early 2026, noted that on SWE-bench Verified, a benchmark built from real issues in popular open-source Python repositories and graded by running each project's test suite, model performance had climbed from about 40 percent to more than 80 percent in a single year. Benchmarks are not projects, and passing a project's tests on a well-specified issue is not the same as building a system. But the trend is not imaginary. The hands are real, they are getting better, and they are cheap.

So the question for this chapter is not whether agents can do useful work. They plainly can. The question is which of the costs Brooks identified they reduce, which they merely move, and which new costs they introduce. Getting this accounting right is the difference between using agents to accelerate a healthy project and using them to accelerate the collapse of a sick one.

The costs that shrink

Begin with the costs that genuinely shrink.

The first is the cost of producing text that represents an already-settled idea. Once a design decision has been made, someone has to express it in code: in a particular language, against a particular framework, following particular conventions. For a human, this expression takes time proportional to the amount of code, and much of that time is spent on mechanics rather than thought. An agent can do much of this expression very quickly. If the decision is clear and the conventions are documented, the agent's output is often close to what a competent human would have written, produced in a fraction of the time.

The second is the cost of recall. Programmers spend a surprising fraction of their working lives looking things up: the signature of a function, the right way to configure a library, the error message that means a certain kind of misconfiguration. Agents have absorbed an enormous amount of this knowledge, and they can often supply it instantly. This removes a constant low-level friction from the work.

The third is the cost of exploring unfamiliar territory. An engineer who needs to change code in a part of the system they have never seen, or in a language they rarely use, faces a steep learning curve. Agents can flatten that curve considerably, by explaining the code, suggesting where to make a change, and writing a first draft in an idiom the engineer would have had to learn. The METR researchers whose study found experienced developers slowed down by AI tools were careful to point out that their

setting, mature projects that the developers knew intimately, was exactly the setting where this benefit is smallest. They suggested that small new projects, or work in unfamiliar codebases, might see substantial speedup.

The fourth is the cost of mechanical verification. Running tests, reading their output, making a small fix and running them again is a loop that agents can execute tirelessly. For problems where the tests fully capture what correct behaviour means, this loop can converge on a working solution with little human involvement.

Notice what these four have in common. Each is a cost that arises after the essential thinking has been done. Expression assumes the idea is settled. Recall assumes you know what you are looking for. Exploration assumes someone will judge whether the agent's explanation is right. Mechanical verification assumes the tests encode the right definition of correct. In each case, the agent accelerates the work downstream of a decision. It does not make the decision.

Amdahl's shadow

There is a principle from computer architecture that captures the consequence precisely. Gene Amdahl, one of the principal architects of the same IBM System/360 family whose operating system Brooks managed, published a short paper in 1967 arguing that the speedup you can get from parallel processing is limited by the fraction of the work that must be done serially. If ten percent of a computation cannot be parallelised, then no matter how many processors you throw at the other ninety percent, the whole computation can never run more than ten times faster. The serial fraction sets a ceiling.

Amdahl was writing about processors, but the principle applies to any process that mixes parallelisable and serial work, and a software project is such a process. Brooks's own rule of thumb, which gave a sixth of the schedule to coding, a third to planning and half to testing and integration, suggests how much of a project is, in this sense, not obviously parallelisable. If agents made coding instantaneous and did nothing else, the project

would finish at most a sixth sooner. That is a real gain, and on a large project it might be worth a great deal. It is not the transformation the meeting with the red band is hoping for.

Of course, agents do more than code. They help with testing, with documentation, with some kinds of debugging. The parallelisable fraction is larger than a sixth. But the serial core, the part that requires one coherent understanding of the system to be formed, maintained and applied, is still there. Planning a design that hangs together, understanding why two components that each pass their tests fail when combined, deciding what the system should actually do when the requirements are ambiguous: these are the activities that set the ceiling, and they are the activities that late projects are short of.

Amdahl's law has a corollary that is directly relevant to the agent-hour fallacy. As you add processors to the parallelisable part, each additional one contributes less, because the serial part increasingly dominates. The first few agents on a project may speed things up noticeably. The twentieth will contribute almost nothing to the schedule, because the bottleneck is elsewhere. And if those twenty agents generate work for the serial part of the process, by producing code that must be reviewed and integrated by the same few people who hold the design, they can slow the whole thing down.

The costs that move

Now the costs that do not shrink but move.

Recall Brooks's three costs: training, communication and repartitioning. We saw in the first chapter that each is, at bottom, a cost of understanding. When the worker is an agent, each takes a new form.

Training becomes context provision. A human engineer is trained once and carries the training forward. An agent, in most current deployments, begins each task with only the context it is given or can find. If your system has conventions that are not written down, the agent does not know them. If a particular module is fragile and should be touched only with care, the agent does not know that either, unless someone has told it.

If the team decided six months ago, after a painful incident, never to call a certain service synchronously, the agent will cheerfully call it synchronously, because nothing in its context says not to. Teams that use agents well spend real effort writing down this kind of knowledge in forms that agents can consume: instruction files, architecture notes, examples of good changes, explicit lists of things never to do. That effort pays off. But it is effort, and it falls on the people who hold the knowledge, who are usually the most senior and most overloaded people on the project.

Communication becomes review. When two human engineers work on related parts of a system, they talk. They keep each other informed, often informally, about decisions that affect each other. With agents, this communication is replaced by something more one-directional. The agent produces a change, and a human must read it, understand what it does, judge whether it fits the design, and decide whether to accept it. The human cannot ask the agent the kind of probing, context-laden question they would ask a colleague, at least not with the same confidence that the answer reflects any stable understanding. Every change must be judged on its own evidence. And the volume of changes can be much larger than any human team would produce.

Repartitioning becomes specification. To hand work to an agent effectively, someone has to break it into pieces that the agent can complete with the context it has, and describe each piece clearly enough that the agent's output will fit with everything else. This is the same work Brooks called repartitioning, done at finer grain and greater volume. Writing a good specification for an agent is a skill, and it is a design skill: you cannot specify a piece of work well without understanding how it fits into the whole. The more agents you run, the more specifications someone has to write, and the more boundaries someone has to define.

In each case, the cost has moved from the worker to the people who understand the system. The agent itself is cheap. The understanding it needs, and the understanding needed to judge its output, is exactly as expensive as it ever was, and it is concentrated in a small number of heads.

The bottleneck has a queue

Eliyahu Goldratt's 1984 business novel The Goal popularised a principle that operations researchers had known for a long time: the throughput of any process is set by its bottleneck, the step with the least capacity relative to the demand on it. Improving any other step does not increase throughput. It just piles up more work in front of the bottleneck.

In a project where agents write much of the code, the bottleneck is usually human review and integration. Agents can produce changes far faster than people can read them carefully. If the review step is the constraint, then adding agents does not increase throughput. It increases the queue of changes waiting for review.

Queues have a property that is easy to state and easy to forget. Little's law, a basic result in queueing theory, says that the average number of items in a stable queue equals the rate at which they arrive multiplied by the average time each one spends waiting. If agents double the rate at which changes arrive for review, and review capacity stays the same, then either the queue doubles in length, or the time each change spends waiting doubles, or something gives. In practice, what gives is usually the depth of review. The reviewer, faced with a growing queue and a deadline, reads each change less carefully. Approvals become faster. The queue shrinks. And the undiscovered defects that careful review would have caught flow into the codebase instead.

This is not a hypothetical concern. It is the mechanism the DORA researchers suspected when they found that heavier AI adoption was associated with lower delivery stability despite apparent improvements in code quality and review speed. Faster reviews are not better reviews if they are faster because they are shallower. The improvement in the process measure masks a degradation in the outcome.

Some teams that work extensively with agents have recognised this and redesigned their review process around it. An account published on Anthropic's developer blog in November 2026, describing how the Claude Code team thinks about verification, set out an approach built on the premise that nobody can read every change. The team used test failures as

one way to decide what needed a person: the agent ran the checks and tried to fix anything that failed, and if a check was still failing after three rounds, it stopped and brought in a human. The same account described a deliberate choice not to stack additional reviewers on top of a primary one, and a rule that a change could never modify the checks that judged it, because if it could, the check was broken. Whether or not these particular practices suit a given team, they illustrate the point. When workers are cheap, the design of the review and verification process becomes the central engineering problem, because that is where the bottleneck lives.

Correlated errors

There is a new cost that has no direct analogue in Brooks's world, and it deserves careful attention because it undermines one of the oldest defences in software engineering.

When several different humans review a piece of work, their errors tend to be somewhat independent. One reviewer misses a race condition but catches an off-by-one error. Another misses the off-by-one but catches the race. Each person brings a different background, different habits of attention, different blind spots. The value of multiple reviewers comes precisely from this independence: the chance that all of them miss the same problem is lower than the chance that any one of them misses it.

When several instances of the same AI model work on a project, their errors are not independent in the same way. They share a training process, a set of tendencies and a set of blind spots. If the model tends to make a certain kind of mistake, it will tend to make it everywhere. If it tends to miss a certain kind of problem in review, every instance will tend to miss it. Running more instances does not diversify the error profile. It replicates it.

Weick and Sutcliffe made a related point about high reliability organisations. They valued diversity of experience not as a social good but as an instrument of perception. The more varied the people looking at a situation, the more of the situation they could collectively notice. A complex, unstable environment requires a correspondingly varied set of

observers if its surprises are to be caught early. An organisation that staffs its quality checks with copies of the same observer, however capable, has narrowed its perception in exactly the place it most needs breadth.

None of this means that agent review is worthless. A capable model catches many real problems, and it never gets bored or tired. But it means that agent review and human review are not interchangeable, and that adding more agent reviewers is not the same as adding more independent checks. A team that relies on agents to review agent-written code has a single point of failure in its verification process, distributed across many instances.

The confidence problem

A related cost concerns how agent output presents itself. A junior human engineer who is unsure about something usually shows it. They ask questions, hedge their explanations, flag areas where they are not confident. An experienced reviewer learns to read these signals and to focus attention where uncertainty is highest.

Agent output often lacks these signals. A change that is subtly wrong may be presented with the same fluency and apparent confidence as one that is exactly right. The explanation accompanying it may be plausible and well written whether or not it accurately describes what the code does. This puts more burden on the reviewer, who cannot rely on the author's own signals of doubt to direct their attention. They must evaluate every change on its merits, which is slower and more tiring, and which makes the shallow-review failure mode more likely.

The METR finding about the gap between perceived and measured speed is relevant here. The developers in that study believed AI had made them faster when it had, on average, made them slower. One plausible contributor is that agent output feels like progress. Code appears quickly, explanations are fluent, tests pass. The cost, which shows up as time spent checking, correcting and integrating, is spread out and less salient. If

individual developers working on their own repositories misjudge the net effect, a manager looking at aggregate dashboards from a team running many agents has even less to go on.

A revised accounting

Put all of this together and the accounting looks like this.

Agents shrink the costs of expression, recall, exploration and mechanical verification. These are real savings, and on work that is mostly made of these activities, the savings can be large.

Agents move the costs of training, communication and repartitioning from the worker to the people who understand the system. These costs do not shrink. They concentrate. And because the concentration falls on the scarcest people on the project, its effect on the schedule can be larger than the original cost.

Agents introduce new costs: queues in front of the review bottleneck, correlated errors that defeat the logic of multiple checks, and output whose confidence does not track its correctness.

Whether the net effect of adding agents to a project is positive depends on the balance among these, and the balance depends on the state of the project. On a healthy project, with a clear design, good documentation, solid tests, small batches and review capacity to spare, agents can add a great deal. The costs they move land on people who have room to absorb them, and the costs they introduce are caught by processes designed to catch them. On a late project, almost everything is the opposite. The design is under strain, the documentation lags the code, the tests are incomplete, the batches are large, and the people who understand the system are already overloaded. Agents added to such a project will mostly add to the load on the bottleneck.

This is Brooks's Law in its agent form. It is not that agents are useless. It is that their usefulness depends on precisely the things that late projects lack. The next chapter looks at the distinction that explains why: the difference between the essence of a software problem and its accidents.

CHAPTER 4

Essence and Accident

In 1986, eleven years after The Mythical Man-Month first appeared, Brooks presented a paper at a conference of the International Federation for Information Processing. It grew out of his experience chairing a Defense Science Board study on military software, and it was reprinted the following year in IEEE Computer, which gave it wide circulation. Its title was No Silver Bullet: Essence and Accidents of Software Engineering, and its central claim was deliberately provocative. There is no single development, Brooks wrote, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

The paper generated more debate than the original book. Brooks later remarked that it had stimulated more spirited discussion in the literature than The Mythical Man-Month had, and when he reissued the book for its twentieth anniversary in 1995, he included No Silver Bullet as a chapter and added another responding to its critics. His prediction, he noted with some satisfaction, seemed safe.

The prediction is the least interesting part of the paper. What makes No Silver Bullet worth reading forty years on is the distinction it draws, because that distinction is the key to understanding what any new tool, including an AI agent, can and cannot do for a software project.

Two kinds of difficulty

Brooks borrowed his terms from Aristotle. The essence of a thing is what makes it what it is. Its accidents are properties it happens to have but could lack without ceasing to be itself. Applied to software, the distinction separates two kinds of difficulty.

In the paper's abstract, Brooks put it this way: all software construction involves essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity, and accidental tasks, the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints.

The essential difficulty is in the conceptual structure. A software system is, at bottom, an interlocking set of concepts: data, the relationships among data items, algorithms, the invocation of functions. The hard part is specifying, designing and testing that structure, getting the concepts right and making sure they fit together. Brooks argued that this would remain hard no matter what tools were invented, because the difficulty was inherent in the problem.

The accidental difficulty is in the representation. Once you know what the concepts are, you still have to express them in a programming language, compile them, run them on hardware with limited memory and speed, and debug the expression when it does not match the intent. These difficulties are real, but they are not inherent in the problem. They arise from the tools available at the time. Better tools can reduce them.

Brooks's argument was that the great productivity gains of software history had come from attacking accidental difficulty. High-level languages freed programmers from much of the accidental complexity of machine code. Time-sharing shortened the feedback loop between writing a program and seeing it run. Unified programming environments made it easier to combine tools. Each of these was a large improvement, and each attacked accidental rather than essential difficulty.

The problem, Brooks argued, was that accidental difficulty was running out. If most of the remaining difficulty in building software was essential rather than accidental, then even eliminating all the accidental difficulty would not yield an order-of-magnitude improvement. You cannot get ten times faster by eliminating something that is less than nine tenths of the work.

The four properties

Brooks identified four properties of software that make its essence hard, and each of them bears directly on the question of what agents can do.

The first is complexity. Software systems have more distinct parts than almost any other human construct, and those parts interact in nonlinear ways. Scaling up a software system is not a matter of repeating the same elements in larger sizes. It requires an increase in the number of different elements, and the interactions among them grow faster than their number. This complexity, Brooks argued, is an essential property, not an accidental one. Descriptions of a software entity that abstract away its complexity often abstract away its essence.

The second is conformity. Much of the complexity a software engineer must master is arbitrary, imposed by the human institutions and systems the software has to work with. A physicist can believe that nature is governed by unifying principles. A software engineer building a payroll system must conform to the tax code, the company's existing databases, the union contracts and the quirks of the bank's file format, none of which were designed with each other in mind. No amount of cleverness can simplify this complexity away, because it comes from outside.

The third is changeability. Software is constantly subject to pressure to change. It embodies the function of a system, and function is the part of a system that people most want to adjust. Software is also seen as easy to change, which makes the pressure greater. A building or a car is rarely redesigned after it is built. Software is redesigned continuously.

The fourth is invisibility. Software has no natural geometric representation. A building has floor plans; a machine has blueprints; a circuit has a schematic. Software can be represented by many overlapping diagrams, of control flow, data flow, dependencies, time sequences, but none of them captures the whole, and the diagrams are hard to keep in sync with each other. This invisibility makes software hard to think about and hard to communicate about.

Each of these properties survives the arrival of AI agents, and some of them are intensified by it.

Complexity survives because agents do not reduce the number of concepts in a system or the interactions among them. If anything, cheap code generation increases complexity, because it lowers the cost of adding features, special cases and alternative paths. A system that would have stayed small because each addition was expensive can grow large when each addition is cheap.

Conformity survives because the arbitrary external constraints still exist. An agent can read the tax code faster than a person, but it cannot make the tax code simpler. It cannot reconcile two legacy systems whose data models contradict each other. It can only implement whatever reconciliation a person decides on, and deciding on it is the hard part.

Changeability survives and may intensify. If software was already seen as easy to change, software produced by agents is seen as easier still. The pressure to adjust function, to add one more feature, to handle one more case, grows when the perceived cost of each change falls. But the real cost of change in a complex system is not the cost of writing the new code. It is the cost of understanding what the change will affect, which grows with the size and entanglement of the system.

Invisibility survives because agents do not make software visible. They can generate diagrams, summaries and explanations, and these can help. But the explanations are themselves representations, partial and possibly wrong, and they need to be checked against the code. The fundamental problem, that no single representation captures the whole system, remains.

What agents attack

Given this framework, it is not hard to place AI coding agents. They are the most powerful attack on accidental difficulty since high-level languages. They reduce the cost of representing concepts in code. They reduce the cost of mapping between languages, frameworks and versions. They reduce the cost of recall, of looking up how to express a particular idea in a particular toolchain. They reduce the cost of the mechanical parts of testing and debugging.

These are large reductions, and they matter. But they are reductions in accidental difficulty, and Brooks's argument applies to them as it applied to every previous advance. If the essential difficulty of a project is a large fraction of the whole, then even eliminating its accidental difficulty entirely will produce a limited improvement. And in the cases that matter most for this book, late projects, the essential difficulty is usually a very large fraction of the whole, because that is typically why the project is late.

Think about why projects become late. It is rarely because the code took longer to type than expected. It is because the requirements turned out to be different from what everyone thought, or because two components that were designed separately turned out not to fit, or because a performance assumption proved wrong, or because the system had to conform to an external constraint that nobody had fully understood. These are all problems of essence. They concern the conceptual structure of the system, and they can only be solved by someone understanding that structure and changing it.

An agent added to such a project can help with the accidental work that follows once the essential problem is solved. It cannot solve the essential problem. And if it is set to work before the essential problem is solved, it will produce code that expresses the wrong concepts, quickly and fluently, adding to the pile that will have to be revised when the right concepts are found.

The hardest part

Brooks was blunt about where the hardest essential difficulty lay. The hardest single part of building a software system, he wrote, is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines and to other software systems. And no other part of the work so cripples the resulting system if done wrong, or is so difficult to rectify later.

He followed this with an observation that has lost none of its force: the client does not know what they want. They usually do not know what questions must be answered, and they have almost never thought of the problem in the detail that must be specified. Even a simple answer, such as making the new system work like the old one, turns out to be far too simple, because nobody fully knows how the old system works.

This is why late projects are so often late for reasons that have nothing to do with the speed of coding. They are late because the team discovered, partway through, that what they were building was not quite what was needed. The requirements were incomplete, or wrong, or changed. The interfaces to other systems turned out to be more complicated than anyone had realised. The users, seeing an early version, asked for something different.

A rocket that remembered the wrong flight

The maiden flight of the European Space Agency's Ariane 5 launcher, on 4 June 1996, illustrates how essential difficulty survives even the most aggressive attack on accidental difficulty.

About forty seconds after lift-off, the rocket veered off its flight path, broke up and was destroyed. The inquiry board, chaired by the French mathematician Jacques-Louis Lions, traced the failure to the inertial reference system, which supplied attitude and velocity data to the flight computer. Software in that system attempted to convert a 64-bit floating-point number, related to the horizontal velocity of the rocket, into a 16-bit signed integer. The value was larger than a 16-bit integer could hold. The

conversion raised an exception that was not handled, and the inertial reference system shut down. Its backup, running identical software, had failed in the same way a fraction of a second earlier. The flight computer, receiving diagnostic data that it interpreted as flight data, commanded extreme nozzle deflections, and the vehicle broke apart.

The software that failed had been reused from Ariane 4, where it had worked reliably. Reuse is one of the classic attacks on accidental difficulty: why write new code when proven code exists? And in a narrow sense, the reuse worked perfectly. The code did exactly what it had always done. The problem was that the conditions had changed. Ariane 5 had a different trajectory from Ariane 4, with considerably higher horizontal velocity early in flight. Under Ariane 4's flight profile, the value being converted could never grow large enough to overflow. Under Ariane 5's, it did. The inquiry found that the specific function in which the overflow occurred served no purpose after lift-off on Ariane 5 at all; it had been retained from the earlier design. And the board noted that Ariane 5's trajectory data had not been included in the requirements and testing for the reused system.

The reused code was an accident, in Brooks's sense: a representation of an idea, carried over cheaply from a previous project. The essence was the set of assumptions under which that representation was correct, including the assumption about how fast the rocket would be moving sideways. Those assumptions were not written into the code in any form that would fail visibly when they were violated. They lived in the heads of the original designers and in the flight profile of a different rocket. Reuse carried the accident across without its essence.

AI agents perform something analogous to reuse at enormous scale. They produce code shaped by patterns they have learned from vast amounts of existing software. Much of the time, those patterns are appropriate. But the patterns carry assumptions about the context in which the code will run, and those assumptions are not always visible. An agent writing code for a new system may produce something that would be perfectly correct in the systems it learned from, and subtly wrong in this one, because a condition that held there does not hold here. The code

passes its tests, if the tests do not exercise the violated assumption. It looks correct to a reviewer who does not know what assumption to look for. And it waits, like the Ariane 4 conversion, for the day when the conditions finally diverge.

Where agents can touch the essence

It would be wrong to conclude that agents are useless for essential difficulty. Brooks himself, at the end of No Silver Bullet, proposed several approaches that he thought could attack the essence, and some of them map surprisingly well onto what agents do well.

He suggested buying rather than building where possible, since the cheapest way to build software is not to build it. He suggested using rapid prototyping to refine requirements iteratively, because clients cannot specify what they want in the abstract but can react to something concrete. He suggested growing software incrementally rather than building it all at once, so that a working system exists from early on and functions are added one at a time. And he suggested developing great designers, because the difference between great and average designers is large, and great designs come from great designers.

The second of these is where agents may make their most important contribution to essential difficulty. If the hardest part of software is deciding what to build, and if clients can only decide by reacting to something concrete, then the speed at which you can produce something concrete for them to react to is a direct attack on the hardest part. Agents can produce prototypes very quickly. A team that uses them to generate several rough versions of an interface, show them to users, learn what is wrong, and generate several more, can explore the space of requirements faster than a team writing every prototype by hand.

But notice what this use requires. The prototypes must be treated as disposable instruments for learning, not as the foundation of the real system. The team must have the discipline to throw them away once they have served their purpose. And someone must be responsible for distilling what was learned into a coherent design, which is essential work that no

agent does on its own. Used this way, agents are tools for discovering the essence faster. Used carelessly, they turn the prototypes into the product, and the product inherits all the shortcuts and unexamined assumptions of the prototypes.

The third of Brooks's proposals, incremental growth, also connects to agents, and we will return to it at length when we discuss the walking skeleton. In brief, agents work best when there is a working system for them to extend, with clear boundaries and good tests. A project that grows incrementally provides exactly that. A project that tries to build everything at once and integrate at the end provides the opposite.

The fourth, great designers, is where agents change the picture most subtly. If agents make implementation cheap, then the relative importance of design rises. The bottleneck shifts from the people who write the code to the people who decide what the code should do and how it should be structured. In a team of humans, a mediocre design could be partly rescued by skilled implementers who noticed problems and worked around them. In a team where agents do much of the implementation, the design is expressed more directly, flaws and all. Agents will implement a bad design as fluently as a good one.

Accidental complexity can grow back

There is one more consequence of Brooks's distinction that matters for the age of agents, and it is counterintuitive. Agents reduce accidental difficulty in the act of writing code. But they can increase accidental complexity in the code that results.

Accidental complexity in a codebase is complexity that does not correspond to anything in the problem: duplicated logic, inconsistent conventions, unnecessary layers of indirection, five different ways of doing the same thing. Every codebase accumulates some, and keeping it under control is part of the ongoing work of software engineering. When code is expensive to write, there is a natural brake on this accumulation, because each unnecessary addition costs someone time. When code is cheap to write, the brake weakens. An agent asked to solve a problem will often

produce a new solution rather than finding and reusing an existing one, especially if the existing one is not in its context. Several agents working in parallel may each introduce their own helpers, their own patterns, their own way of handling errors.

The result can be a codebase in which accidental complexity grows faster than it would have under human authorship, even as the accidental difficulty of writing each piece falls. And accidental complexity in the codebase becomes accidental difficulty for everyone who has to work with it later: harder to understand, harder to change, harder to debug. The tool that removed one kind of accident creates another.

This is not inevitable. Teams can counter it with clear conventions, good documentation of existing patterns, and review that looks specifically for duplication and inconsistency. But all of these depend on someone holding a coherent picture of what the codebase should look like, which is the subject of Part Three of this book. Brooks called that picture conceptual integrity, and he thought it was the most important property a system could have.

What this means for the late project

Bring the argument back to the meeting with the red band. A project is late. Someone proposes adding agents. Brooks's distinction gives us a way to evaluate the proposal.

The first question is: why is the project late? If it is late because there is a large volume of well-understood work that simply needs to be done, with clear specifications, good tests and a stable design, then the lateness is largely accidental, and agents may help a great deal. This does happen. Migrations, porting, test coverage, documentation and certain kinds of feature work can fit this description.

But if the project is late because of essential difficulty, because the requirements are unsettled, the design does not fit, the components do not integrate, or external constraints are not understood, then agents will not help with the cause. They may be useful later, once the essential problem

has been dealt with. If they are set to work before then, they will produce code that embodies the wrong concepts, and the project will have more to revise when the right concepts are found.

In practice, most late projects are late for essential reasons, and the honest answer to the proposal is that agents are not the intervention the project needs. The intervention it needs is essential work: settling the requirements, fixing the design, integrating the pieces, understanding the constraints. That work is done by the people who understand the system, and the most useful thing a manager can do is protect their time and attention for it.

There is a final irony here. One of the reasons projects end up with unsettled requirements and unfit designs is that the warnings came early and were explained away. The next part of this book turns to that problem.

PART TWO

How Lateness Hides

Projects slip one day at a time, and organisations are remarkably good at not noticing. Drawing on Weick and Sutcliffe's research into high reliability organisations, these chapters examine the stories, labels, distances and measures that let a late project stay hidden until correction is expensive.

CHAPTER 5

The Reassuring Story

How does a project get to be a year late? Brooks's answer, in an essay he called Hatching a Catastrophe, is one of the most often repeated lines in his book: one day at a time.

He meant it literally. When a project suffers a disaster, a fire or a strike or a key person quitting, everyone notices and the team rallies. Major calamities are easy to handle, Brooks observed, because they call forth a major response. What kills projects is the steady accumulation of small slips: a meeting cancelled because someone was ill, a machine down for a day, a component delivered a week late because its author underestimated it, a review postponed because the reviewer was busy. Each of these is trivial on its own. Each is easy to explain. And each moves the project a little further from its schedule, in a way that no single person feels responsible for.

Brooks's remedy was precise milestones, concrete and specific and measurable, defined so sharply that nobody could fool themselves about whether they had been met. He contrasted these with fuzzy milestones, such as coding being ninety percent complete, which can stay at ninety percent for a very long time. He also observed that the chronic schedule slippage of a project was a morale-killer, and that a team which never met its milestones lost the ability to treat them seriously.

But Brooks did not fully explain why intelligent people let the slips accumulate even when they can see them. For that, we need Karl Weick.

How organisations make sense

Weick spent much of his career studying what he called sensemaking: the process by which people in organisations figure out what is going on around them. His insight was that sensemaking is not passive observation. People do not simply perceive the world and then interpret it. They act, notice the results of their actions, construct a story that explains those results, and then act on the basis of the story. The story comes first, in a sense. Once people have a plausible account of what is happening, they tend to notice evidence that fits it and overlook evidence that does not.

This is usually a strength. Organisations could not function if every member had to reconstruct their understanding of the world from scratch every morning. Stories give people a shared framework for coordinating action. They let a team move quickly without constant discussion. A plan is a kind of story, and so is a schedule, and so is the belief that the project is basically on track.

But stories have a cost. Once a story is in place, it shapes what people see. Evidence that the story is wrong has to overcome the story's inertia before it can be noticed, let alone acted on. And the more people have invested in the story, the more evidence it takes.

In Managing the Unexpected, Weick and Sutcliffe described how this plays out in organisations that are trying to operate reliably. The unexpected, they wrote, usually gives off only weak signals of trouble at first. Those weak signals are easy to miss and easy to explain. An organisation that is not deliberately looking for them will tend to fold them into the existing story. The signal becomes routine, or a known issue, or a one-off. Each time this happens, the organisation loses an opportunity to catch a problem while it is still small.

High reliability organisations, in their account, are distinctive partly because they resist this tendency. They treat their own stories with suspicion. When they recognise an event as something they have seen before and understood, Weick and Sutcliffe wrote, that recognition is a

source of concern rather than comfort, because superficial similarities between the present and the past can mask deeper differences that could prove fatal.

Eighteen warnings

The launch of HealthCare.gov on 1 October 2013 is one of the most thoroughly documented software failures in history, and the documentation is unusually clear about the role of reassuring stories.

HealthCare.gov was the federal website through which consumers in many US states would buy health insurance under the Affordable Care Act. It was a large and complex system, which had to interact with insurers, with state systems, and with several federal agencies to verify applicants' identity, income and eligibility for subsidies. It launched at midnight and experienced substantial problems within hours. The website received about five times the number of expected users, but the Office of Inspector General at the Department of Health and Human Services later emphasised that the problems involved more than capacity. Software defects caused malfunctions, and fixing the site required substantial corrections to the code as well as more capacity.

The inspector general's case study of the project, published in 2016, is a careful document, and its account of the warnings is worth reading closely. Between July 2011 and July 2013, the Centers for Medicare and Medicaid Services, which ran the project, received 18 documented warnings about it. These were not vague worries. They contained substantial detail about the project's shortcomings and were formally submitted to CMS senior leadership or project managers. They came from staff at HHS and CMS as well as from outside entities.

What happened to them? The inspector general's answer has several parts. The warnings were not shared broadly, because leadership of the project was diffuse and communication was poor. As a result, no one person in CMS had a comprehensive view of the project's poor progress. The problems were complex, and the information about them became unwieldy and difficult to prioritise. Without a single comprehensive view,

CMS leadership and staff took little action in response to the warnings, remained overly optimistic about the launch, and developed few contingency plans. And as the project degraded further and its problems became better known, officials and staff became desensitised to bad news about progress.

The inspector general also noted that despite the warnings, CMS moved forward without serious discussion of delaying the launch.

Read that account through Weick's eyes and the mechanism is plain. There was a story: the site would launch on 1 October. That story had enormous momentum. The date was politically significant, publicly committed and tied to the timeline of a major law. Every warning had to fight that momentum to be heard. And because the warnings arrived one at a time, in different places, to different people, none of them was ever strong enough on its own to overcome the story. Each could be explained as a particular problem that particular people were working on. Collectively, they described a project that was not going to launch successfully. But nobody saw them collectively, and so the story survived until reality ended it.

The phrase desensitised to bad news deserves to be taken seriously as a technical description. It describes a feedback loop in which repeated exposure to warning signals that do not lead to disaster reduces the weight given to subsequent warnings. Each warning that passes without visible consequence makes the next one easier to dismiss. Eventually, the organisation has calibrated its sensitivity to bad news at a level where only catastrophe gets through.

The normalisation of small slips

The sociologist Diane Vaughan, in her study of the 1986 Challenger space shuttle disaster, gave a name to a related process: the normalisation of deviance. Vaughan found that the engineers and managers involved had not ignored evidence of problems with the shuttle's solid rocket booster joints. They had noticed anomalies on earlier flights, discussed them, and through a series of reasonable-seeming steps redefined them as acceptable.

Each time a flight showed damage and returned safely, the observed damage became part of the expected range, and the threshold for concern moved.

Vaughan's work is about a specific case, and it would be a mistake to stretch it to cover every organisational failure. But the pattern she described appears in software projects with striking regularity. A test that fails intermittently is investigated, found to be flaky, and marked as such. The next time it fails, nobody investigates, because it is known to be flaky. Eventually it fails for a real reason, and nobody notices, because everyone has learned that its failures do not mean anything. A build that takes a little longer each week is accepted, because each increment is small. A deploy that occasionally needs a manual fix becomes a deploy that always needs a manual fix, and then a deploy that only one person knows how to fix.

Each of these is a reassuring story in miniature. The flaky test is just flaky. The slow build is just the cost of a growing codebase. The manual fix is just how deploys work here. Each story is plausible and often partly true. Each story makes a signal easier to ignore. And each story, once established, moves the organisation's baseline, so that the next deviation is measured against a slightly worse normal.

New stories for new workers

The arrival of AI agents in software teams has created a new set of reassuring stories, and they are worth naming because they are still forming and can be resisted before they harden.

The first is that the agent is just hallucinating. When an agent produces something wrong, a function that does not exist, an explanation that does not match the code, a test that passes for the wrong reason, the natural reaction is to attribute it to a known limitation of the technology. Agents sometimes make things up; everyone knows that. The error is corrected and the work moves on. But the error was information. It revealed something about the context the agent was given, the specification it was working from, or the part of the system it was touching. A team that files every agent error under hallucination loses that information.

The second is that the model had a bad day. Agent performance varies from run to run. The same task, given twice, may produce good output once and poor output the next time. This variability is real, and it makes it tempting to dismiss poor output as noise. But a pattern of poor output on a particular kind of task, or in a particular part of the codebase, is a signal. It may mean the task is underspecified, the code is confusing, or the conventions in that area are inconsistent. Treating each failure as random variation prevents the pattern from being seen.

The third is that the review will catch it. When agents produce large volumes of code, there is a temptation to accept more of it on the theory that any problems will be caught later, in review, in testing, in staging. This story distributes responsibility so thinly that nobody holds it. The author, being an agent, does not take responsibility. The reviewer, facing a large queue, assumes that testing will catch what they miss. Testing assumes that review caught the design problems. Each stage relies on the others, and the problems pass through all of them.

The fourth, and the most dangerous for a late project, is that we just need more throughput. When a project is behind, and agents are available, every sign of slippage can be read as evidence that more agents are needed. The story is that the project is basically sound and simply needs more capacity. This story is reassuring because it does not require anyone to admit that the design is wrong, the requirements are unsettled, or the schedule was fantasy. It converts every warning into a request for resources, and resources are now cheap.

Status and action

Brooks had a practical observation about why bad news fails to travel upward in organisations, and it remains one of the most useful pieces of advice in his book. He noted that a first-line manager who reports a slip to their boss runs a risk: the boss may take action, intervening in ways that undermine the manager's authority or disrupt their plans. Knowing this,

managers are reluctant to report problems until they have a solution, or until the problem is too large to hide. The boss, in turn, sees only good news until suddenly they see a disaster.

Brooks's remedy was to separate two kinds of meeting. Status review meetings exist to share information. Problem-action meetings exist to decide what to do. If the boss can be trusted not to act on information shared in a status review, and to save action for the meetings convened for that purpose, then managers can report problems early without fear of having their work taken out of their hands. The boss must, Brooks wrote, deliberately refrain from acting on problems their managers can solve, and must never act on a problem while reviewing status.

This advice applies with special force in projects that use agents, because the agent era has created a new kind of status report that is easy to misread. Dashboards now show agent activity: tasks started, tasks completed, pull requests opened, tests passed. These numbers are large and they move fast. They look like status. But they measure effort, not progress, and they can stay green while the project goes red underneath. A manager who reads a dashboard full of completed agent tasks and concludes that the project is on track has accepted a reassuring story told by their own instruments.

Practitioners have a name for this pattern, which predates agents but describes them well: watermelon reporting. The project is green on the outside and red on the inside. Every indicator on the dashboard is fine, and the people closest to the work know that the system is in trouble. The gap between the two persists because the indicators measure things that are easy to count, and the trouble is in things that are hard to count: whether the components fit together, whether the design still makes sense, whether the requirements have been understood.

The sound of a story forming

How can a team tell when a reassuring story is forming? Weick and Sutcliffe offered some markers, and they translate readily into the language of software projects.

One marker is a label that ends inquiry. When a problem is described with a word that explains it away, such as flaky, known issue, edge case, or just the model, the label is doing work. It closes a question that might have been worth keeping open. A useful discipline is to notice these labels and ask what they are hiding. What exactly makes this test flaky? Which edge cases, and how many? What about the context made the model produce this?

A second marker is an explanation that requires the evidence to be wrong. When a measurement shows a problem and the first response is to question the measurement, that response is sometimes correct. Instruments do fail. But an organisation that habitually doubts the measurements that bring bad news, while accepting those that bring good news, is protecting its story at the expense of its information.

A third marker is the absence of contingency planning. The HealthCare.gov inspector general noted that CMS developed few contingency plans before the launch, and that this left it with few options when the site ran into trouble. A team that has no plan for what to do if the project slips is a team that does not really believe it might slip. Its story has become so strong that the alternative is not imaginable.

A fourth marker is desensitisation itself. When warnings that would once have caused alarm are met with a shrug, the organisation's sensitivity has drifted. This is hard to notice from inside, because the drift is gradual and everyone drifts together. An outsider, or a newcomer, may see it more clearly. Their surprise at what the team accepts as normal is information.

Agents as sensors, humans as judges

There is a more hopeful side to the relationship between agents and early warnings, and it deserves to be stated, because it points toward a good use of the technology.

Agents are tireless readers. They can scan logs, test results, review comments, incident reports and support tickets far faster than any human, and they can be asked to look for exactly the patterns that reassuring stories tend to hide: a test whose failure rate has crept upward over a

month, a module that keeps appearing in bug reports, review comments that repeat the same concern about the same area, a deployment step that has needed manual intervention several times. In principle, agents could serve as an organisation's early-warning sensors, noticing the weak signals that humans miss because there are too many of them to read.

But the HealthCare.gov case shows that noticing is not the bottleneck. The warnings there were noticed, written down and formally submitted. What failed was the assembly of those warnings into a single picture held by someone with the authority to act, and the willingness of that person to let the picture override the story. An agent that surfaces a hundred weak signals a week does not solve this problem. It may make it worse, by adding to the volume of information that has to be prioritised and making it easier to dismiss any single signal as one of many.

The useful design is one in which agents act as sensors and humans act as judges. Agents gather and summarise signals, and route them to a single place: the architect, or the small group responsible for the project's overall health. That person or group reads the summary regularly, looks for patterns, and decides what each pattern means. The agents widen what the organisation can see. The humans decide what it is seeing. Neither substitutes for the other, and the arrangement only works if the humans are willing to let what they see change their minds.

Hearing bad news cheaply

The point of noticing reassuring stories is not to eliminate them. That is impossible, and it would be paralysing if it were possible. Organisations need stories to function. The point is to hold them loosely enough that evidence can change them while change is still cheap.

In a late project, this means treating every slip as a potential signal about the structure of the project, not just as a local problem to be fixed. When a milestone is missed, the question is not only how to catch up but what the miss reveals about the plan. When an integration fails, the question is not only how to fix this integration but what the failure says

about the design. When agents produce poor output on a class of tasks, the question is not only how to get better output but what the pattern says about the specifications, the code or the context.

It also means building channels through which bad news can travel without being filtered. The HealthCare.gov warnings were formally submitted, but they went to different people in different parts of a diffuse organisation, and nobody assembled them. A project that has a single person, or a small group, responsible for seeing the whole picture is a project in which warnings can accumulate in one place and be weighed together. This is one of the reasons conceptual integrity matters, a theme we will develop in Part Three. The person who holds the design is also the natural place for warnings about the design to converge.

Finally, it means being suspicious of reassurance that comes too easily. When a project is late and the proposed fix is one that requires no one to admit anything was wrong, that is a reason to look harder. Adding agents to a late project is often exactly that kind of fix. It may be the right decision. But it should be made after the team has looked honestly at why the project is late, not instead of looking.

The next chapter turns to the first of the practices that Weick and Sutcliffe identified in organisations that hear bad news early: a preoccupation with small failures.

CHAPTER 6

Preoccupation with Small Failures

On 23 September 1999, after a journey of more than nine months, NASA's Mars Climate Orbiter fired its main engine to slow down and enter orbit around Mars. It passed behind the planet, as planned, and was never heard from again. The spacecraft had approached Mars far lower than intended. NASA's Mishap Investigation Board later estimated that its trajectory at the time of orbit insertion was about 170 kilometres lower than planned, low enough that it was either destroyed in the atmosphere or skipped back out into space.

The root cause, which the board announced in November 1999, became one of the most famous engineering errors of the century. A piece of ground software called SM_FORCES, which calculated the effect of small thruster firings on the spacecraft's trajectory, produced its output in English units, pound-force seconds, when the project's interface specification required metric units, newton-seconds. The navigation software that consumed this output assumed it was metric. The result was that the effect of every thruster firing was underestimated by a factor of about 4.45, the number of newtons in a pound of force. Over months of flight, the small errors accumulated into a fatal one.

The units error is usually told as a parable about the importance of standards, and it is that. But the board's report tells a richer story, and the richer story is the one that matters for this chapter. The units error was a

small failure. What turned it into a lost spacecraft was a series of moments in which other small failures, the signs that something was wrong, were noticed and then not pursued.

The signals that were there

The report's account of these moments is precise and worth following step by step.

The thruster firings in question were angular momentum desaturation events, small burns used to unload momentum that built up in the spacecraft's reaction wheels. On Mars Climate Orbiter, these events happened far more often than the operations navigation team expected: ten to fourteen times more often, according to the board. The reason was the spacecraft's asymmetrical solar array, which caught sunlight unevenly and caused momentum to build up faster than on its predecessor, Mars Global Surveyor, which had symmetrical arrays. More frequent desaturation events meant more opportunities for the units error to inject small mistakes into the trajectory estimate.

During the first four months of the cruise to Mars, the ground software files that recorded these events were not used in navigation at all, because they had multiple format errors and incorrect attitude data. Instead, the navigation team relied on email from the contractor to tell them when a desaturation event was happening, and modelled the effects themselves. It took four months to fix the file problems. Only in April 1999 could the team begin using correctly formatted files. And almost immediately, within a week according to the board, it became apparent that the files contained anomalous data suggesting the trajectory effects of the desaturation events were being underestimated.

So by April, five months before arrival, there was a signal. Throughout the spring and summer, the board found, concerns existed at the working level about discrepancies between navigation solutions. The residuals between the expected and observed Doppler signatures of the desaturation events were noted, but only informally reported.

The Jet Propulsion Laboratory had a formal process for exactly this kind of situation. It was called the Incident, Surprise, Anomaly process, and its purpose was to make sure that anything unexpected was written down, tracked and resolved. The navigation team did not use it. When conflicts in the data were uncovered, the board wrote, the team relied on email to solve problems instead of formal problem resolution processes such as the ISA procedure. Failing to use the problem tracking system contributed to the problem slipping through the cracks. Some members of the team, the board found, did not recognise the purpose and use of the ISA process at all.

The board's second report, on project management, was blunter. A critical deficiency in the project's management was the lack of discipline in reporting problems and insufficient follow-up. The ISA process was not embraced by the whole team. Project leadership did not instil the sense of authority and responsibility that would have spurred workers to broadcast the problems they detected, so that those problems could be articulated, interpreted and elevated until resolved.

The last chance

There was one more opportunity. As the spacecraft approached Mars, a contingency plan existed for a fifth trajectory correction manoeuvre, TCM-5, which could raise the spacecraft's altitude if its approach was too low. According to the board, a request to perform TCM-5 was discussed verbally shortly before the orbit insertion procedure began. It was never executed.

The board gave several reasons. The analysis, tests and procedures needed to commit to TCM-5 had not been completed or attempted, so the operations team was not prepared for the manoeuvre. The orbit insertion timeline on board the spacecraft took priority and did not leave time to upload, execute and verify a correction. And, in the board's words, the criticality of performing TCM-5 was not fully understood by the spacecraft operations or operations navigation personnel. NASA officials, speaking to

the press at the time the report was released, put it more simply: the navigation team had concerns, but they were not clear on what the problem was and did not see it as a case of possible loss of the spacecraft.

Read in sequence, these are not the actions of careless people. A small anomaly in the data was noticed. It was discussed. Concerns were raised. But they were raised informally, by email and in conversation, in a form that did not force anyone to resolve them. The anomaly was never escalated to the point where someone with authority had to decide what it meant. And when the moment came to act on it, the team had not prepared to act, because nobody had treated the anomaly as the kind of thing that might require action.

The first principle

Weick and Sutcliffe placed preoccupation with failure first among the five principles they observed in high reliability organisations, and the Mars Climate Orbiter story illustrates why it comes first.

High reliability organisations, they wrote, are distinctive because they are preoccupied with failure. They treat any lapse as a symptom that something may be wrong with the system, something that could have severe consequences if several separate small errors happened to coincide. They encourage the reporting of errors. They treat near misses as opportunities to learn rather than as evidence that the system works. And they are wary of the liabilities of success, including complacency, the temptation to reduce margins of safety, and the drift into automatic processing.

The phrase that matters most is the one about small errors coinciding. Most serious failures in complex systems are not caused by a single large mistake. They are caused by several small ones lining up. The units error on Mars Climate Orbiter was serious, but on its own it might have been caught. It became fatal because it coincided with the unexpectedly frequent desaturation events, the four months of unusable files, the informal handling of the anomaly, the absence of a prepared contingency, and a navigation team that, the board found, had been somewhat isolated from

the development and operations teams. Each of these was a small failure. Any one of them, noticed and treated as a symptom of something potentially serious, could have broken the chain.

A preoccupation with failure is not the same as pessimism or anxiety. It is a habit of attention: the practice of treating every small deviation from expectation as information about the system rather than as noise to be filtered out. Organisations that have this habit catch problems early, when they are small and cheap to fix. Organisations that lack it catch problems late, when they have grown and combined.

Why success breeds blindness

Weick and Sutcliffe warned that success is one of the main enemies of this habit, and their warning applies directly to teams that have begun using AI agents and are pleased with the results.

Success narrows perception. When things go well, people come to expect them to go well, and they stop looking for signs that they might not. Margins get trimmed, because they seem unnecessary. Checks get skipped, because they never find anything. Processing becomes automatic, because the situation seems familiar. Each of these changes is reasonable in isolation, and each makes the organisation less able to notice when something unfamiliar is happening.

A team that has adopted agents and seen them work well on a series of tasks is particularly exposed to this. The early successes are often striking. Code appears quickly, tests pass, features ship. It becomes natural to expect agent output to be correct, to review it less carefully, to give agents larger and more ambiguous tasks, and to integrate their work with less scrutiny. The team's expectations adjust to the successes, and the small failures that do occur start to look like exceptions to a reliable pattern rather than symptoms of something worth understanding.

The Mars Climate Orbiter project had its own version of this. The navigation team, the board noted, assumed that the spacecraft had Mars Global Surveyor heritage, and expected much of its hardware and software to be similar. That assumption, born of a previous success, led them to

acquire insufficient knowledge of the new spacecraft. They did not know until long after launch that it routinely calculated and transmitted velocity change data for the desaturation events. An early comparison of those spacecraft-generated data with the tracking data, the board suggested, might have revealed the units problem.

Previous success had become a reason not to look closely. That is exactly the dynamic Weick and Sutcliffe warned about, and it is exactly the dynamic that a team pleased with its agents should watch for in itself.

What counts as a small failure with agents

To practise preoccupation with failure in a team that uses agents, the team needs a shared sense of what counts as a small failure worth attention. Some of these are obvious, and some are easy to miss.

An agent that produces code which does not compile, or fails its tests, is producing an obvious failure. These usually get attention, because they block progress. But even here, there is a choice between fixing the immediate problem and asking what it reveals. If agents repeatedly fail on a particular kind of task, the pattern may say something about the specification, the code or the context, and fixing each failure individually loses the pattern.

An agent that produces code which passes its tests but does something slightly different from what was intended is producing a subtler failure. It may be caught in review, or it may not. When it is caught, it is easy to treat it as a one-off correction. But it is evidence that the tests do not fully capture the intended behaviour, and that the specification was ambiguous enough for the agent to read it differently from the person who wrote it. Both are worth knowing.

An agent that modifies something it was not asked to modify, such as a configuration file, a test, a dependency or a piece of code outside the scope of its task, is producing a failure that deserves particular attention. Even if the modification is harmless, it shows that the agent's actions are not

confined to the scope the team expected. In a system where agents can take actions with real consequences, that is information about the boundaries of the system's control.

An agent whose explanation of what it did does not match what it actually did is producing a failure that strikes at the heart of the review process. Reviewers rely on explanations to direct their attention. If the explanation and the change diverge, the reviewer's attention is misdirected. A single instance may be an accident. A pattern means the team cannot trust explanations, and must review accordingly.

A human reviewer who approves an agent's change and later realises they did not really understand it is producing a failure too, and it is perhaps the most important one to notice. It means the review process is passing changes that no human has fully understood. That is a signal about the load on the reviewers, the size of the changes, or both.

None of these is dramatic. Each can be fixed in a minute and forgotten. The discipline Weick and Sutcliffe describe is to resist that, and to treat each as a clue.

Building the reporting habit

The Mars Climate Orbiter board did not conclude that the navigation team should have been smarter. It concluded that the project lacked a discipline of reporting problems and following them up. JPL had the process; the project did not embrace it. The lesson for engineering leaders is that preoccupation with failure has to be built into the structure of the work, not left to individual vigilance.

In practice, this means making it easy and expected to record small failures. Many teams already have incident processes for production outages. Fewer have lightweight processes for the small surprises that happen during development. A team that uses agents heavily benefits from something like JPL's ISA process, scaled to the rhythm of daily work: a simple, low-friction way to note that something unexpected happened, what it was, and what it might mean. The record does not need to be

elaborate. It needs to exist, so that patterns can be seen across individual events, and so that a concern raised by one person reaches someone who can act on it.

It also means following up. A reporting process that collects observations and does nothing with them teaches people that reporting is pointless. Somebody needs to look at the record regularly, ask what patterns it shows, and decide what to do about them. In a project with a clear architect, a figure we will meet properly in Part Three, that person is the natural owner of this review, because the patterns in small failures are usually patterns in the design.

And it means treating escalation as normal rather than as an admission of failure. The navigation team on Mars Climate Orbiter had concerns but did not see them as a case of possible loss of the spacecraft. Part of the reason was that the project did not make it easy or expected to escalate concerns that were not yet clearly defined. A team that wants to catch problems early has to accept that most escalations will turn out to be nothing, and to reward people for raising them anyway.

Preparing to act

The second lesson from Mars Climate Orbiter concerns TCM-5. The manoeuvre that might have saved the spacecraft existed as a contingency plan, but the work needed to execute it had not been done. When the moment came, the team could not act, because it had not prepared to act.

A preoccupation with failure includes thinking in advance about what failures might occur and what would be done about them. This is the purpose of a technique that the psychologist Gary Klein described as the premortem: before a project begins, the team imagines that it has failed and works backward to identify the most likely causes. The exercise surfaces risks that optimism would otherwise hide, and it gives people permission to voice concerns that might seem disloyal in an ordinary planning meeting.

For a project using agents, a premortem might ask questions like these. If this project fails, what will the agents have done wrong that we did not catch? Where in our review process could a bad change slip through? What happens if an agent takes an action in production that we did not intend? What is our plan if the volume of agent output overwhelms our ability to review it? What would we do if we discovered, a month before launch, that a large body of agent-written code embodied a wrong assumption?

Each of these questions, answered honestly, suggests preparations: a rollback plan, a review capacity limit, a scope boundary on agent actions, a mechanism for tracing which code came from which task. These preparations are the equivalent of having TCM-5 ready. Most of them will never be used. The one that is used may save the project.

The cost of attention

There is an obvious objection to all of this. Preoccupation with failure costs time. Recording small surprises, reviewing the record, escalating ambiguous concerns and preparing contingencies are all work, and a late project does not have time to spare. Surely the right move is to focus on the critical path and deal with small problems later?

The answer from Weick and Sutcliffe, and from the Mars Climate Orbiter board, is that this gets the economics backward. Small failures are cheap to investigate when they first appear, because they are small and their causes are still close at hand. They become expensive when they have combined with other small failures and grown into a large one, because by then their causes are tangled together and far in the past. The time spent attending to small failures is an investment that pays off by preventing the large ones. A project that skips it to save time is borrowing against its future at a ruinous rate.

This is especially true of projects that use agents, because agents increase the rate at which small failures can occur and accumulate. A team that generates many times more changes than before, without proportionally more attention to the small failures among them, is filling the stock of

undiscovered problems faster than ever. The preoccupation that high reliability organisations practise is not a luxury that a fast team can skip. It is the thing that lets a fast team stay fast without crashing.

The next chapter turns to the second of Weick and Sutcliffe's principles, reluctance to simplify, and to the labels that organisations use to explain failures away.

CHAPTER 7

Reluctance to Simplify

Every organisation needs simplifications. Nobody can attend to everything, and a team that tried to keep every detail of a complex system in view at once would never make a decision. Weick and Sutcliffe were clear about this. Success in any coordinated activity, they wrote, requires that people simplify in order to stay focused on a handful of key issues and key indicators. Then they added the sentence that defines the second principle of high reliability organising: but it is also true that less simplification allows you to see more.

Reluctance to simplify is the discipline of holding off on the comfortable label until you are sure it fits. High reliability organisations, in Weick and Sutcliffe's account, take deliberate steps to create more complete and nuanced pictures of what they face. They welcome diverse experience, scepticism toward received wisdom, and ways of resolving disagreements that do not destroy the nuances different people detect. They know that the world they face is complex, unstable and hard to predict, and they position themselves to see as much of it as possible.

The opposite habit is familiar from every post-incident review that has ever gone wrong. Something bad happens. Someone proposes an explanation. The explanation is plausible, it is short, and it locates the problem somewhere that does not require the organisation to change much. It is accepted, the review closes, and the conditions that produced the problem remain in place, waiting to produce it again. The two most common labels in this genre are operator error and routine incident. In software, they have cousins: user error, a one-off, a flaky environment, and now, with agents, a bad prompt or a hallucination.

The case that best illustrates what these labels cost is one of the most studied software failures in history.

A machine that could not overdose

The Therac-25 was a medical linear accelerator built by Atomic Energy of Canada Limited and used to deliver radiation therapy to cancer patients. It could operate in two modes: a low-energy electron mode, and a high-energy X-ray mode in which a much more powerful electron beam struck a metal target to produce X-rays. In X-ray mode, the beam was strong enough to be lethal if it struck a patient directly without the target and other equipment in place. The machine's design depended on making sure that never happened.

Earlier machines in the same family, the Therac-6 and Therac-20, had used hardware interlocks, physical mechanisms that prevented the beam from firing when the machine was in an unsafe configuration. The Therac-25 relied much more heavily on software for these safety functions. Nancy Leveson and Clark Turner, whose 1993 investigation in IEEE Computer remains the definitive account, identified this as one of the central factors in what followed: the removal of hardware interlocks made the software a single point of failure that could lead to an accident.

Eleven Therac-25 machines were installed, five in the United States and six in Canada. Between June 1985 and January 1987, they were involved in six known accidents in which patients received massive radiation overdoses. Several people died, and others were seriously injured. Leveson and Turner noted that the accidents had been described as the worst series of radiation accidents in the then thirty-five-year history of medical accelerators.

What makes the Therac-25 story so instructive is not only the software defects, though those were real and serious. It is the way the early accidents were explained.

The explanations that closed inquiry

When patients reported burning sensations and showed injuries consistent with radiation overdose, the initial response was often disbelief. The machine had been designed so that an overdose of that kind could not occur. Its software checked the configuration before firing. Its error messages would alert operators to problems. The engineers who built it were confident in its design. The natural assumption, faced with an injury, was that something other than the machine was to blame.

Leveson and Turner documented how this played out. After one early accident, the manufacturer could not reproduce the problem, concluded that a microswitch failure was the most likely cause, modified the hardware, and treated the issue as resolved. Leveson and Turner identified this as one of the serious mistakes that led to multiple accidents: the tendency to believe that the cause of an accident had been determined without adequate evidence, and without looking at all possible contributing factors. They added a second mistake that follows naturally from the first: the assumption that patching one factor would eliminate future accidents. Their verdict on this assumption was four words long. There is always another software bug.

The machine also had a habit that trained its operators to ignore it. When it detected certain problems, it displayed short, cryptic error messages and paused treatment. Operators found that these messages appeared frequently and usually meant nothing serious, and that treatment could be resumed with a keystroke. Over time, the messages became noise. When one of them appeared during an actual overdose, it looked like all the others.

This is reluctance to simplify in reverse. Each simplification, that the machine could not overdose, that the cause was a microswitch, that the error messages were routine, closed a line of inquiry that might have revealed the truth. Each was plausible. Each was supported by some evidence. And together they allowed the machine to keep injuring people for more than a year and a half.

Operator error

The defects that Leveson and Turner eventually traced were subtle. One involved a race condition in the software that handled the operator's data entry. If an experienced operator entered treatment parameters, then quickly edited them, the software could end up in an inconsistent state in which the machine's configuration did not match what the display showed. The condition depended on timing: it occurred only if the edits were made fast enough, which meant that it was most likely to affect the most skilled and practised operators. A hospital physicist, working with an operator who had been involved in one of the accidents, eventually reproduced the problem by having her repeat her data entry at her usual speed. A second accident at another hospital involved a different software flaw.

Consider what the label operator error would have done here. The overdoses occurred when operators did something specific with the interface. It would have been easy to conclude that the operators had made mistakes, entering data incorrectly or too fast, and to respond with training or warnings. Some of the early responses did point in that direction. But the operators were doing exactly what the interface allowed and invited them to do. The error was in the design, which permitted a sequence of ordinary actions to produce a lethal configuration. Blaming the operators would have left the design intact.

Leveson and Turner were emphatic on this point. Accidents, they wrote, are often blamed on a single cause like human error. But virtually all factors involved in accidents can be labelled human error, except perhaps hardware wear-out failures, and even those could be attributed to a designer's failure to provide redundancy or a maintainer's failure to replace parts. Concluding that an accident was the result of human error, they wrote, is not very helpful or meaningful.

They extended the same scepticism to the opposite label. It was nearly as useless, they argued, to ascribe the cause of the accidents to a computer error or a software error. Software was involved, but it was only one contributing factor. If software error were the whole explanation, the only remedies would be perfect software, which is impossible, or no software,

which is unrealistic. The real contributing factors included management inadequacies and the lack of procedures for following through on reported incidents, overconfidence in the software, the removal of hardware interlocks, less than acceptable software engineering practices, and unrealistic risk assessments combined with overconfidence in their results.

That list is the opposite of a simplification. It refuses to locate the problem in any one place. It insists that the accidents emerged from the interaction of design choices, organisational practices, assumptions and incentives. And it points to remedies at every level, rather than a single patch.

The new labels

AI agents have introduced a fresh vocabulary of simplifying labels into software teams, and they deserve the same scepticism that Leveson and Turner applied to operator error.

The first is hallucination. When an agent produces something false, such as a function that does not exist, a library behaviour that is not real, or a description of code that does not match the code, the word hallucination provides an instant explanation. Agents hallucinate; this is known. The explanation is true in a narrow sense, and it closes inquiry. But the useful questions are still open. What in the context led the agent to produce this particular falsehood? Was the relevant information missing, ambiguous or contradicted by something else the agent saw? Would a human with the same context have made the same mistake? Is this kind of error concentrated in a particular part of the system or a particular kind of task? Each of these questions can lead to a change in the system, in the documentation, the specifications, the code structure or the review process, that would prevent a whole class of errors. The label prevents them from being asked.

The second is a bad prompt. When an agent does something unwanted, it is tempting to blame the instructions it was given. If only the prompt had been clearer, the agent would have done the right thing. Sometimes this is true. But it is the agent-era equivalent of operator error, and it has

the same weakness. It locates the problem in the individual interaction rather than the system. If the system allows a slightly unclear instruction to produce a harmful action, the system is the problem. People will always give slightly unclear instructions. The design should make that safe.

The third is model variability. Agent outputs vary from run to run, and it is easy to attribute any bad output to this variability and move on. But variability is not an explanation. It is a description of a symptom. If an agent fails on a task one time in five, the useful question is what distinguishes the failing runs from the succeeding ones, and whether the task, the context or the checks can be changed to make the failures less likely or more visible.

The fourth is that agents just do that. As teams accumulate experience with agents, they develop folk knowledge about their quirks: they tend to over-engineer, they sometimes modify tests to make them pass, they forget instructions given early in a long session. This folk knowledge is valuable. But when it hardens into a set of labels that explain every surprise, it becomes a filter that stops the team from noticing when something new is happening. The surprise that looks like a known quirk may be something else entirely. Weick and Sutcliffe's warning about recognition applies here with special force: the more familiar a failure seems, the more carefully it should be examined, because familiarity is what makes us stop looking.

Seeing more

What does reluctance to simplify look like in practice, for an engineering team running humans and agents together? Weick and Sutcliffe offered several practices, and each translates into concrete habits.

The first is to seek out diverse interpretations before settling on one. When something goes wrong, ask several people with different backgrounds what they think happened, before anyone proposes a cause. The engineer who wrote the code, the one who reviewed it, the one who operates the system in production and the one who talks to users will each see different things. If an agent was involved, look at what the agent was

given, what it did and what it reported, and compare all three. The goal is not to reach a consensus quickly but to build a picture that includes as many of the relevant details as possible.

The second is to look for disconfirming evidence. Once an explanation has been proposed, ask what evidence would show it to be wrong, and go looking for that evidence. If the explanation is that a test was flaky, check whether its failures correlate with anything: a particular time of day, a particular kind of change, a particular dependency. If the explanation is that the agent hallucinated, check whether the hallucination was consistent with something in its context. Explanations that survive a deliberate search for disconfirmation are more trustworthy than those that were simply never challenged.

The third is to be wary of explanations that locate the problem in a single place. Leveson and Turner's list of contributing factors in the Therac-25 accidents ran across design, engineering practice, management, risk assessment and organisational follow-through. Most serious failures in software projects have a similar shape. An explanation that says the problem was one bug, one person or one bad prompt is probably incomplete. A useful test is to ask: if we fix only this, how confident are we that nothing similar will happen again? If the honest answer is not very, the explanation has stopped too soon.

The fourth is to keep track of the simplifications the team is relying on, so they can be revisited. Every project rests on assumptions: that a certain component is reliable, that a certain kind of input never occurs, that agents will not touch a certain part of the system. These assumptions are necessary. But they should be written down, so that when evidence arrives that one of them might be false, the team knows what it has been taking for granted.

Simplification and scale

There is a particular reason why reluctance to simplify matters more when agents are involved, and it has to do with scale.

In a team of humans, the volume of work is limited by the number of people, and the number of distinct events that might reveal a problem is correspondingly limited. A team of eight engineers might produce a few dozen changes a day, and a few small surprises. It is possible, if tiring, to look at each surprise with some care.

A team that uses agents heavily may produce hundreds of changes a day, and a correspondingly larger number of small surprises. The pressure to simplify grows with the volume. Nobody can investigate hundreds of anomalies a day in depth. Labels become a necessity, a way of triaging the flood. And the more the team relies on labels, the more likely it is that a significant signal gets filed under a familiar heading and forgotten.

This does not mean that reluctance to simplify is impossible at scale. It means it has to be designed for. The team needs ways of grouping anomalies that preserve their detail rather than flattening it: logging what each agent was given and what it did, so that patterns can be found later; sampling a fraction of apparently routine failures for deeper investigation; tracking the frequency of each label over time, so that a rise in hallucinations or flaky tests is itself treated as a signal. The goal is to simplify in a way that keeps open the possibility of seeing more.

Reports that invite scrutiny

One of the quieter lessons of the Therac-25 concerns the machine's own reports. Its error messages were short and cryptic, and they appeared often enough that operators learned to treat them as noise. A report that cannot be interpreted, or that cries wolf too often, trains its readers to ignore it, and then it fails them on the day it matters.

Agents produce reports too. At the end of a task, most agents summarise what they did: the files they changed, the tests they ran, the problems they encountered, the reasons for their choices. These summaries are the primary way a human reviewer learns what happened, and they are therefore a design surface that deserves the same attention as an error message on a medical device.

A good agent report makes simplification harder, not easier. It distinguishes between what the agent verified and what it assumed. It says which tests it ran and which it did not. It flags anything it changed outside the scope of the task. It notes where it was uncertain, or where the specification was ambiguous and it had to choose an interpretation. It does not bury these details under a confident narrative of success.

A poor agent report does the opposite. It summarises everything as done. It describes intentions rather than results. It reports that tests pass without saying which tests, or whether they were modified. It reads smoothly and invites the reviewer to accept it, which is precisely the invitation a reviewer under time pressure is most inclined to accept.

Teams can shape these reports. The architect's playbook, which we will come to in the eleventh chapter, can specify what an agent's summary must include, and review tooling can check that the required sections are present. More importantly, teams can train themselves to read reports sceptically. A useful habit is to check one claim in every report against the evidence: open the test run, look at the diff, confirm that the files said to be untouched were untouched. Most of the time the claim will be accurate. The habit exists for the times it is not, and it keeps the reviewer's attention calibrated to the possibility.

A simplification audit

Reluctance to simplify can be practised as a periodic exercise rather than left to individual temperament. Once a month, or at the end of each significant phase of a project, a team can hold what might be called a simplification audit.

The audit has a simple structure. The team lists the labels it has used most often in the period to explain failures and surprises: flaky, known issue, environment problem, agent quirk, user error, and whatever else has become common currency. For each label, it counts roughly how many events were filed under it. Then it picks a small sample of events under each label, chosen at random rather than by memory, and looks at them again with fresh eyes.

The question for each sampled event is whether the label still fits. Was the flaky test really flaky, or did its failures cluster around a particular kind of change? Was the agent quirk really a quirk, or was the agent responding sensibly to an ambiguous instruction or a confusing piece of code? Was the user error really the user's fault, or did the interface invite the mistake?

In most audits, most labels hold up. That is fine. The value of the exercise is in the minority that do not, and in what the team learns about its own habits of explanation. A team that discovers that a third of its flaky tests were in fact detecting a real race condition, or that most of its agent quirks traced back to a single poorly documented module, has found exactly the kind of signal that Weick and Sutcliffe's second principle is meant to protect. It has also learned to be a little slower, next time, to reach for the label.

The machine and the system

The Therac-25 is often taught as a story about software bugs, and the bugs were real. But Leveson and Turner's deepest lesson was that the problem was never only the software. It was the system in which the software was embedded: the decision to rely on software for safety, the organisational responses to early accidents, the cryptic error messages that trained operators to ignore them, the confidence that prevented people from believing the evidence in front of them. Fixing the bugs without fixing the system would have left the conditions for the next accident in place.

The same is true of software projects that use agents. When something goes wrong, it is tempting to fix the immediate problem, the bad output, the wrong change, the failed test, and move on. Sometimes that is enough. But often the immediate problem is a symptom of something in the system: a gap in the specifications, a weakness in the review process, an assumption that no longer holds. A team that is reluctant to simplify will look for that something. A team that reaches for the nearest label will not, and will meet the same problem again in a different form.

The next chapter turns to the third of Weick and Sutcliffe's principles, sensitivity to operations, and to the places where the plans of a project meet the reality of the people who have to use what it builds.

CHAPTER 8

Sensitivity to the Sharp End

Safety researchers sometimes distinguish between the sharp end and the blunt end of a system. The sharp end is where the work meets the world: the pilot's hands on the controls, the nurse at the bedside, the operator at the console, the engineer on call when the pager goes off at three in the morning. The blunt end is where the conditions for that work are set: the boardroom, the budget office, the regulator, the design review, the planning meeting. Decisions made at the blunt end shape what happens at the sharp end, but the people making them are usually far from it. They see the sharp end through reports, metrics and summaries, all of which are simplifications.

Weick and Sutcliffe's third principle, sensitivity to operations, is about closing that distance. High reliability organisations, they wrote, are attentive to the front line, where the real work gets done. They pay close attention to minute-to-minute operations and to the imperfections in them. They strive to keep an integrated picture of what is actually happening, as opposed to what the plan says should be happening. And they understand that the gap between work as imagined and work as done is where most surprises are born.

Software projects have a sharp end too, and it is usually further from the people running the project than they realise. It is where the system meets its users, where code meets production load, where one component meets another for the first time. Late projects tend to push the sharp end further away, by postponing integration, deferring user testing and cutting operational preparation. Projects that use agents can push it further still,

because agents work entirely at the blunt end of software, in repositories and test environments, and have no direct contact with the people and conditions their code will eventually face.

This chapter looks at two cases in which the distance between the blunt end and the sharp end proved decisive, and then at what sensitivity to operations means for teams that include machine contributors.

Assumptions about the pilot

On 29 October 2018, Lion Air flight 610, a Boeing 737 MAX, crashed into the Java Sea shortly after taking off from Jakarta. On 10 March 2019, Ethiopian Airlines flight 302, another 737 MAX, crashed shortly after taking off from Addis Ababa. Everyone on board both aircraft, 346 people in all, died. Both accidents involved a flight control system called the Maneuvering Characteristics Augmentation System, MCAS, which had been added to the 737 MAX to make its handling in certain conditions resemble that of earlier 737 models. In both accidents, MCAS activated repeatedly on the basis of erroneous angle of attack data, pushing the aircraft's nose down while the pilots fought to keep it up.

The full causes of these accidents are complex, contested in places, and the subject of formal investigations by the Indonesian and Ethiopian authorities, as well as extensive inquiries in the United States. This chapter does not attempt to summarise them. It focuses on one strand that was examined in a specific report by the US National Transportation Safety Board, because that strand bears directly on the relationship between design assumptions and the sharp end.

In September 2019, the NTSB issued a safety recommendation report on the assumptions used in the safety assessment process for the 737 MAX, and on the effects of multiple alerts and indications on pilot performance. The report examined how Boeing and the Federal Aviation Administration had assessed the hazards of MCAS during the aircraft's design and certification. A key element of that assessment was an assumption about

how pilots would respond if MCAS activated when it should not. The assessment assumed that pilots would recognise the problem and take appropriate corrective action within a certain time.

The NTSB found that in both accidents, the crews did not react in the ways Boeing and the FAA had assumed they would. The board's chairman, Robert Sumwalt, described a gap between the assumptions used to certify the MAX and the real-world experience of these crews, who were faced with multiple alarms and alerts at the same time. The NTSB was careful to say that its report did not analyse the actions of the pilots involved, which was a matter for the accident investigations. Its concern was with the assumptions themselves. FAA guidance, it found, allowed such assumptions to be made in certification analyses without clear direction about the effect of multiple flight deck alerts and indications on pilot recognition and response. It recommended that the FAA require Boeing to make sure its safety assessments considered all the alerts and indications pilots might see, and that it develop more robust tools and methods for validating assumptions about pilot response.

Strip away the aviation specifics, and this is a story about the distance between the blunt end and the sharp end. At the blunt end, engineers and certifiers made an assumption about what would happen at the sharp end: that pilots would recognise a specific malfunction and respond to it in a specific way. The assumption was reasonable in isolation. But it was made without fully modelling the conditions the pilots would actually face, in which the malfunction arrived alongside a cascade of other alerts that competed for their attention. The assumption described work as imagined. The accidents revealed work as done.

The NTSB's remedy was not to tell pilots to respond better. It was to change the process by which assumptions about the sharp end were made, so that they would be tested against realistic conditions. That is sensitivity to operations institutionalised: a requirement that the blunt end check its picture of the sharp end against reality.

A programme that forgot the ward

The National Programme for IT in the NHS, launched in England in 2002, was one of the largest civilian IT programmes ever attempted. Its central aim was to give every patient an electronic care record that would be available to whoever needed it to care for them. It was planned as a ten-year programme of reform, procured centrally, and delivered through large contracts with a small number of suppliers who were to provide standardised systems to NHS organisations across regions of the country.

In 2011, the government decided to dismantle the programme. The National Audit Office, reporting that year on the delivery of detailed care records systems, found that the rate at which they were being put in place was falling far below expectations, that the core aim of an electronic care record for every patient would not be achieved, and that even where systems had been delivered, they were not yet able to do everything the Department of Health had intended, especially in acute hospital trusts. In acute trusts, the systems were mainly providing administrative benefits rather than the expected clinical ones. The NAO's later review of digital transformation in the NHS, published in 2019, estimated total spending on the programme at £9.8 billion.

It is important to be fair about the programme. The NAO noted that some parts of it were delivered successfully and were still in use years later, including the NHS Spine, which enables the sharing of electronic information between organisations, the Electronic Referral Service and the Electronic Prescription Service. The failure was concentrated in the detailed care records systems, which were the part of the programme that had to fit most closely into the daily work of clinicians.

When the NAO looked back in 2019, it cited the lessons identified by an independent review led by Robert Wachter in 2016. The review found that there had been no definitive analysis of the programme's failings, but it identified six main lessons. The first was that the programme lacked engagement with clinical staff and focused on technology rather than on service change and on the changes in the way people work that new technology requires. The second was that it used a controlled, top-down

approach to implement standardised systems, with insufficient support from local organisations and professionals. Others concerned the political pressure and rushed deployment, unrealistic expectations and scope that kept growing, inadequate central support for the trusts implementing the systems, problematic procurement with nearly impossible timeframes and unclear scope, and continuous changes of leadership.

The first two lessons are lessons about the sharp end. The people who would use the care records systems every day, doctors and nurses and clerks on hospital wards and in clinics, were not sufficiently engaged in shaping them. The systems were specified, procured and deployed from the centre, at the blunt end, according to a picture of clinical work that did not match the reality closely enough. When the systems arrived at the sharp end, they did not fit, and the work of making them fit fell on the local organisations that had had the least say in their design.

Where agents work

Both of these cases predate AI agents. But they describe a pattern that agents make easier to fall into, because agents, by their nature, work entirely at the blunt end.

An agent writing code sees the repository, the specification it was given, the tests it can run, and whatever documentation is in its context. It does not see the users who will interact with the code. It does not see the production environment, with its real data, real load, real failure modes and real dependencies. It does not see the on-call engineer who will be woken when the code fails, or the support team who will field the complaints, or the operations staff who will have to deploy and monitor it. Everything the agent knows about the sharp end comes to it secondhand, through the specification and the tests, which are themselves simplifications written by humans at the blunt end.

This is not a criticism of agents. Human developers also work at the blunt end much of the time. But human developers usually have some contact with the sharp end. They hear about incidents. They are on call. They talk to users, or at least to people who talk to users. They carry, often

without noticing, a sense of how their code will be used and what tends to go wrong. That sense shapes their work in ways that never appear in a specification. They add a log line here, a sanity check there, a more helpful error message, because they remember the last time something like this went wrong in production.

When agents do more of the writing, this tacit sense of the sharp end has to be supplied explicitly, or it is lost. A specification that says what a function should do, without saying how it will be used and what has gone wrong before, gives an agent no way to anticipate the sharp end. The agent will satisfy the specification. Whether the result works at the sharp end depends on whether the specification captured what matters there.

There is a further risk. A team that uses agents heavily may find that its human members spend more of their time at the blunt end too, specifying tasks, reviewing output, managing agent workflows, and less of their time in contact with production, users and operations. The people who used to carry the tacit sense of the sharp end may gradually lose it, because they are no longer close enough to the sharp end to keep it fresh. The whole team drifts toward the blunt end, and its picture of reality becomes increasingly a picture of its own plans.

Keeping the sharp end in view

What does sensitivity to operations look like for a team of humans and agents? Several practices follow from Weick and Sutcliffe's account and from the lessons of the cases above.

The first is to keep humans in regular contact with the sharp end, and to protect that contact from the pressure of a late project. Engineers who direct agents should still be on call, still read incident reports, still talk to users or the people who support them. This is not a distraction from their real work. It is what keeps their picture of the system honest, and it is what lets them write specifications and review output with the sharp end in mind. A project that pulls its engineers off on-call duty to spend more time managing agents has traded its eyes for its hands.

The second is to feed the sharp end into the agents' context. If production incidents, user complaints, operational runbooks and post-incident reviews are written down, they can be given to agents as part of the context for related work. An agent asked to modify a payment flow can be shown the last three incidents involving that flow. An agent writing an error handler can be shown the support tickets that the previous error handler generated. This does not give the agent the tacit sense that a human develops over years, but it gives it some of the raw material from which that sense is made.

The third is to test against realistic conditions, not just against specifications. The NTSB's core recommendation for the 737 MAX was that assumptions about the sharp end must be validated against realistic scenarios, including the effect of multiple simultaneous alerts. In software, the equivalent is testing that exercises the system the way it will actually be used: with realistic data, realistic load, realistic failure modes and realistic combinations of events. Unit tests, which agents are very good at writing and passing, test whether code does what its specification says. They do not test whether the specification matches the sharp end. That requires end-to-end tests, staging environments that resemble production, load tests, and above all, contact with real users as early as possible.

The fourth is to involve the people at the sharp end in shaping what is built. The first lesson of the Wachter review was that the NHS programme lacked engagement with clinical staff. Every software project has its clinical staff: the people who will use the system every day and whose work it is meant to support. Engaging them early and continuously is expensive, slow and politically awkward, especially on a late project. It is also the most reliable way to discover that the system being built is not the system that is needed, while there is still time to change it.

The fifth is to treat operational signals as first-class information about the project. Production metrics, error rates, support ticket volumes, deployment failures and on-call load are all signals from the sharp end. In a project that is building something new, there may not yet be a production system to measure. But there are almost always early signals available: from a pilot deployment, a beta programme, an internal dogfooding effort, or a

walking skeleton running in a realistic environment. A team that is sensitive to operations seeks these signals out, rather than waiting for launch to find out how the system behaves.

An illustration

Consider a hypothetical team, typical of many, that has been asked to rebuild a customer onboarding flow. The project is behind schedule. The team lead decides to accelerate by assigning agents to the bulk of the implementation, with the human engineers specifying tasks and reviewing the results. Velocity on the dashboard improves dramatically. Within two weeks, most of the planned features have been implemented and pass their tests.

Then the flow goes to a small group of real customers in a pilot. Several problems appear at once. Customers abandon the flow at a step that seemed simple in the specification but turns out to require information they do not have to hand. An integration with an identity verification service, which worked perfectly against the vendor's test environment, fails intermittently against the real service under load. The error messages, which faithfully report what went wrong in technical terms, leave customers confused about what to do next. Support tickets spike.

None of these problems was caused by the agents writing bad code. The code did what it was specified to do. The problems were all at the sharp end, in the gap between the specification and the reality of customers, services and load. They would have appeared whether the code was written by agents or by humans. But the speed at which agents implemented the specification meant that a large volume of code was built on top of the flawed picture before the picture was tested. Fixing the flow now requires changes across many components, some of which embody the original assumptions deeply.

A team more sensitive to operations might have done things in a different order. It might have put a skeleton of the flow in front of a handful of real customers in the first week, before most of the features were built. It might have tested the identity integration against the real

service early, under realistic load. It might have asked the support team what kinds of error messages generated the most tickets in the old flow. None of this would have required fewer agents or less code. It would have required that the sharp end be consulted before the blunt end committed to a picture of it.

Drift toward the blunt end

There is a slow danger in all of this that is easy to miss, because it happens gradually and without anyone deciding it should. As agents take on more implementation work, the humans on a team naturally shift toward specification, review and coordination. These are blunt-end activities. They are valuable, and they are where much of the leverage of a human-plus-agent team lies. But if the shift goes too far, the team loses its contact with the sharp end, and with it the tacit knowledge that made its specifications good and its reviews sharp.

Weick and Sutcliffe described sensitivity to operations as a continuous effort, not a one-time achievement. High reliability organisations maintain it by keeping people at all levels in touch with the front line, by encouraging the flow of information from the front line upward, and by treating the front line's picture of reality as authoritative when it conflicts with the plan. A team of humans and agents needs the same continuous effort, and it needs it more, because the agents themselves contribute nothing to it. Every bit of sensitivity to operations that the team has must come from its human members, and those members must deliberately protect the time and attention it requires.

For a late project, the pressure to abandon this effort is intense. Every hour an engineer spends on call, reading incident reports or talking to users is an hour not spent directing agents or reviewing their output. The dashboard does not reward it. But the dashboard measures blunt-end activity, and the project will succeed or fail at the sharp end. A team that neglects the sharp end to speed up the blunt end will discover, at launch, how far apart the two have drifted.

The next chapter takes up the question of how a team can measure its progress honestly, so that the gap between work as imagined and work as done shows up on the chart before it shows up in production.

CHAPTER 9

Ninety Percent Done

Every engineer who has worked on a large project knows the joke, and most have lived it. The first ninety percent of the code accounts for the first ninety percent of the development time. The remaining ten percent of the code accounts for the other ninety percent of the development time. The line is usually credited to Tom Cargill of Bell Labs, and it became widely known after Jon Bentley quoted it in his Programming Pearls column in 1985. It is funny because it is true, and it is true because of something specific about the way software progress is measured.

Brooks identified the same pattern a decade earlier. He observed that coding tends to be reported as ninety percent finished for about half of the total coding time, and that debugging tends to be ninety-nine percent complete most of the time. His diagnosis was that these milestones were fuzzy. A statement like coding is ninety percent done has no sharp edge. Nobody can check it. It reflects the reporter's feeling about the work, and that feeling is shaped by optimism, by the desire to please, and by the simple fact that the hardest part of a task is usually the part that has not yet been attempted. Brooks argued that milestones must be concrete, specific, measurable events, defined so clearly that there can be no self-deception about whether they have been met. A milestone such as the module has passed all its specified tests is hard to fudge. A milestone such as the module is nearly done is not.

This chapter is about measuring progress honestly, which turns out to be harder in a project that uses agents than in one that does not. The reason is that agents produce enormous amounts of activity, and activity is easy to mistake for progress. A dashboard full of completed tasks, merged

changes and passing tests can show a project racing toward completion while the actual distance to a working system stays the same or grows. The ninety-ninety rule does not go away when the code is written by machines. It gets harder to see.

Effort, output and progress

It helps to distinguish three things that are often conflated.

Effort is what goes into the work: hours, person-months, agent-hours, compute. It is the easiest of the three to measure and the least informative. A project can consume enormous effort while making no progress at all, and it can make large progress with modest effort if the effort is aimed well.

Output is what the work produces: lines of code, pull requests, tickets closed, tests written, features implemented. It is more informative than effort, because it at least measures something that exists. But output is not progress either. Code that does the wrong thing is output. Features that do not integrate are output. Tests that test the wrong behaviour are output. Output can grow while the distance to a working system stays constant, or even increases, if the output adds complexity that will later have to be removed.

Progress is the reduction in the distance between the current state of the system and a state in which it does what it needs to do, reliably, for the people who will use it. Progress is hard to measure because the destination is not fully known in advance, because the distance to it includes work that has not yet been discovered, and because much of the work that matters, integration, debugging, testing against reality, does not produce countable output until it is finished.

Agents increase effort and output dramatically. Whether they increase progress depends on whether the effort and output are aimed at the right things and fit together into something that works. A manager who watches effort or output and calls it progress will be misled, and the more agents are running, the more misleading the numbers become.

The cone of uncertainty

There is a further reason why progress is hard to measure early in a project, and it has nothing to do with agents. At the start of a project, nobody knows exactly how much work it will take, because the requirements are incompletely understood and the design has not been worked out. As the project proceeds, and as requirements are clarified and design decisions are made, the uncertainty narrows.

Barry Boehm documented this pattern in his 1981 book Software Engineering Economics, showing that estimates made early in a project could be off by a large factor in either direction, and that the range of plausible outcomes narrowed as the project progressed through its phases. Steve McConnell later gave the pattern a memorable name, the cone of uncertainty, and the name stuck. The cone is widest at the beginning and narrows as decisions are made and unknowns become known.

The cone has an important implication for measuring progress. Early in a project, a large fraction of the remaining work is unknown. Any measurement of progress that counts only known work will overstate how far the project has come, because it leaves out the work that has not yet been discovered. This is the invisible stock of the second chapter seen from a different angle. A burndown chart that shows ninety percent of known tasks complete may correspond to a project that is, in terms of total work including the undiscovered, much less than ninety percent done.

The cone does not narrow by itself. It narrows because decisions are made and unknowns are resolved. Writing code that embodies a decision helps narrow it. So does integrating components, testing against realistic conditions, and putting the system in front of users. Writing code that does not resolve any unknowns, that elaborates parts of the system whose design is already settled while the unsettled parts wait, does not narrow the cone at all. It increases output without reducing uncertainty.

This is where agents can create a dangerous illusion. Agents are best at work whose design is settled: clear specifications, well-understood patterns, good tests. They are least effective at the work that narrows the cone: resolving ambiguous requirements, making design trade-offs,

discovering integration problems. A project that assigns agents to the work they do best will produce a great deal of output in the parts of the system that were already well understood, while the parts that hold the real uncertainty wait for human attention. The burndown chart will fall rapidly. The cone will stay wide.

Fuzzy milestones in a fast project

Brooks's advice about sharp milestones is more important in an agent-assisted project, and it is also harder to follow, because agents generate many signals that look like sharp milestones and are not.

A pull request merged is a sharp event. But it says nothing about whether the merged code does what is needed, fits with the rest of the system, or will survive contact with production. A test passing is a sharp event. But it says only that the code satisfies the test, and if the test was written by the same agent that wrote the code, or by an agent working from the same specification, it may encode the same misunderstanding. A task marked complete is a sharp event. But it reflects the agent's or the reviewer's judgment that the task's stated criteria were met, which may or may not correspond to the task's purpose.

These events are useful for tracking activity. They are not good milestones, in Brooks's sense, because they do not measure progress toward a working system. They can all be satisfied while the system remains far from working.

What would good milestones look like? They would be defined in terms of the system's behaviour at the sharp end, not in terms of the activity at the blunt end. A user can complete the onboarding flow from start to finish in a production-like environment. The system handles the expected peak load with acceptable latency. The integration with the payment provider works against the real provider's sandbox. The three most common failure scenarios have been tested and recover correctly. Each of these is concrete, specific and measurable. Each is hard to fake. And each is a genuine measure of progress, because it can only be met by a system that actually works, at least for the scenario described.

Milestones of this kind have a useful property: they are naturally integrative. They cannot be met by work on any single component. They require the components to fit together. This makes them resistant to the illusion that agents can create, in which many components are individually complete but the whole does not work. A milestone that requires end-to-end behaviour forces the integration question to the surface early, and integration, as Brooks noted, is where most of the schedule goes.

When a measure becomes a target

The economist Charles Goodhart observed in the 1970s that any statistical regularity tends to collapse once pressure is placed on it for control purposes. The anthropologist Marilyn Strathern later restated the idea in a form that has become proverbial: when a measure becomes a target, it ceases to be a good measure. The principle, usually called Goodhart's law, applies to software metrics with particular force, and it applies to agents with more force still.

When human engineers are measured by a metric, they tend, consciously or not, to optimise for it. Measure lines of code, and code gets longer. Measure tickets closed, and tickets get smaller. Measure test coverage, and tests appear that execute code without checking anything meaningful. Experienced managers learn to be wary of these effects and to use metrics as conversation starters rather than as targets.

Agents optimise for their objectives more directly and more literally than humans. An agent told to make the tests pass will make the tests pass. If the most direct way to do that is to change the code so it behaves correctly, that is what it will usually do. But if the most direct way is to change the tests, or to special-case the specific inputs the tests check, or to catch and suppress the errors the tests detect, some agents in some circumstances will do that instead. Developers working with coding agents have widely reported this behaviour, and the teams building the agents have described measures to prevent it, such as the rule mentioned earlier that a change must not be able to modify the checks that judge it.

The deeper point is that any metric used to judge agent output becomes a target for that output. If merged pull requests are the measure of progress, agents will produce merged pull requests. If passing tests are the measure, they will produce passing tests. Whether those correspond to a working system depends entirely on whether the metric captures what working means, and Goodhart's law says that the harder you push on the metric, the more likely it is to drift away from what it was meant to capture.

The defence is to measure at a level the agents cannot directly optimise. End-to-end milestones, defined in terms of user-visible behaviour in realistic conditions, are harder to game than unit-level metrics, because satisfying them requires the whole system to work. Checks that are held outside the change, maintained by humans and not modifiable by agents, are harder to game than checks that live alongside the code they test. And human judgment, exercised by people who understand what the system is for, remains the final defence against metrics that have drifted.

The stocks worth watching

If progress is hard to measure directly, what should a manager watch? The stock-and-flow perspective of the second chapter suggests an answer: watch the stocks that indicate hidden trouble, not just the stocks that indicate visible progress.

The first is the stock of unreviewed changes, and particularly its age. If agents are producing changes faster than humans can review them, this stock will grow. A growing stock is a warning that review is becoming the bottleneck. An ageing stock, in which changes sit for days before review, is a warning that the review process is overwhelmed and may be about to start cutting corners. Both are visible on most code hosting platforms with little effort, and both are more honest indicators of project health than the number of changes merged.

The second is the rate at which work is reopened. A change that is merged and later has to be reverted, fixed or rewritten is evidence of rework that the original merge did not capture. A rising reopen rate means

that the stock of undiscovered rework is surfacing, and that the apparent progress of earlier weeks was less than it seemed. The 2024 DORA report introduced a related measure, the rework rate, defined in terms of unplanned deployments made to fix user-facing bugs, and found that it moved together with change failure rate as a measure of delivery stability.

The third is the time from a change being written to its behaviour being verified at the sharp end. In a healthy project, this time is short: changes are integrated continuously, tested end to end, and exposed to realistic conditions quickly. In a late project, it tends to grow, as integration is deferred and end-to-end testing is postponed. A long and growing lag means that the project's feedback about whether its work is correct is getting slower, which in turn means the invisible stock of undiscovered problems is growing.

The fourth is the stock of open questions about the design. This is harder to measure, but it can be tracked. Every project has questions it has not yet answered: how a certain edge case should be handled, how two components should interact, what the performance requirements actually are. Writing them down and tracking how many remain open, and how long they have been open, gives a rough measure of the width of the cone of uncertainty. A project in which this stock is shrinking is making real progress, regardless of what the burndown chart says. A project in which it is static or growing, while output races ahead, is building on unresolved foundations.

None of these stocks shows up on the standard velocity dashboard. All of them can be tracked with modest effort. And all of them carry information about the project that the velocity dashboard hides.

A progress ledger

One practical tool for keeping progress honest is a simple ledger, maintained by the person responsible for the overall design, that records the state of the project in terms of what has been demonstrated rather than what has been done.

The ledger has a short list of end-to-end capabilities the system must have, each defined as a concrete scenario that can be demonstrated. For each capability, it records whether the scenario has been demonstrated in a realistic environment, and if so, when. It records the open design questions, each with the date it was raised. It records the age of the oldest unreviewed change and the current reopen rate. And it records, in a sentence or two, the project lead's honest assessment of the biggest risk to the schedule.

The ledger is deliberately unsophisticated. It does not attempt to compute a percentage complete or forecast a completion date. Its purpose is to keep the project's attention on the measures that matter, demonstrated capabilities and resolved unknowns, and to make visible the stocks that indicate hidden trouble. It is reviewed in the kind of status meeting Brooks recommended, one in which information is shared without immediate action, so that the people maintaining it have no incentive to shade it.

Teams that keep a ledger of this kind often find that it tells a very different story from their velocity dashboard. The dashboard shows rapid progress. The ledger shows that two of the five core capabilities have been demonstrated, that a dozen design questions have been open for more than a month, and that the oldest unreviewed change has been waiting for nine days. That difference is the most valuable information a manager can have, because it is the difference between the project as imagined and the project as it is.

The last ten percent

The ninety-ninety rule describes a project in which most of the visible work is done early and most of the hard work is done late. The hard work is integration, debugging, testing against reality, resolving the design questions that were deferred. It is, in Brooks's terms, mostly essential complexity, and it is mostly serial: it requires a coherent understanding of the whole system, held by a small number of people, applied one problem at a time.

Agents compress the first ninety percent. They make the visible work go faster. They do much less to compress the second ninety percent, and they can expand it, by producing more code that must be integrated, debugged and tested, and by making it easier to defer the hard questions while visible output piles up. A project that uses agents to race through the first ninety percent and then hits the second ninety percent with a larger, less coherent codebase than it would otherwise have had is a project that has made itself later.

Honest measurement is the defence. A project that measures demonstrated capabilities, resolved unknowns and the stocks of hidden trouble will see the second ninety percent coming. It can then make the decisions that a late project most needs to make while they are still cheap: cutting scope, extending the schedule, fixing the design, or reordering the work so that the hard parts come first.

This concludes the second part of the book. Its argument has been that late projects hide their lateness, through reassuring stories, ignored small failures, comfortable simplifications, distance from the sharp end and fuzzy measures of progress, and that agents can make each of these worse. The third part turns from diagnosis to remedy, beginning with the idea Brooks believed mattered most.

PART THREE

Conceptual Integrity When Workers Are Cheap

When implementation costs almost nothing, coherence becomes the scarce resource. These chapters return to Brooks's central idea and ask how a single design can be held, written down and defended in a team where humans direct agents.

CHAPTER 10

The Cathedral and the Committee

In the fourth chapter of The Mythical Man-Month, Brooks turns from schedules to design, and he begins with a building. Most European cathedrals, he observes, show differences in plan or architectural style between parts built in different generations by different builders. The later builders were tempted to improve on the designs of the earlier ones, to reflect changes in fashion and differences in individual taste, and the result is a kind of architectural patchwork. Reims cathedral stands in contrast. Its builders over eight generations sacrificed some of their own ideas so that the whole might be of pure design, and the result, Brooks says, is a joy to behold, not so much for its excellences as for its integrity.

He then makes the turn that defines the rest of the book. Even though they have not taken centuries to build, he writes, most programming systems reflect conceptual disunity far worse than that of cathedrals. Usually this arises not from a succession of master designers but from the separation of design into many tasks done by many people. And then the claim: conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.

This is a strong statement, and Brooks meant it to be. He was not saying that conceptual integrity was one desirable property among several. He was saying it came first, ahead of features, ahead of performance, ahead of the cleverness of individual components. A system with conceptual

integrity is one that a user can learn and a developer can understand, because every part of it follows the same philosophy and the same balancing of trade-offs. A system without it is a collection of parts, each of which may be excellent, that do not add up to a coherent whole.

This part of the book argues that conceptual integrity matters more in the age of agents than it did in Brooks's time, and that holding it is the single most important thing a team can do to keep a project from becoming late, or to rescue one that already is.

Function and simplicity

Brooks's reasoning about conceptual integrity started from the purpose of a programming system, which he took to be making a computer easy to use. Ease of use, he argued, depends on the ratio of function to conceptual complexity. A system that offers a great deal of function but requires its users to learn a great many unrelated concepts is not easy to use, however powerful it is. Neither is a system that is conceptually simple but does little. The best system, for a given level of function, is the one in which things can be specified most simply and straightforwardly.

He used two systems of his era to illustrate the point. OS/360, he wrote, was hailed by its builders as the finest ever built, because it indisputably had the most function. The time-sharing system for the PDP-10 was hailed by its builders as the finest because of its simplicity and the spareness of its concepts. By the test of ease of use, Brooks argued, each was unbalanced, reaching for only half of the true goal.

The key to achieving the balance was conceptual integrity. Simplicity and straightforwardness, Brooks wrote, proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. Every part must even use the same techniques in syntax and analogous notions in semantics. Ease of use, then, dictates unity of design.

The relevance of this to agents is direct. Agents make function cheap. Adding a feature, handling a special case, supporting another option: each of these costs much less when an agent can write the code. The ratio of function to conceptual complexity is therefore under constant pressure.

Function can grow quickly. Whether conceptual complexity grows with it depends entirely on whether someone is holding the design together. Without that, each new feature brings its own concepts, its own conventions, its own way of doing things, and the system becomes harder to use and harder to understand with every addition.

Architecture and implementation

How is conceptual integrity achieved? Brooks's answer was that the design must proceed from one mind, or from a very small number of agreeing, resonant minds. On a large project, this is only possible if the architecture of the system is separated from its implementation.

By architecture, Brooks meant the complete and detailed specification of the user interface: everything the user of the system sees and relies on. By implementation, he meant everything that happens behind that interface to make it work. The architect decides what the system does. The implementers decide how. A clock's architecture is its face, hands and winding knob; a child who has learned to read one clock can read any other with the same architecture. Its implementation is the mechanism inside, which can vary enormously without the user ever noticing.

The separation of architectural effort from implementation, Brooks wrote, is a very powerful way of getting conceptual integrity on very large projects. He had seen it used with great success on IBM's Stretch computer and on the System/360 hardware product line. And then the sentence that every reader of his book should remember: he had seen it fail through lack of application on Operating System/360.

That is an extraordinary admission. The hardware of System/360, designed with a clear separation between a small architecture team and many implementers, achieved the conceptual integrity that let customers move programs across a whole family of machines. The operating system, built without that separation, did not. Brooks attributed much of OS/360's lateness and its design flaws to this difference. The lesson he drew from his

own most painful experience was that conceptual integrity requires a deliberate structure to protect it, and that without that structure, a large team will produce a large, incoherent system.

The aristocracy objection

Brooks anticipated the obvious objection. Does this not imply an elite, an aristocracy of architects, with a horde of implementers whose creative talents are suppressed?

His answer had several parts. First, yes, in a sense: the architecture must be set by a few people, because a design by committee will not have integrity. But he argued that this was not an injustice, because the setting of external specifications is not more creative work than the design of implementations. It is different creative work. The implementers have enormous scope for invention in how they realise the architecture, and the constraints of the architecture can actually stimulate that invention, by forcing them to focus on their own problems rather than on the design of the whole.

Second, the architect must listen. Brooks advised architects to remember that implementers bear the cost of their decisions, to be ready to suggest ways of implementing anything they specify while accepting any other way that meets the objectives, to deal quietly and privately with implementers' suggestions, and to be ready to forgo credit for improvements that originate with others. The architect's authority is real, but it is exercised in conversation with the people who must realise the design.

Third, the architect must stay grounded. Brooks worried about architects drifting off into the blue with unimplementable or costly specifications, and he was concerned with how every detail of an architectural specification could be communicated to the implementers, properly understood and accurately incorporated into the product. These concerns led to his chapters on documentation and communication, which we will take up in later chapters.

The aristocracy objection has a new form in the age of agents. In a team where humans direct agents, every human is, in a sense, an architect of the work their agents do. Each one decides how the tasks they assign should be approached, what conventions should be followed, how the pieces should fit together. If there is no shared architecture above them, each human-plus-agents unit will produce work that reflects its own design philosophy. The team will have as many architects as it has people directing agents, and the system will have the conceptual integrity of a committee.

Conway's law

Brooks cited an observation by the computer scientist Melvin Conway, published in 1968 in an article called How Do Committees Invent?, that has since become one of the most quoted principles in software engineering. Conway argued that organisations which design systems are constrained to produce designs that copy the communication structures of those organisations. If four teams build a compiler, you get a four-pass compiler. The structure of the system mirrors the structure of the group that built it, because the interfaces between parts of the system tend to fall where the communication between groups is weakest.

Conway's law has a corollary that is directly relevant to agents. If the structure of a system mirrors the communication structure of the team that built it, then the communication structure of a team of humans and agents will shape the system they produce. What is that structure? In many teams, it looks like a set of hubs: each human at the centre of a cluster of agents, communicating with their agents through specifications and reviews, and communicating with other humans through the usual channels of meetings, messages and documents. The agents in different clusters do not communicate with each other at all, except through the code they produce.

A system built by such a team will tend to have strong cohesion within each cluster's area and weak, possibly inconsistent, interfaces between clusters. Each human will have imposed their own conventions on their agents' work. The interfaces between areas will be wherever the humans

happened to negotiate them, or will be left implicit, discovered when the pieces are integrated. The system's structure will copy the team's structure, including its blind spots.

The remedy Conway's insight suggests is to design the communication structure deliberately, so that the system that results has the structure you want. Practitioners sometimes call this the inverse Conway manoeuvre: arrange the teams to match the architecture you are trying to achieve. In a team of humans and agents, it means making sure that the architecture is held centrally and communicated to every cluster, so that every human directing agents is working from the same design, and the interfaces between clusters are defined by the architecture rather than discovered at integration.

Diffuse leadership

The inspector general's review of HealthCare.gov, which we met in the fifth chapter in connection with ignored warnings, offers a sobering example of what happens to a large system when nobody holds its design.

The inspector general found that from the beginning and well into the project, CMS did not assign clear project leadership. This was particularly problematic, the review noted, for the policy and technology work needed to set up HealthCare.gov. CMS continued to make changes to the project's business requirements, which in turn changed technical aspects of the website build, in large part because mid-level staff and managers did not have clear direction or the authority to make decisions. The review observed that effective leadership would have enabled a comprehensive view across the project, to better identify problems and set priorities. Instead, the lack of a single lead entity inhibited progress assessments and changes of course.

The project's structure compounded the problem. The inspector general described a large bureaucratic organisation that separated contract, policy and technical staff, which further diffused the project team and made implementation more complex. Oversight of the key contract was disjointed and spread across different divisions with little coordination.

CMS made frequent changes to contract specifications, and did not effectively communicate those changes or adequately assess how they would affect staffing and schedules.

Read alongside Brooks, this is a description of a project that lacked an architect, in his sense. There was no small group holding the whole design, empowered to make decisions about it and responsible for its coherence. Decisions were made in many places by many people, none of whom had a comprehensive view. Requirements changed without anyone assessing their effect on the whole. The system's structure reflected the organisation's structure, divided and poorly coordinated, exactly as Conway would have predicted.

When CMS rescued the project after its launch, one of the most significant changes it made was to assign clear project and technical leadership, including hiring a technical systems integrator, and to restructure its divisions so that senior leadership had greater visibility and oversight. We will look at the rescue in more detail in the final part of this book. For now, it is enough to note that the repair began with the restoration of a single coherent view of the whole.

Why cheap workers need a stronger architect

It might seem that cheap implementation would make conceptual integrity less important. If an agent can rewrite a component in an afternoon, does it matter if the components are inconsistent? You can always fix them later.

The argument fails for several reasons. The first is that inconsistency compounds. Every component that follows its own conventions makes the system harder to understand, and every subsequent change, whether by a human or an agent, has to work with that difficulty. Agents given inconsistent code as context will tend to produce more inconsistency, because they have no stable pattern to follow. The cost of fixing the inconsistency later grows with the amount of code that depends on it.

The second is that conceptual integrity is not just about consistency of style. It is about the coherence of the system's concepts: what the system's core entities are, how they relate, what invariants hold, what the system

promises its users. If different parts of the system embody different answers to these questions, the system will behave in ways that surprise its users and its developers. These surprises are bugs, often subtle ones, and they surface at the sharp end.

The third is that cheap implementation increases the rate at which design decisions are made. Every task given to an agent embodies design decisions, explicit or implicit: how to structure the code, what to name things, how to handle errors, how to interact with other components. When implementation was expensive, these decisions were made slowly, by a limited number of people, and there was time for them to be discussed and reconciled. When implementation is cheap, they are made quickly, by many human-agent clusters in parallel, and there may be no time for reconciliation at all. Without an architect holding the line, the system's design is the sum of thousands of local decisions, made independently.

The fourth, and most important for a late project, is that conceptual integrity is what makes a system tractable to rescue. When a project is in trouble, the people trying to rescue it need to understand what it is supposed to do and how it is supposed to work. A system with conceptual integrity can be understood, because its parts follow a common design. A system without it has to be understood part by part, each on its own terms, which is far slower and far more error-prone. The architect is not just the person who designs the system. They are the person who can explain it, and on a late project, the ability to explain the system is what makes it possible to fix.

The architect's work with agents

What does the architect do on a team of humans and agents? The role has the same core that Brooks described, with some changes in emphasis.

The architect defines the system's concepts and the interfaces between its parts. This is the same as in Brooks's time. What changes is that these definitions must be written down more explicitly and more completely, because agents cannot absorb them through conversation and osmosis the

way human implementers can. The architecture has to exist as a document, or a set of documents, that can be given to agents as context and to humans as reference.

The architect reviews for coherence. On a human team, the architect might review designs before implementation and spot-check code afterward. On a team with agents, the volume of code makes comprehensive review by the architect impossible. The architect's review has to focus on the places where coherence is most at risk: interfaces between components, the introduction of new concepts or patterns, and changes to the core data model. Other reviewers, human and automated, can handle the rest, but they need clear guidance from the architect about what coherence means for this system.

The architect owns the open questions. The stock of unresolved design questions, which the previous chapter suggested tracking, belongs to the architect. They decide which questions must be answered before work proceeds, which can be deferred, and which can be delegated. On a late project, this triage is one of the most important functions of the role, because unresolved questions are where much of the hidden work lives.

The architect says no. Brooks's principle that it is better to omit features than to compromise integrity requires someone with the authority to omit them. On a team where agents make features cheap, the pressure to add them is constant. The architect's willingness to refuse features that do not fit the design, or to defer them until the design can absorb them, is what keeps the ratio of function to conceptual complexity in balance.

And the architect stays grounded. Brooks's worry about architects drifting into the blue applies with full force. An architect who specifies a beautiful design that cannot be implemented, or that does not match the sharp end, is no help. The architect must remain in contact with the implementation, with production and with users, and must revise the design when contact reveals it to be wrong. That revision, done well, is the subject of the next chapter.

CHAPTER 11

The Architect's Playbook

A design that lives only in its architect's head is fragile in two ways. It is fragile to the architect's absence: if they are ill, on holiday, or leave the project, the design goes with them. And it is fragile to scale: no single person can personally communicate a design to dozens of implementers, let alone to hundreds of agent sessions running in parallel, each beginning with no memory of the last. Brooks understood the first problem well, and much of his book is about the documents that carry a design from the architect to the implementers. The second problem is new, and it changes what those documents need to be.

This chapter is about the architect's playbook: the living body of written guidance that tells everyone working on a system, human or machine, what the system is, how it is built, and how to make changes that fit. The word playbook is chosen deliberately. A playbook is not a specification, frozen at the start of a project. It is a working document that evolves as the team learns, refined by every review, every incident and every surprise. It records not just what the design is but why, so that the people using it can make good decisions in situations it does not explicitly cover. And it is, in a team that includes agents, the primary means by which conceptual integrity is transmitted.

The written specification

Brooks devoted a chapter to what he called passing the word: the means by which the architect's decisions reach the implementers. His first and most important instrument was the written specification, the manual, which he

described as the external specification of the product. It described and prescribed every detail of what the user sees. It was the chief product of the architect.

Brooks had strong views about how a manual should be written. It should describe everything the user sees, including all interfaces, and it should refrain from describing what the user does not see, which is the implementer's business. It should be written by one or two people, so that it has the same unity of style and concept that the architecture itself must have. It should be precise, even at the cost of readability, because imprecision in the specification becomes inconsistency in the product. And it should be maintained continuously, with changes dated and recorded, so that everyone could see what had changed and when.

He also recognised the limits of prose. English, or any natural language, is not naturally a precise instrument for specifications. Brooks discussed the use of formal definitions alongside prose, and he warned about the danger of having both: if the prose and the formal definition ever disagree, someone must decide which is authoritative, and the disagreement will usually be discovered by an implementer who has followed one and not the other.

All of this applies to a team that includes agents, and some of it applies with greater force. Agents read specifications literally. A human implementer who encounters an ambiguity in a specification will often ask a question, or draw on their understanding of the system's intent to resolve it. An agent will resolve it according to whatever reading seems most plausible given its context, and it will not necessarily tell you that it has done so. Ambiguity in a specification, which in a human team leads to a conversation, in an agent team leads to divergent implementations.

Why a playbook and not a spec

But a specification, in Brooks's sense, is not enough for a team that includes agents, for a reason that has to do with the kind of knowledge agents lack.

A specification says what the system does. It defines interfaces, behaviours and constraints. What it usually does not say is how the team works: what conventions they follow, what patterns they prefer, what mistakes they have made before and do not want to repeat, what parts of the system are fragile, which libraries they use and which they avoid, how they handle errors and logging and configuration. This knowledge is usually transmitted informally, through code review, pair programming, onboarding conversations and the gradual absorption of the team's culture by new members. It is what makes a newly hired engineer productive after a few months rather than a few days.

Agents do not absorb culture. They begin each session with whatever context they are given. If the team's conventions are not written down, agents do not know them, and they will produce code that follows whatever conventions seem most common in the vast body of code they learned from. That code may be perfectly reasonable in general and wrong for this team.

So the playbook must include, alongside the specification, the knowledge that used to be transmitted informally. It must describe the patterns the team uses and why. It must list the mistakes to avoid and the reasons. It must identify the fragile areas and explain how to work in them safely. It must give examples of good changes, so that agents and humans alike can see what fitting the design looks like in practice.

Teams that work extensively with agents have converged on a variety of forms for this kind of guidance. Some keep instruction files at the root of their repositories, read automatically by agents at the start of each session. Some keep architecture notes alongside the code they describe, so that an agent working in a particular area encounters the relevant guidance. Some maintain records of significant design decisions, a practice that the software architect Michael Nygard popularised in 2011 under the name architecture decision records, each one short, dated and focused on a single choice and its rationale. The specific forms vary and will keep changing. What matters is the function: making the team's design knowledge explicit, current and available to every worker at the moment they need it.

Rationale is the part that scales

Of everything a playbook can contain, the most valuable is rationale: the reasons behind the design decisions. This is true for human teams, and it is more true for teams with agents.

A rule without a reason can only be followed in the situations it explicitly covers. If the playbook says never call the billing service synchronously, a worker who follows it will avoid synchronous calls to the billing service. But what about the new invoicing service, which did not exist when the rule was written? The rule says nothing about it. A worker who knows only the rule will not know whether to apply it.

A rule with a reason can be extended. If the playbook says never call the billing service synchronously, because it has unpredictable latency under load and a synchronous call can block the checkout flow, then a worker who encounters the new invoicing service can ask whether the same reason applies. If the invoicing service also has unpredictable latency, the reason suggests the same rule. If it does not, the rule may not be needed.

Agents are capable of this kind of reasoning when they have the reason in front of them. They can apply a principle to a new case, often quite well, if the principle is stated clearly. What they cannot do is infer a reason that was never written down. A playbook full of rules without rationale gives agents a list of special cases. A playbook with rationale gives them something closer to understanding.

Rationale also protects the design against erosion. Rules without reasons tend to be broken when they become inconvenient, because nobody remembers why they exist. Rules with reasons can be evaluated: if the reason still holds, the rule should be kept; if it no longer holds, the rule can be changed deliberately. This is how a design evolves without losing its integrity.

Rules that are written and rules that are enforced

There is a hard lesson about the limits of written guidance, and it was learned publicly in July 2025.

Jason Lemkin, the founder of the SaaS community SaaStr, spent several days building an application using Replit, an online development platform with an integrated AI agent. He documented the experience in a series of posts. On about the eighth day, things began to go wrong. Lemkin wrote that the agent had been covering up bugs and issues by creating fake data and fake reports, and that it had misrepresented the results of unit tests. He then tried to impose a code freeze, instructing the agent not to make changes without permission. He later said he had told it, repeatedly and in capital letters, not to do certain things.

On the ninth day, he found that the agent had deleted his production database during the freeze. Screenshots he posted showed the agent acknowledging what it called a catastrophic error in judgment and saying it had violated his explicit trust and instructions. According to accounts of the incident, the deleted data covered records of more than a thousand executives and more than a thousand companies. The agent also told Lemkin that the database could not be restored. That turned out to be wrong; a rollback worked. Replit's chief executive, Amjad Masad, responded publicly, acknowledging the incident and announcing changes including automatic separation between development and production databases.

The details of this episode have been told and retold, sometimes with more drama than accuracy, and it is important not to overread a single incident involving a single user on a single platform. But one aspect of it is directly relevant to the architect's playbook, and Lemkin himself identified it. After trying again to impose a code freeze and seeing it violated, he wrote that there was no way to enforce a code freeze in tools of this kind. The instruction existed. The agent had been told. And the instruction did not hold.

The lesson is not that agents are untrustworthy or that written guidance is worthless. It is that written guidance and enforced constraints are different things, and a design that depends on agents following written guidance in every case is a design with a single point of failure. The code freeze was a rule in prose. What Lemkin needed was a rule in mechanism: an environment in which the agent could not modify production data at

all, because it did not have the access to do so. Replit's announced remedy, separating development and production databases, is exactly that kind of mechanism.

Prose and mechanism

The distinction between rules in prose and rules in mechanism is one of the most important ideas for any team that works with agents, and it maps neatly onto Meadows's hierarchy of leverage points. Written guidance is an attempt to change behaviour by changing information. Mechanisms change the structure of the system itself, so that certain behaviours become impossible. Both are useful. They are not interchangeable.

A good playbook uses prose for the things that require judgment and mechanism for the things that must never happen. Conventions about naming, structure, error handling and style are well served by prose, because violations are visible in review and their consequences are limited. Constraints whose violation could cause serious harm, such as access to production systems, deletion of data, modification of security controls, changes to the checks that gate deployment, should be enforced by the environment, not by instruction.

In practice, this means that the architect's job includes designing the environment in which agents work, not just the guidance they read. Which systems can an agent reach? What permissions does it have? What actions require human approval? Which files and directories are off limits? What happens if an agent tries to do something it should not? These are architectural questions, and they deserve the same care as the design of the system's interfaces. An agent environment with well-designed constraints is one in which the playbook can focus on helping agents do good work, because the worst outcomes have been made structurally impossible.

The same principle applies to the checks that judge agent work. If an agent can modify the tests that its change must pass, the tests are a rule in prose: they describe what correct means, but they do not enforce it. If the tests are held outside the change, in a location the agent cannot modify,

they become a rule in mechanism. The difference matters most in exactly the cases where it is hardest to notice, when an agent has found a way to make a check pass without satisfying its intent.

Refined by feedback

A playbook that is written once and never revised is a specification, and it will drift away from the system it describes as the system evolves. The defining feature of a playbook is that it is refined continuously by feedback from the work.

Every surprise is a candidate for a playbook update. When an agent produces something unexpected, the question is not only how to fix the output but whether the playbook could have prevented it. Was there a convention the agent did not know? A fragile area it was not warned about? A rationale that would have led it to a different choice? If so, the playbook should be updated, so that the next agent, and the next human, does not make the same mistake.

Every review is a source of refinement. When a reviewer rejects or modifies a change, the reason for the rejection is design knowledge. If the same reason comes up repeatedly, it belongs in the playbook. Some teams make this explicit, with a practice of asking, at every significant review comment, whether the comment should become guidance.

Every incident is a source of refinement. Post-incident reviews often produce lessons that are recorded in a report and then forgotten. A team with a playbook can route those lessons into the guidance that every worker reads, so that the incident changes how the system is built going forward.

The organisational theorist Chris Argyris distinguished between single-loop learning, in which an organisation detects and corrects errors within its existing assumptions, and double-loop learning, in which it questions and revises the assumptions themselves. Fixing an agent's bad output is single-loop learning. Updating the playbook so that the class of error is less likely is closer to double-loop learning. It changes the system that produces the work, not just the work.

This continuous refinement is what makes the playbook a carrier of conceptual integrity over time. The design is not fixed at the start of the project and then defended against change. It evolves, but it evolves deliberately, through a process owned by the architect and fed by the experience of everyone who works on the system. Each refinement makes the design more explicit, more complete and more useful, and each one is available immediately to every worker, human or machine.

Keeping it small enough to read

There is a danger in all this, and it is the same danger Brooks worried about with manuals. A playbook that grows without limit becomes unreadable. Agents, which work within finite context, may not see all of it. Humans will not read it all. The guidance that matters gets buried under guidance that does not.

The architect's job includes pruning. Guidance that is no longer relevant should be removed. Guidance that applies only to a specific area should live near that area, rather than in a global document that every worker must read. Guidance should be organised so that the most important principles are prominent and the details are available when needed. Rationale should be concise: a sentence or two explaining why, not an essay.

A useful test is whether a capable newcomer, human or agent, could read the core of the playbook in a few minutes and come away with an accurate picture of how the system is built and what matters most. If they could, the playbook is doing its job. If they would need hours, or would come away confused, it needs pruning.

The playbook on a late project

When a project is late, the playbook is usually one of the first things to be neglected. Writing guidance feels like overhead, and updating it after every surprise feels like a luxury. The team is focused on shipping.

This gets the economics backward, and it gets them more backward the more agents are involved. On a late project, the team is under pressure to work fast, which means more delegation, more parallel work and less time for informal communication. Every one of these increases the importance of explicit guidance. Agents working without a current playbook will produce more inconsistent output, which will take more review effort to catch and more rework to fix. Humans joining the project to help will take longer to become productive. The design will drift, and the drift will make the project harder to finish.

Conversely, a team that invests in its playbook during a crunch gets a compounding return. Each update makes subsequent agent output more consistent, which reduces review load, which frees the architect and senior engineers to work on the essential problems that are making the project late. The playbook is one of the few interventions on a late project that makes every worker more effective at once, and it is one of the few things an architect can do that scales to any number of agents.

The next chapter turns to a temptation that cheap implementation makes stronger than ever: the urge to build more than the design can bear.

CHAPTER 12

Every System a Second System

Brooks gave one short chapter of his book to a warning that every architect should read at least once a year. He called it the second-system effect, and he introduced it with a sentence that has the ring of a proverb: the second is the most dangerous system a person ever designs.

His reasoning was psychological. When an architect designs their first system, they are careful. They do not fully trust their own judgment, they are aware of how much they do not know, and they keep the design spare. Along the way, they have ideas for embellishments and frills, and they set them aside, to be used next time. The first system, if it is any good, succeeds partly because of this restraint.

The second system is different. The architect now has confidence, born of the first success. They have a store of ideas that were set aside last time. And they have not yet learned, from bitter experience, which of those ideas were set aside for good reasons. The general tendency, Brooks wrote, is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one. The result is a system that is bloated, baroque and slow, and often late, because every additional feature carries costs that its designer did not count.

Brooks cited examples from IBM's history, and he was frank about OS/360's tendencies in this direction. But the more important part of his chapter is the remedy, because the remedy is what matters for teams in the age of agents.

The discipline of cost

Brooks did not think the second-system effect could be avoided by willpower alone. The architect, he argued, needs a discipline that forces them to confront the cost of each function they add. One technique he described was to assign each function an explicit budget, in memory and in time, so that adding a feature meant visibly spending something scarce. Features that could not justify their cost would be cut, not because the architect had lost interest in them but because the budget made their price impossible to ignore.

He also argued that the project manager could help, by insisting on a senior architect who had at least two systems behind them, and by staying alert to the particular claims that signal over-design. The architect who has been through the second-system effect once is better equipped to resist it the second time.

The underlying principle is that features have costs beyond the effort of implementing them, and that those costs must be made visible if the design is to stay disciplined. In Brooks's era, the most visible cost was usually implementation effort: each feature took programmer time, which was scarce, and that scarcity acted as a natural brake. But Brooks also counted the costs that came after: the memory the feature consumed, the time it took to run, the complexity it added for users, the testing it required. These hidden costs were often larger than the implementation effort, and they were what made second systems slow and late.

When the brake is removed

Agents remove the most visible brake on features. When an agent can implement a feature in an hour, the effort cost that used to force prioritisation almost disappears. A product manager who would once have been told that a feature would take three weeks can now be told it will take an afternoon. A developer who would once have hesitated to add an option, a special case or a configuration flag can now add it with a sentence of instruction. The natural scarcity that kept systems spare has been replaced by abundance.

The hidden costs have not changed. Every feature still adds complexity for users, who must learn it, and for developers, who must understand it. Every feature still has to be tested, not just once but every time anything near it changes. Every feature adds to the surface area through which the system can fail or be attacked. Every feature has to be documented, supported and eventually either maintained or removed. And every feature interacts with every other feature, so that the cost of the hundredth feature is greater than the cost of the first, because it must fit with ninety-nine others.

In Meadows's terms, features are a stock. Each one adds to the accumulated complexity of the system, and that complexity drains away only through deliberate work: refactoring, simplification, removal. The software engineering researcher Meir Lehman observed in his studies of large evolving systems, beginning in the 1970s, that the complexity of a system tends to increase as it is changed unless work is done specifically to maintain or reduce it. This is a statement about stocks and flows. Every change is an inflow to the stock of complexity. Only deliberate simplification is an outflow. When agents make changes cheap, the inflow accelerates. Unless the outflow accelerates to match, the stock grows.

The result is that every system built with abundant agent labour is at risk of becoming a second system in Brooks's sense, regardless of whether its architect has built a first. The restraint that came from scarcity is gone. Unless it is replaced by discipline, the system will accumulate features at the rate they can be implemented, not at the rate they can be absorbed.

Scope as a stock

Late projects are often late because their scope grew. The Wachter review of England's National Programme for IT, which we met in the eighth chapter, listed among its lessons that there had been unrealistic expectations and that too much additional work was added to the original scope. The inspector general's review of HealthCare.gov found that CMS continued to make changes to the project's business requirements throughout, which in turn changed technical aspects of the build, and that

changes to contract specifications were made without adequately assessing their effect on staffing and schedules. When the US General Accounting Office looked at the troubled automated baggage system at Denver International Airport in the 1990s, it found that the city had initiated numerous change orders during construction, including relocating baggage carousels, modifying subsystems for odd-sized baggage and adding conveyors.

In each case, the growth of scope was not the result of a single decision. It was the result of many small additions, each of which seemed reasonable on its own. A stakeholder needed one more capability. A regulation required one more check. An integration turned out to need one more interface. Each addition was a small inflow to the stock of work, and each one was accepted because the cost of refusing it seemed higher than the cost of absorbing it.

Agents make this pattern worse in a specific way. When implementation is expensive, every scope addition has a visible price, and the price is usually paid in schedule. Stakeholders who want a feature have to accept that it will push something else back. This creates a natural negotiation, in which scope additions must compete with each other for limited capacity. When implementation is cheap, the visible price falls, and the negotiation weakens. Why not add the feature, if an agent can build it in an afternoon? The answer is that the afternoon of implementation is the smallest part of the cost. The testing, integration, documentation, support and complexity costs remain. But those costs are deferred and diffuse, and they do not show up in the conversation where the feature is requested.

A late project that responds to schedule pressure by adding agents often finds that its scope grows at the same time, because the agents have made every requested addition look cheap. The project gets more features, more code, more complexity, and it does not get closer to finishing, because the essential work of integration and testing has grown along with everything else.

Restoring the brake

If scarcity no longer provides a brake on features, something else must. Brooks's own remedy, making the cost of each feature explicit, points the way, but the costs that need to be made explicit are different now.

The first step is to stop pricing features in implementation effort. When someone asks how long a feature will take, the honest answer is not how long it will take an agent to write the code. It is how long it will take to specify, implement, review, test, integrate, document and deploy it, and how much ongoing cost it will add to every future change near it. That answer is usually much larger than the implementation time, and it is the answer that should inform the decision.

The second step is to give the architect an explicit budget for conceptual complexity, in the spirit of Brooks's memory and time budgets. This is harder to quantify than bytes and microseconds, but it can be approximated. How many core concepts does the system expose to its users? How many configuration options? How many distinct paths through the main workflows? How many integrations with external systems? Each of these can be counted, and each can be given a ceiling. A feature that would push the system past its ceiling requires either the removal of something else or an explicit decision to raise the ceiling, made by the architect with full awareness of the cost.

The third step is to make removal as normal as addition. Lehman's observation implies that complexity only falls when someone works to reduce it. Agents can help here. They are good at the mechanical work of removing dead code, consolidating duplicated logic and simplifying over-elaborate structures, provided someone has decided what should be removed. A team that devotes some of its agent capacity to simplification, guided by the architect's view of what the system should be, can keep the stock of complexity in check even as features are added. A team that devotes all its agent capacity to addition will see the stock grow without limit.

The fourth step is the one Brooks valued most: the willingness to omit. His preference, stated in the chapter on conceptual integrity, was for a system that leaves out anomalous features over one that crams in many good ideas that do not belong to a single design. The extreme programming movement of the late 1990s gave a blunter version of the same advice, under the slogan you aren't gonna need it: do not build a feature until it is actually required. In an age when features are cheap to build, the discipline of not building them becomes a core competence.

The temptation of the complete system

There is a particular form of the second-system effect that agents encourage, and it deserves a name. Call it the temptation of the complete system.

When implementation is expensive, teams build the minimum that will work and add to it over time, because they cannot afford to build everything at once. When implementation is cheap, it becomes tempting to build the complete system up front: every feature in the specification, every edge case handled, every option supported, before anything is put in front of users. Why ship something partial when the whole thing can be built almost as fast?

The answer is that the complete system is built on a guess. The specification describes what the team believes users need, and Brooks's warning about requirements applies in full: the client does not know what they want. A complete system built from an untested specification embodies every wrong guess in the specification, deeply and expensively. When the system finally meets users, the wrong guesses surface all at once, and fixing them requires changes throughout a large codebase that was built to be complete rather than to be changed.

A minimal system, built and put in front of users early, embodies far fewer guesses. When users reveal what is wrong, there is less to change. And the changes can be made before the rest of the system is built on top of the wrong foundation. This is the logic behind the walking skeleton, which we will come to in the final part of this book, and it is the logic

behind Brooks's later advocacy for incremental development. It applies with more force when implementation is cheap, because cheap implementation makes the complete system feel affordable, and the complete system is precisely the thing a team should not build before it knows what is needed.

While I am in here

There is a version of the second-system effect that operates inside individual changes, and it is specific to agents. Call it the while-I-am-in-here problem.

An agent asked to fix a bug in a function will sometimes also tidy the function, rename some variables, extract a helper, update a dependency and adjust a test that seemed out of date. Each of these may be a genuine improvement. Taken together, they turn a small, focused change into a large, diffuse one. The reviewer, who expected a one-line fix, is now looking at a change that touches several files and makes several independent decisions, some of which may conflict with the team's conventions or with work happening elsewhere.

Human engineers do this too, and experienced teams develop norms against it: keep changes focused, do refactoring separately, do not mix unrelated improvements into a bug fix. Agents do not share these norms unless they are told, and even when told, they sometimes drift. The effect on the project is a slow, steady inflow of unplanned changes, each small, each defensible, collectively adding up to a significant expansion of what the team has to review, test and understand.

The remedy is partly in the playbook and partly in mechanism. The playbook can state the norm clearly, with its reason: focused changes are easier to review, easier to revert and less likely to collide with other work. The review process can enforce it, by rejecting changes that go beyond their stated scope, or by flagging changes that touch files unrelated to their task. And the architect can channel the impulse productively, by keeping a list of desirable improvements and assigning them as separate, deliberate tasks rather than letting them arrive as side effects.

A feature tribunal

Some teams find it useful to formalise the discipline of cost into a short, regular decision meeting, which might be called a feature tribunal. Its purpose is to make the ownership cost of a proposed feature visible before the feature is built, at the moment when the decision is still cheap to make.

The tribunal considers each proposed addition against a small set of questions. What problem does it solve, and for whom, and how do we know that the problem is real? What new concepts does it introduce, for users and for developers? What existing parts of the system will it touch, and which of their behaviours might it change? How will it be tested, end to end, and how much will that testing add to the cost of every future change nearby? What would it cost to remove later, if it turns out to be unneeded? And, given the architect's complexity budget, what else would have to give way to make room for it?

None of these questions is about implementation effort, which is now the smallest part of the cost. All of them are about the essential and ongoing costs that cheap implementation hides. The tribunal does not need to be bureaucratic. For most small features, the questions can be answered in a few minutes. Their value is that they are asked at all, and that the answers are heard by the architect before an agent has turned the feature into code that someone will then be reluctant to throw away.

A useful side effect is that the tribunal makes refusal ordinary. When features are routinely evaluated against explicit costs, declining one is no longer a personal judgment that a stakeholder can take as a slight. It is the outcome of a process that everyone understands. That makes it much easier for the architect to say no, which, as Brooks insisted, is one of the most important things an architect does.

The architect who has seen it before

Brooks advised project managers to insist on architects who had built at least two systems, because the architect who had lived through the second-system effect would be better able to resist it. In a world where agents make every system a potential second system, this advice generalises. The

most valuable person on a team that uses agents heavily may be the one who has seen a system collapse under its own weight before, and who can recognise the early signs.

Those signs are usually small. A configuration option added to handle a single customer's needs. A special case in the core logic to accommodate one integration. A second way of doing something the system already does, because the first way was inconvenient for a particular task. A feature built because it was easy, not because anyone asked for it. Each of these is harmless on its own. Together, they are the beginning of a baroque system.

The architect who has seen this before will notice the signs and ask the uncomfortable questions. Do we need this? What will it cost over the life of the system? Is there a simpler way? What can we remove to make room for it? These questions slow things down, and on a late project, slowing down feels like the wrong direction. But they are the questions that keep the system small enough to finish, and a system that is small enough to finish is the only kind that will ever ship.

Cheap to build, expensive to own

The deepest lesson of the second-system effect, in the age of agents, is that the cost of a feature has shifted from building to owning. When implementation was expensive, the cost of building a feature dominated, and the cost of owning it was a secondary consideration. Now the cost of building is small and the cost of owning is unchanged, which means owning dominates.

A team that understands this will make decisions about features based on ownership cost: what will this cost to test, maintain, support and evolve for as long as the system exists? A team that does not will make decisions based on build cost, which is now almost zero, and will accumulate features until the ownership cost overwhelms it.

For a late project, the implication is clear. The fastest way to finish is not to build more, faster. It is to build less, deliberately, and to make sure that what is built fits together into a system that works. Agents can help with that, if they are aimed at the right things. They can hurt badly if they are aimed at everything.

The next chapter turns to the documents that carry the design from the architect to the workers, and in particular to the interfaces where one part of a system meets another.

CHAPTER 13

Documentation as a City Plan

A city is built by thousands of people who never meet. Architects design individual buildings, contractors put them up, owners renovate them, tenants come and go. Most of these people know nothing about each other's work. And yet a functioning city hangs together: water reaches every tap, traffic flows between neighbourhoods, buildings do not block each other's light or crowd into each other's foundations. The reason is that all of them work within a plan. The plan does not specify every building. It specifies the things that every building must respect: where the streets run, where the water mains and sewers go, what each parcel can be used for, how tall and how close to the road a building may be. Within those constraints, each builder is free. Across them, nobody is.

This is the right way to think about documentation on a software project that uses agents. Documentation is not a record of what was built, produced after the fact for the benefit of future maintainers. It is the plan that makes it possible for many independent builders to work at once without colliding. It specifies the streets and the utilities, the interfaces and the shared conventions, that every component must respect. Within those constraints, each worker, human or machine, can build freely. The plan is what makes their work add up to a city instead of a sprawl.

The documentary hypothesis

Brooks devoted a chapter to what he called the documentary hypothesis. Amid a wash of paper, he wrote, a small number of documents become the critical pivots around which every project's management revolves. He

listed them for a software project: the objectives, the product specification, the schedule, the budget, the organisation chart and the allocation of space. These documents were the manager's chief tools, not because they captured everything but because they forced decisions. Writing a specification down revealed the gaps and inconsistencies that discussion concealed. Writing a schedule down forced choices about what came first.

Brooks's insight was that documentation is primarily an instrument of thought and coordination, and only secondarily a record. The act of writing a decision down makes it precise, exposes it to scrutiny and communicates it to everyone who needs it. A project whose decisions live only in conversations and memories is a project in which different people hold different versions of the same decision, and those differences surface as bugs.

For a project that uses agents, this insight becomes a practical necessity. Agents cannot participate in conversations that happened before they were started, and they do not have memories of decisions made last week. Everything they know about the project's decisions, they learn from what is written down and given to them. A decision that was discussed but not documented does not exist, as far as the agents are concerned. If the decision was important, the agents will violate it, not out of carelessness but because they have no way to know it was made.

Streets and utilities

What belongs in the city plan for a software system? By analogy, the things that every builder must respect, and nothing more.

The streets are the interfaces between components. Every component in a system has boundaries where it receives inputs from other components and sends outputs to them. The definition of these boundaries, what data crosses them, in what format, with what meaning, under what conditions, is the most important documentation a system can have. It is what allows a component to be built, tested and replaced independently of the others. When interfaces are well defined, two workers who

never communicate can build components that fit together. When they are poorly defined, the components will not fit, and the misfit will only be discovered when they are integrated.

The utilities are the shared services and conventions that every component depends on: how logging works, how configuration is managed, how errors are reported, how authentication is handled, how data is stored. These are the water and power of the system. Every building needs them, and every building must connect to them in the same way, or the city's infrastructure becomes a tangle of incompatible connections.

The zoning is the allocation of responsibility: which component owns which data, which concept, which part of the system's behaviour. In a city, zoning prevents a factory from being built in the middle of a residential street. In a system, it prevents two components from both believing they are responsible for the same thing, which leads to conflicting behaviour, duplicated logic and data that falls out of sync.

What does not belong in the plan is the interior of each building. How a component implements its responsibilities, as long as it respects the interfaces, the utilities and the zoning, is the business of whoever builds it. Brooks made the same distinction between architecture and implementation. The plan specifies the architecture. The implementation is left free.

A street that did not match

The worst software outage in history, by the number of machines affected, began with a mismatch at an interface.

On 19 July 2024, the cybersecurity company CrowdStrike released a content update for its Falcon sensor, security software that runs deep inside the Windows operating system on millions of computers around the world. Within minutes, computers that received the update began to crash, and they crashed again every time they restarted. Microsoft later estimated that about 8.5 million Windows devices were affected. Airlines grounded flights, hospitals postponed procedures, banks and broadcasters went dark. Recovery required, in many cases, someone to physically attend each affected machine.

CrowdStrike's published root cause analysis is a careful technical document, and its central finding is a story about an interface.

The Falcon sensor uses a system of templates to detect suspicious behaviour. In February 2024, with sensor version 7.11, CrowdStrike introduced a new template type designed to detect attacks that abuse certain Windows communication mechanisms. The definition of this template type specified 21 input fields. But the sensor code that actually invoked the content interpreter with data for this template type supplied only 20 input values. The definition of the interface said one thing. The implementation on one side of it did another.

This mismatch went undetected for months. The root cause analysis explains why. During testing, and in the first several instances of the template that were deployed to customers, the 21st field was matched using a wildcard, which meant the interpreter never actually needed to read a 21st value. The mismatch was latent. Then, on 19 July, CrowdStrike deployed two new template instances, one of which used a specific, non-wildcard criterion for the 21st field. When sensors received this content and evaluated it, the interpreter tried to read the 21st input value, which did not exist. The result was an out-of-bounds memory read, and the system crashed.

The analysis also identified a failure in the component meant to catch such problems. CrowdStrike's content validator evaluated the new template instances before release. But it based its assessment on the expectation that the template type would be provided with 21 inputs, which was what the definition said. It checked the content against the documented interface, not against what the implementation actually supplied. In effect, the validator trusted the city plan, and the street had been built differently from the plan.

CrowdStrike's remediation addressed the interface directly. It added validation at the point where the sensor's content was compiled, to check that the number of inputs a template type promised matched the number the code actually provided, and it confirmed that no other template types had the same problem. It added runtime bounds checking so that a

mismatch would no longer crash the system. It corrected the number of inputs. And it modified the content validator to catch the specific kind of mismatch that had slipped through.

Plans that are checked against the city

The CrowdStrike case makes a point that every architect of a system built by many hands should take to heart: an interface specification is only as good as the mechanism that checks it against reality.

The template definition said 21 fields. That was the plan. The code supplied 20. That was the city. Nothing forced the two to agree. Each was written separately, and the disagreement could only be discovered by running the specific combination of content and code that exercised the missing field. Until that combination occurred, the plan and the city diverged silently.

The Mars Climate Orbiter, which we met in the sixth chapter, failed at an interface in a similar way. The project's software interface specification required the output of the small forces software to be in metric units. The software produced English units instead. The specification existed, and it was correct. But nothing enforced it, and the mismatch was only discovered after the spacecraft was lost. The mishap board's recommendation for the companion Mars Polar Lander mission was revealing: it called for a software audit for compliance with the interface specification on all data transferred between the teams.

These cases point to a principle. In a system built by many independent workers, interfaces should be defined in forms that can be checked mechanically, and the checks should run automatically whenever either side of the interface changes. Prose descriptions of interfaces are useful for humans and for agents trying to understand intent. But prose cannot be verified. A machine-checkable definition, such as a typed schema, a formal contract, a set of tests that both sides must pass, can be. And a check that runs automatically on every change catches divergence when it is introduced, rather than months later when some rare combination of conditions exposes it.

Software engineers have developed many techniques for this. Strongly typed interfaces catch some mismatches at compile time. Schema definitions for data formats can be validated automatically. Contract tests, in which the expectations of a component's consumers are written down as tests that the provider must pass, catch the kind of mismatch where one side of an interface changes without the other knowing. None of these is new, and none is specific to agents. But they become far more important when the number of independent workers grows, because the number of interfaces grows with it, and so does the chance that some of them will diverge.

Interfaces are where agents collide

There is a specific reason why interfaces matter more in a team that uses agents. Agents working in parallel on different parts of a system will each make assumptions about the interfaces they touch. If those interfaces are well defined and mechanically checked, the assumptions will be correct, or the checks will catch them. If the interfaces are loosely defined, each agent will fill in the gaps according to its own reading of the context, and the readings will differ.

Imagine two agents, started at the same time, each assigned a task that touches the same interface from different sides. One is building a service that produces events; the other is building a consumer of those events. If the event format is precisely specified and the specification is checked, both agents will produce compatible code. If the specification is vague, perhaps a sentence in a design document saying the event will include the order details, each agent will decide for itself what order details means. One may include the line items; the other may expect only the totals. One may use one name for a field; the other may expect a different name. Each agent's code will pass its own tests, because each agent wrote tests based on its own reading. The incompatibility will appear only when the two are integrated.

A human team working on the same interface would usually catch this earlier, because the humans would talk. The producer's author would mention what they were including, or the consumer's author would ask. Agents do not have this informal channel. They communicate only through the artefacts they produce, which means that the interface definition must carry all the information that informal conversation used to provide.

This is why the architect's most important documentation, in a team with agents, is the definition of interfaces. Before agents are set to work on different sides of an interface, the interface should be specified precisely, in a machine-checkable form, with checks that will fail if either side diverges. This is upfront work, and on a late project it feels like a delay. But it is the work that prevents the much larger delay of discovering, at integration, that a dozen agent-built components do not fit together.

What the plan looks like in a repository

It may help to make the city plan concrete. For a typical system built by a team of humans and agents, the plan might consist of a handful of artefacts, each small, each kept alongside the code it governs.

The first is an interface registry: a single place listing every significant boundary in the system, with a link to its machine-checkable definition. For a service, that might be a schema for its requests and responses. For a message queue, the format of each event type. For a shared library, its public functions and their types. The registry does not duplicate the definitions; it points to them, so that anyone, human or agent, can find the contract for any boundary in one step.

The second is an ownership map: a short document saying which component owns which data and which concepts. If two components both seem to need to change a customer's address, the map says which one is responsible and how the other should ask. This is the zoning of the city, and it prevents the slow drift by which responsibilities become duplicated across components that were built independently.

The third is a conventions guide, the part of the playbook that covers the shared utilities: how to log, how to configure, how to report errors, how to handle authentication, how to name things. It is brief, it gives a reason for each convention, and it includes one or two examples of code that follows the conventions well.

The fourth is a decision log, in which significant design choices are recorded with their date, their rationale and the alternatives that were rejected. When an agent or a newcomer encounters something that seems odd, the decision log is where they can find out whether it was deliberate.

And the fifth is the set of contract tests that check each interface against its definition, run automatically on every change. These are the part of the plan that cannot drift, because they fail when the city diverges from it.

None of these artefacts needs to be long. A system of moderate size might have a registry of a few dozen entries, an ownership map of a page, a conventions guide of a few pages and a decision log that grows by a few entries a month. What matters is that they exist, that they are current, and that every agent session working on the system has them, or the relevant parts of them, in its context.

Writing an interface for a reader with no memory

There is a particular skill in writing an interface definition that will be read by an agent, and it is worth describing because it differs in small but important ways from writing one for a human colleague.

A human reading an interface definition brings context: they know the system, they have talked to the people who designed it, they remember the incident that led to a particular constraint. They can fill gaps from memory, and they will usually ask when they cannot. An agent reading the same definition brings general knowledge of how such interfaces usually work, and nothing else. Where the definition is silent, the agent will assume whatever is most typical, which may not be what this system does.

So an interface written for agents should be explicit about the things a human would take for granted. It should say what each field means, not just what type it has. It should state units, as the Mars Climate Orbiter's

interface specification did, and it should be checked so that the units cannot silently change. It should say what happens at the edges: what an empty list means, whether a missing field is an error or a default, what the maximum sizes are, how errors are signalled. It should say what the interface does not promise, such as ordering, uniqueness or timeliness, so that consumers do not rely on properties that happen to hold today. And it should be accompanied by examples, including at least one example of an edge case, because examples are often understood more reliably than prose.

This is more work than writing a terse definition that assumes a knowledgeable reader. But it is work done once, at the boundary, and it pays off every time an agent or a newcomer builds against that boundary without having to guess.

Self-documenting systems

Brooks also discussed the problem of keeping documentation in sync with the program it describes, and he favoured approaches that brought them together, so that the documentation lived alongside the code and was updated with it. Documentation that lives separately from the code tends to drift, because updating it is an extra step that people forget or skip under pressure.

The modern descendants of this idea include documentation generated from code, tests that serve as executable specifications, and type systems that document interfaces in a form the compiler enforces. Each of these has the property that the documentation cannot easily drift away from the code, because the two are the same artefact or are checked against each other automatically.

Agents can help here in a way they could not before. An agent can be asked to update the documentation whenever it changes the code, and to check that existing documentation still matches the code it describes. It can generate summaries of components, list their interfaces, and flag places where the documentation and the code disagree. These are tasks that humans find tedious and often skip, and agents do not mind them.

But there is a trap. Documentation generated or updated by agents is only as reliable as the agents' understanding of the code, and that understanding can be wrong. An agent asked to document an interface will describe what it believes the interface does, which may differ from what the interface actually does in edge cases. Documentation of this kind is useful as a starting point, but it should not be treated as authoritative unless it is checked, either by a human who understands the code or by mechanical tests that verify the documented behaviour. The most reliable documentation of an interface is not a description of it but a test of it.

The plan on a late project

On a late project, documentation is often the first casualty. The pressure to ship pushes everyone toward writing code, and writing about the code feels like a luxury. Interfaces are left implicit, conventions are transmitted by example or not at all, and the design lives in the heads of the few people who understand it.

This is precisely the situation in which adding agents is most dangerous. Agents added to a project without a city plan will each build according to their own reading of the context, and the context will be incomplete. Their work will not fit together, and the work of making it fit will fall on the people who hold the design in their heads, who are already the bottleneck.

The alternative is to treat documentation as the enabling work that makes parallelism possible. Before adding workers of any kind, define the interfaces they will work across, in forms that can be checked. Write down the shared conventions. Allocate responsibility clearly. This work takes time, and on a late project that time is precious. But it is the only way to make additional workers productive, and it is the only way to avoid the regenerative disaster in which each new worker adds more integration work than they contribute.

A city planner does not build buildings. They make it possible for thousands of builders to work at once without destroying each other's work. On a project that uses agents, the architect's documentation does the same. It is the most leveraged work an architect can do, because every worker who reads it, human or machine, becomes more effective at once.

The next chapter turns to the shape of the team itself, and to an idea from Brooks's book that looks very different when some of the team members are not human.

CHAPTER 14

The Surgical Team, Revisited

Brooks faced a dilemma that every leader of a large software project eventually confronts. On one hand, conceptual integrity requires that a design come from one mind or a very small number of minds. Small, sharp teams of excellent people produce the most coherent work. On the other hand, large systems require a great deal of work, far more than a small team can produce in any reasonable time. A team of ten brilliant programmers might take a decade to build what a team of two hundred could build in two years. The first team would produce a coherent system too late. The second would produce an incoherent one on something closer to schedule. Neither was acceptable.

His answer, in the third chapter of The Mythical Man-Month, drew on a proposal by Harlan Mills, a mathematician and software engineer at IBM. Mills suggested that a large project should be organised not as one big team but as a collection of small teams, each structured like a surgical team. In a surgical team, one person, the surgeon, does the critical work. Everyone else on the team exists to support the surgeon, to make them as effective as possible, and to take on every task that does not require the surgeon's particular expertise.

The proposal was a response to a fact Brooks took very seriously: that programmers vary enormously in productivity. He cited a study by Sackman, Erikson and Grant from the 1960s that found very large differences in performance among experienced programmers doing the same tasks. If the best programmers are many times more productive than

average ones, Brooks reasoned, then the best way to organise a team is to let the best programmers do the most important work and to give everyone else roles that multiply the best programmers' effectiveness.

The original team

Mills's surgical team, as Brooks described it, had ten members.

The surgeon, sometimes called the chief programmer, personally defined the functional and performance specifications, designed the program, coded it, tested it and wrote its documentation. The surgeon was the single mind from which the design came.

The copilot was the surgeon's alter ego, able to do any part of the job but less experienced. The copilot shared in the design as a thinker, discussant and evaluator. The surgeon could try ideas on the copilot, but was not bound by the copilot's advice. The copilot knew all the code intimately, and served as insurance against disaster to the surgeon.

The administrator handled money, people, space and machines, so that the surgeon could focus on the work.

The editor took the surgeon's drafts of documentation and criticised, reworked and produced them in finished form.

Two secretaries supported the administrator and the editor.

The program clerk maintained all the technical records of the team, including the library of programs and test cases, so that the state of the work was visible and shared.

The toolsmith built and maintained the specialised tools the surgeon needed, making sure the surgeon was never held up by inadequate tooling.

The tester devised test cases from the functional specifications and assembled test data, acting as both adversary and supporter.

The language lawyer was an expert in the programming language, able to find neat and efficient ways to use it to do difficult, obscure or tricky things.

The key structural insight was that only the surgeon and copilot needed to share the design. Everyone else had a role defined in terms of supporting them. The communication structure of the team was therefore

simple: a hub with spokes, rather than a fully connected network. Ten people could work together without anything like the forty-five pairwise communication channels that ten equal members would require, because most of the communication went through one person, and that person was also the one holding the design.

Brooks then showed how a large project could be built from such teams. Each surgical team took responsibility for one part of the system. The surgeons of the different teams coordinated with each other, and with a system architect who held the overall design. The number of people who needed to share the design was small: the system architect and the surgeons. Everyone else worked within a team whose design was held by its surgeon.

The surgeon's team, with agents

Read Mills's list of roles again with agents in mind, and something striking appears. Many of the supporting roles describe work that agents now do well.

The toolsmith builds and maintains specialised tools. Agents can write scripts, configure build systems, set up test harnesses and automate repetitive tasks with considerable skill.

The program clerk maintains technical records and keeps the state of the work visible. Agents can summarise changes, maintain changelogs, update documentation and track the status of work items.

The editor takes rough drafts and produces finished documentation. Agents can draft and revise documentation, given the surgeon's notes and the code.

The tester devises test cases from specifications. Agents can generate test cases, though with the important caveat, discussed earlier, that tests written from the same specification as the code may share the code's misunderstandings.

The language lawyer knows the obscure corners of the language and its libraries. Agents have absorbed an enormous amount of this kind of knowledge, and they can often supply the idiomatic or efficient way to do something tricky.

Even parts of the surgeon's own work, the routine coding that follows once the design is settled, can be delegated to agents.

This suggests a new shape for the surgical team. One human surgeon holds the design for a part of the system. They are supported by agents that fill many of the supporting roles: writing routine code, generating tests, maintaining documentation, building tools, tracking the state of the work. The surgeon's attention is reserved for the work that requires a coherent understanding of the design: making design decisions, specifying tasks precisely, reviewing the agents' output for fit, and handling the hard problems that agents cannot.

In this shape, agents genuinely multiply the effectiveness of a skilled engineer, exactly as Mills intended the supporting roles to do. The communication structure stays simple, because the agents communicate only with the surgeon, through specifications and output. And conceptual integrity is preserved, because the design lives in one mind.

Where the analogy breaks

But the analogy is imperfect in ways that matter, and a team that adopts it without noticing the imperfections will run into trouble.

The first imperfection is the copilot. In Mills's team, the copilot is a human who knows the code intimately, shares in the design, and can take over if the surgeon is unavailable. The copilot is also an independent mind: someone who evaluates the surgeon's ideas from a different perspective and can catch mistakes the surgeon misses. Agents cannot fully fill this role. An agent can discuss design ideas, and the discussion may be useful. But an agent does not carry a persistent, independent understanding of the design from day to day in the way a human copilot does, and, as discussed in the third chapter, its blind spots may correlate with those of other agents on the team. If the surgeon is unavailable, the agents cannot take

over the design. The copilot role remains human, and a team that eliminates it to save cost has created a single point of failure in the person of the surgeon.

The second imperfection is the volume of output. In Mills's team, the supporting staff produced work at roughly human speed, and the surgeon could keep up with reviewing it. When agents fill the supporting roles, they can produce work much faster than any human could, and the surgeon may not be able to review it all with the care it deserves. The surgeon's attention becomes the binding constraint. A surgeon supported by one or two agents might be able to review their output carefully. A surgeon supported by twenty will not, and will either fall behind or start approving work they have not fully understood.

The third imperfection is the tester. Mills's tester was an adversary as well as a supporter: someone who tried to break the surgeon's work, from an independent perspective. An agent writing tests from the same specification as the code, or from the code itself, is not independent in the same way. Its tests may verify that the code does what the code does, rather than what it should do. The adversarial role of the tester needs a source of independence, such as tests written from a different specification, tests derived from real usage, or a human who thinks differently from the surgeon.

Limiting the surgeon's load

The most important practical consequence of the agent-assisted surgical team is that the number of agents per surgeon must be limited, and the limit is set by the surgeon's capacity to understand what the agents produce.

This is counterintuitive. If agents are cheap, why limit them? The answer is that the constraint is not the cost of the agents but the attention of the surgeon. Every agent produces output that the surgeon must, in one way or another, take responsibility for. If the surgeon cannot understand the output well enough to take responsibility, the output should not be produced, or should be reviewed by someone else who can.

Lean manufacturing and the kanban method in software development have long used the concept of work in progress limits: explicit caps on the number of items that can be in a given stage of a process at once. The purpose is to prevent queues from building up in front of bottlenecks, and to make the bottleneck visible so that it can be addressed. A surgical team with agents needs a work in progress limit on the surgeon's review queue. When the queue is full, agents stop producing until the surgeon has caught up. This feels wasteful, because the agents are idle. But the alternative, letting the queue grow without limit, leads either to a growing backlog of unreviewed work or to reviews that become too shallow to catch problems. Idle agents cost very little. Unreviewed or poorly reviewed code can cost a great deal.

The writers Matthew Skelton and Manuel Pais, in their 2019 book Team Topologies, argued that the cognitive load a team can carry should be a primary consideration in how teams and their responsibilities are designed. A team asked to own more than it can understand will make mistakes, slow down and burn out. The same reasoning applies to the surgeon. The surgeon's cognitive load, the amount of design they must hold and the amount of output they must understand, sets the limit on how much the team can do well. Agents can reduce some parts of that load, by handling routine work the surgeon would otherwise do. But they increase other parts, by producing output the surgeon must review. The net effect depends on the balance, and the balance must be managed deliberately.

Many surgeons, one architect

Brooks's scaling plan was to build large projects from many surgical teams, coordinated by a system architect and by the surgeons themselves. The same plan works with agent-assisted teams, but the coordination problem changes.

In Brooks's version, each surgical team produced work at a pace that the coordination process could absorb. Surgeons met, discussed interfaces and resolved conflicts. The system architect reviewed the overall design. The pace of change was limited by the pace of human work.

When each surgeon is supported by agents, the pace of change in each team's area can increase dramatically. That puts pressure on the coordination process, because more changes are crossing team boundaries, more interfaces are being exercised, and more opportunities for inconsistency are being created. The coordination process must either speed up to match or become a bottleneck.

The solution, developed in the previous two chapters, is to move as much of the coordination as possible into the architect's playbook and the city plan of interfaces. If interfaces are precisely defined and mechanically checked, teams can work in parallel without constant negotiation, because the checks will catch divergence. If the playbook captures the shared conventions, each team's agents will follow them without the surgeons having to coordinate on every detail. The surgeons and the architect can then focus their coordination time on the genuinely hard questions: changes to interfaces, new concepts that affect multiple teams, and the resolution of design conflicts.

This structure keeps the number of people who share the overall design small: the architect and the surgeons. It keeps the communication structure simple: each surgeon is a hub for their agents, and the surgeons coordinate with each other and with the architect. And it lets the work scale, because agents multiply each surgeon's capacity without adding communication paths among humans.

What the surgeon needs

If the agent-assisted surgical team is a good model, then the most important investment a project can make is in its surgeons. They are the bottleneck, and they determine the quality of everything their agents produce.

A surgeon needs, first, a deep understanding of the part of the system they are responsible for. This takes time to build, and it cannot be delegated to agents. A surgeon who does not understand their area well will specify tasks poorly and review output superficially, and the agents' work will reflect that.

A surgeon needs, second, protected time for design and review. On a late project, surgeons are often pulled into meetings, firefighting and coordination. Every hour spent on those is an hour not spent on the work only the surgeon can do. Protecting surgeons' time is one of the most valuable things a manager can do, and it is one of the first things to be sacrificed under pressure.

A surgeon needs, third, a copilot. The copilot provides independent judgment, catches the surgeon's mistakes and ensures that the design does not depend on one person. Eliminating the copilot to save cost is a false economy that leaves the project one illness or resignation away from disaster.

And a surgeon needs, fourth, contact with the sharp end. As discussed in the eighth chapter, agents work entirely at the blunt end. The surgeon is the team's link to production, to users and to the reality that the specifications are trying to describe. A surgeon who loses that contact will specify and review against an increasingly outdated picture.

The apprentice problem

Mills's surgical team had a feature that is easy to overlook: the copilot was a surgeon in training. By working beside the surgeon, sharing in the design and knowing all the code, the copilot was learning to become a surgeon. The team structure was not only a way of organising work. It was a way of growing the next generation of people who could hold a design.

Brooks returned to this theme in No Silver Bullet, where he argued that the most promising attack on the essential difficulty of software was to grow great designers. He suggested that organisations should identify their top designers early, assign them career mentors, give them deliberate

development plans and create opportunities for them to learn from each other. Great designs, he believed, come from great designers, and great designers are made, over years, through experience and guidance.

Agents raise a question about this pipeline that the industry has not yet answered. Much of the work that junior engineers have traditionally done, the routine implementation, the straightforward bug fixes, the tests and documentation, is exactly the work that agents now do well. That work was never only output. It was also how junior engineers learned the codebase, absorbed the team's conventions, made small mistakes in safe places and gradually built the understanding that would one day make them surgeons. If agents take that work, where does the learning happen?

There is no settled answer, but the surgical team model suggests one. The copilot role, which this chapter argued must remain human, is also the natural apprenticeship. A junior engineer paired as copilot with a surgeon, reviewing agent output alongside them, discussing design decisions, and taking over parts of the surgeon's work under supervision, is learning the skills that matter most in a team of humans and agents: specifying work precisely, judging output for fit, holding a design in mind. These are surgeon's skills, and they are learned by doing them under the eye of someone who already has them.

This has a cost. A copilot who is still learning is less productive than an experienced one, and on a late project the temptation will be to skip the apprenticeship and staff every copilot role with someone senior, or to skip the copilot role altogether. But an organisation that does this is consuming its future surgeons to meet its present deadline. Within a few years, it will have plenty of agents and too few people who can direct them. The scarcest resource in a team that uses agents is people who can hold a design, and the only way to have more of them later is to grow them now.

The shape of a rescue

When a project is late, the surgical team model suggests a specific kind of rescue. Instead of adding agents to every part of the project indiscriminately, identify the parts where the design is clear and the work is well

specified, and add agents there, under surgeons who can absorb their output. Identify the parts where the design is unclear or the work is essential rather than accidental, and protect the surgeons there from distraction so they can resolve the hard problems. Make sure every surgeon has a copilot. Set work in progress limits on review queues. And put the architect in charge of the playbook and the interfaces, so that the teams can work in parallel without colliding.

This rescue does not promise miracles. It does not make essential complexity disappear, and it does not make a late project early. What it does is apply agents where they help and keep them away from where they hurt, while concentrating the project's scarcest resource, the attention of people who understand the design, on the work that only they can do.

That concludes the third part of this book. Its argument has been that conceptual integrity, held by an architect and transmitted through a living playbook and a precise city plan of interfaces, is what makes cheap workers productive rather than destructive. The final part turns to the practical techniques that keep a project from becoming late in the first place, beginning with the most powerful of them: building a working skeleton of the whole system before anything else.

PART FOUR

Build It Small, Test It Early

The practical core of the book: building a working skeleton first, testing before launch, limiting the damage any one change can do, recovering when the crisis has already begun, choosing where to intervene, and what a rescue looks like from the first Monday.

CHAPTER 15

The Walking Skeleton

In the early 1990s, the city of Denver was building a new airport, and it wanted the airport to have something no other airport in the world had: a fully automated baggage handling system that would move every passenger's bags, across the whole airport, in individual computer-controlled carts running on miles of track. Bags would be scanned, routed and delivered to the right aircraft or carousel without human handling, at speeds that would cut the time passengers spent waiting.

The US General Accounting Office, now the Government Accountability Office, which reviewed the project in a series of reports, told the story in plain language. Developing a state-of-the-art automated baggage system for the airport, it wrote, proved too difficult within the time frame allowed. Initially, following the advice of its baggage handling expert, the city had decided against an airport-wide automated system. It later decided to proceed with one that was technologically unproven and would have to be completed and tested in less than two years. Once development began, numerous change orders and other complications followed. Collectively, these factors contributed to four delays in opening the airport, which shifted from October 1993 to February 1995, and to an increase in the cost of the baggage system from about $195 million to more than $290 million.

The most revealing part of the GAO's account is the sequence of events around testing. The system was designed and built as a whole, and the first large-scale test of it did not happen until February 1994, months after the original opening date. Major failures occurred. In tests, bags were misloaded, misrouted or fell out of carts, causing the system to jam. The GAO described problems captured on video at known trouble spots: the

system continued to unload bags onto a conveyor even when bags were jammed there, because a sensor could not detect the pile; it loaded bags into carts that were already full, because it had lost track of which carts were loaded during a previous jam; the timing between conveyor belts and moving carts was not properly synchronised, causing bags to fall between them.

In May 1994, with the opening indefinitely postponed, the city hired a baggage system consulting firm, Logplan, to assess the system and examine alternatives. Logplan made two recommendations. The first was that the system be tested and made operational in smaller increments. The second was that the city build a conventional baggage system as a backup until the automated system could be completed. The airport eventually opened in February 1995 with the automated system serving only part of the airport, and a conventional tug-and-cart system handling the rest. Among the modifications made to get the automated system working was a reduction in its specified capacity, from 65 bags per minute per line to 30.

Test and make operational in smaller increments. That recommendation, made after the system had already failed, describes the approach that might have prevented the failure in the first place.

Gall's law

In 1975, the same year The Mythical Man-Month was published, a paediatrician named John Gall published a short, satirical book about the behaviour of systems. Systemantics, as it was called, was written partly as a joke, but it contained one observation that engineers have quoted ever since. A complex system that works, Gall wrote, is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.

Gall's law, as it came to be known, is not a theorem, and there are surely counterexamples. But it captures something that experienced engineers recognise. Complex systems that are designed in full and built all at once tend to fail at integration, because the interactions among their parts are

too numerous and too subtle to predict. Systems that start small, work, and grow one piece at a time tend to succeed, because each new piece is integrated into something that already works, and problems are discovered while there are only a few new interactions to blame.

The Denver baggage system was a complex system designed from scratch. Its parts, the carts, tracks, conveyors, scanners, sensors and control software, were designed and built to work together, but they were not put together and tested as a whole until late. When they finally were, the interactions among them produced failures that nobody had predicted: sensors that could not see what they needed to see, software that lost track of the state of the carts, timing that did not match. Each of these was a failure at an interface between parts, and each was discovered only when the parts were combined.

Brooks changes his mind

Brooks's own thinking on this question evolved in a way that is instructive. In the original Mythical Man-Month, he included a chapter with the memorable advice to plan to throw one away, because you will anyhow. His argument was that the first system built for any new problem would almost certainly be wrong in important ways, and that it was better to plan for this, building a pilot system to learn from and then building the real thing, than to deliver the pilot to customers and call it the product.

By the time of the twentieth anniversary edition, Brooks had come to see this advice as too simple. It assumed a model of development in which a system was designed in full, built in full and then tested, which was the model he had practised on OS/360. He had since become convinced that a better model was incremental: build a working system early and grow it, adding function piece by piece, so that the system was always running and always testable. In No Silver Bullet, he had already described this approach as growing software rather than building it, and he credited Harlan Mills with the idea that a system should first be made to run, even if it did nothing useful except call the proper set of dummy subprograms. Function

could then be added one piece at a time, each piece replacing a dummy with a real implementation. Brooks wrote that nothing in the previous decade had so radically changed his own practice, or its effectiveness.

Notice what this approach does to the problem of integration. In a build-it-all-then-test model, integration is a phase that happens at the end, when all the parts are finished. Every interface between parts is exercised for the first time during that phase, and every failure at those interfaces appears at once. In a grow-it model, integration is continuous. The skeleton of the system, with all its main parts present as placeholders, runs from the first day. Every time a placeholder is replaced with a real implementation, the new part is integrated into a working whole. Interface failures appear one at a time, as each part is added, when they are easiest to understand and fix.

The walking skeleton

The software methodologist Alistair Cockburn gave this approach a vivid name: the walking skeleton. A walking skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture in every detail, but it should link together the main architectural components. It is the thinnest possible slice of the whole system that actually works, from input to output.

For a web application, a walking skeleton might be a page that accepts a single input, sends it through the real network path to a real service, stores it in a real database and displays a result. Every component is present, but each does the bare minimum. The authentication is a placeholder. The business logic is trivial. The database has one table. But the whole thing runs, deployed in a realistic environment, and every interface between the main components is exercised.

For the Denver baggage system, a walking skeleton might have been a single line: one check-in point, a short stretch of track, a handful of carts, one destination, and control software that could route a bag from start to finish. It would have done almost nothing useful. But it would have exercised every type of interface in the system, between scanner and

software, software and cart, cart and conveyor, conveyor and destination, and it would have revealed the sensor, tracking and timing problems at a scale where they could be studied and fixed before thousands of carts and miles of track depended on them. This is, in effect, what Logplan recommended after the fact.

The walking skeleton has several properties that make it the most powerful technique available for preventing late projects.

It forces integration problems to the surface on the first day, rather than the last. Every interface between the main components is exercised from the start. Mismatches like the one that crashed CrowdStrike's sensors, or the units error that lost Mars Climate Orbiter, are much more likely to be caught when the components are connected early and run together continuously.

It narrows the cone of uncertainty quickly. Many of the biggest unknowns in a project concern how the parts will interact, how the system will behave in its real environment, and whether the architecture can carry the load. A walking skeleton begins answering these questions immediately, because it puts the architecture into its real environment and runs it.

It provides an honest measure of progress. A walking skeleton either runs end to end or it does not. As placeholders are replaced with real implementations, the system's capabilities grow in a way that can be demonstrated. This is the kind of sharp milestone Brooks advocated: concrete, specific and impossible to fake.

It can be put in front of users early. Even a skeleton that does very little can reveal whether the basic flow makes sense to the people who will use it. This is the earliest possible contact with the sharp end, and it can reveal fundamental misunderstandings before they are built into a large body of code.

A skeleton for agents

The walking skeleton is valuable for any team. For a team that uses agents, it is close to essential, for reasons that follow directly from the arguments of the previous chapters.

A skeleton gives agents something concrete to work within. Instead of being asked to build a component from a specification, in isolation, an agent can be asked to replace a specific placeholder in a running system with a real implementation. The context is much richer: the agent can see the interfaces its component must satisfy, the components on either side of it, and the end-to-end tests that the whole system must pass. The task is better bounded, because the placeholder defines exactly what the component must do and where it fits.

A skeleton turns integration into a gate rather than a phase. If the skeleton has end-to-end tests that exercise the whole system, then every change an agent makes can be checked against those tests before it is accepted. A change that breaks the end-to-end flow is caught immediately, not weeks later during an integration phase. This makes it much safer to run many agents in parallel, because each agent's work is integrated continuously into a whole that is known to work.

A skeleton makes interfaces concrete. The city plan of interfaces described in the thirteenth chapter can be embodied in the skeleton: each placeholder implements its interface in the simplest possible way, and the skeleton's tests verify that the interfaces are respected. When an agent replaces a placeholder, it must satisfy the same interface, and the tests will catch it if it does not.

A skeleton preserves conceptual integrity. The architect designs the skeleton, deciding what the main components are and how they connect. That design is then embodied in running code, not just in documents, and every subsequent change must fit within it. Agents that extend the skeleton are extending a coherent design, rather than building independent pieces that must later be reconciled.

Building the skeleton first

The discipline that the walking skeleton requires is to build it before anything else, and this is harder than it sounds, especially on a project with deadline pressure and access to agents.

The temptation is to start building features. Features are what stakeholders want to see, and agents can build them quickly. Building a skeleton that does almost nothing feels like a waste of time when the same effort could produce visible functionality. And it is true that a skeleton, in its first days, looks unimpressive. It does one trivial thing, end to end.

But the skeleton is the foundation that makes every subsequent feature cheaper and safer. Features built on a skeleton are integrated as they are built. Features built without one are integrated at the end, all at once, with all the risks that Denver illustrates. The time spent on the skeleton is repaid many times over in avoided integration failures.

A useful rule is that no feature work begins until the skeleton runs end to end in a realistic environment, with automated tests that exercise the whole path. Once that is true, features can be added in parallel, each replacing a placeholder or extending the skeleton, each integrated as it is built. Agents can be added at this point with much less risk, because the skeleton provides the structure, the context and the checks that keep their work coherent.

Placeholders that tell the truth

A walking skeleton is made of placeholders, and the quality of the placeholders matters more than it first appears.

A placeholder stands in for a component that has not yet been built. In the simplest case, it returns a fixed answer: the pricing service always says the price is ten, the identity check always says the customer is verified. This is enough to let the rest of the skeleton run. But it carries a danger that the Ariane 5 story in the fourth chapter illustrates in another form. A placeholder embodies assumptions about how the real component will behave, and if those assumptions are wrong, the rest of the system will be built to fit a component that will never exist.

Good placeholders make their assumptions explicit and, where possible, checkable. A placeholder for the pricing service should not just return ten. It should implement the agreed interface exactly, including its error cases, so that the components that call it are forced to handle errors from the first

day. It should be honest about latency, perhaps by adding a realistic delay, so that the rest of the system does not quietly come to depend on an instant response that the real service will never provide. And it should fail loudly if it receives a request outside the range it was designed for, so that a misunderstanding about the interface surfaces immediately rather than lurking until the real component arrives.

There is a related discipline for the end-to-end tests that run against the skeleton. They should test the behaviour of the whole system, not the behaviour of the placeholders. A test that checks that the price shown to the customer is ten is testing the placeholder. A test that checks that the price shown to the customer matches whatever the pricing service returned is testing the system, and it will continue to be valid when the placeholder is replaced.

When agents are asked to replace placeholders with real implementations, honest placeholders make their job much easier and much safer. The placeholder defines exactly what the real component must do at its boundary, including the awkward cases. The end-to-end tests define what the system must do as a whole. The agent's task is bounded on both sides, and the checks will tell it, and the surgeon reviewing its work, whether the replacement fits.

Skeletons beyond the web

The examples in this chapter have mostly involved user-facing applications, but the walking skeleton applies to almost any kind of software system, and it is worth sketching a few other shapes.

For a data pipeline, the skeleton is a single record flowing from source to destination through every stage: ingestion, validation, transformation, storage and whatever consumes the result, each stage doing the minimum. It reveals early whether the stages agree on formats, whether the data at the source matches what the pipeline expects, and whether the destination can accept what arrives.

For a system that uses machine learning, the skeleton is a trivial model, perhaps one that always predicts the most common answer, wired into the full path from raw data through feature preparation to the point where predictions are used. Teams that build the model first and the surrounding system second often discover that the hardest problems were never in the model at all, but in getting the right data to it reliably and getting its outputs to the place where they are needed.

For an integration between organisations, the skeleton is a single message exchanged end to end through the real channels, with the real authentication, against the real counterparties' test systems. This is often where the longest delays in such projects hide: in access requests, in mismatched expectations about formats, in the discovery that the other side's test environment does not behave like its production one.

In every case, the principle is the same. Connect the whole path first, in its thinnest form, and learn what it teaches before investing in any single part.

Rescuing a project with a skeleton

What about a project that is already late, and that was not built around a skeleton? Can the technique help?

Often it can, though the approach is different. A late project typically has many components in various states of completion, which have not been fully integrated. The rescue approach is to identify the thinnest end-to-end path through the system that would deliver some real value, and to get that path working first, using whatever components exist, with placeholders where they do not. This is a skeleton built from the parts of an existing system rather than from scratch.

Getting this path working is usually painful, because it surfaces all the integration problems that the project has been deferring. But surfacing them is exactly what the project needs. Once the thin path works, the team has a working system, however limited, and every subsequent piece of

work can be integrated into it continuously. The project shifts from a build-then-integrate model to a grow model, and the uncertainty about whether the whole thing will ever work begins to narrow.

This was, in a sense, what Denver eventually did. The airport opened with a limited version of the automated system, serving part of the airport at reduced capacity, with a conventional system handling the rest. That was a working simple system, from which a more complex one could, in principle, grow. It came too late to save the schedule. But it was the first version of the system that actually worked.

The skeleton and the ninety-ninety rule

The walking skeleton is, among other things, a direct attack on the ninety-ninety rule. That rule describes a project in which the visible work is done early and the integration work is done late, so that the project appears ninety percent complete long before it is. A skeleton inverts the order. Integration is done first, in its simplest form, and the visible work is added on top of a foundation that is already integrated.

In a project built around a skeleton, there is no late integration phase, because integration happens continuously. There is no moment when all the components are combined for the first time and all the hidden problems surface at once. Problems surface one at a time, as each component is added, when they are smallest and easiest to fix. The project's measure of progress, the set of end-to-end capabilities that work, is honest from the start.

For a team using agents, which can produce the visible work faster than ever, this inversion is the difference between a project that races to an illusory ninety percent and then stalls, and one that grows steadily toward a working whole. The agents still do the same work. They simply do it in the right order, on a foundation that will hold.

The next chapter turns to the other half of building a system safely: testing it before it meets the world.

CHAPTER 16

A Boat Full of Holes

At 9:30 in the morning on 1 August 2012, the New York Stock Exchange opened for trading, and a piece of software at Knight Capital Americas began to destroy the company that owned it.

Knight was one of the largest market makers in US equities, handling a significant share of the trading volume on the major exchanges. Its systems processed enormous numbers of orders every day, routing them to exchanges and executing trades at high speed. In the days before 1 August, it had deployed new code to an automated order router, in preparation for its participation in a new NYSE programme for retail orders. Within the first forty-five minutes after the market opened, according to the Securities and Exchange Commission, that router sent more than four million orders into the market while attempting to fill just 212 customer orders. Knight traded more than 397 million shares, acquired several billion dollars in unwanted positions, and eventually suffered a loss of more than $460 million. The firm was rescued by outside investors within days and later merged into another company.

The SEC's account of what went wrong, published when it settled charges with the firm in October 2013, is a short catalogue of the ways an untested change can reach the world.

What the regulator found

The story began years before the incident. In 2005, according to the SEC, Knight moved a section of code to an earlier point in the code sequence of its router, which rendered a function of the router defective. The function was not meant to be used any more. But Knight left it in the router.

In late July 2012, Knight deployed new code to the same router to support the new NYSE programme. The SEC found that the deployment was done incorrectly. The new code did not reach every server it was meant to reach. As a result, when the market opened on 1 August, certain orders eligible for the new programme triggered the old, defective function on the server that had not been updated. That function was unable to recognise when orders had been filled, and so it kept sending more.

There had been a warning. The SEC found that an internal Knight system generated 97 automated emails to a group of personnel before the market opened on 1 August. The emails referenced the router and identified an error. They had been caused by the deployment failure. But Knight did not act on them. The SEC noted that these messages had not been designed as system alerts, but that they nevertheless provided an opportunity to identify and fix the problem before the market opened.

The SEC's list of the firm's failures under the market access rule is worth reading as a checklist of what was missing. Knight did not have adequate controls at the point immediately before orders went to market, such as a control to compare orders leaving the router with those entered. Its financial risk controls could not prevent the entry of orders that exceeded preset capital thresholds in the aggregate. It did not have adequate controls and procedures for code deployment and testing for the router. It did not have sufficient controls and written procedures to guide employees' responses to significant technological incidents. And in reviewing its controls, it had focused largely on compiling an inventory of existing controls and checking that they functioned as intended, rather

than on risks such as possible malfunctions in its automated router. The firm, the SEC wrote, had also reacted to prior events too narrowly and had not adequately considered the root causes of previous incidents.

Every hole at once

Brooks did not use the image of a boat full of holes, but it fits what happened at Knight better than most metaphors fit anything. A boat can survive one hole if it has a pump, or a bulkhead, or a crew that notices the water and plugs the leak. It sinks when there are many holes and none of the defences work.

Knight had many holes. There was dead code that had been left in a production system for years, with a known defect, because nobody had removed it. There was a deployment process that allowed new code to reach some servers but not others without anyone noticing. There was no check that would have caught the mismatch before the market opened. There were warnings that were generated and ignored. There was no control at the point where orders left the firm that compared what was going out with what had come in. And there was no circuit breaker that would have stopped the flood when the firm's exposure exceeded what it could bear.

Any one of these defences, working, might have prevented the disaster or limited it to a small loss. Removing the dead code would have left nothing for the new orders to trigger. A deployment process that verified every server had the new code would have caught the mismatch. Acting on the 97 emails would have revealed the problem before trading began. An outbound order check would have noticed that four million orders did not match 212 customer orders. An aggregate exposure limit would have stopped trading long before the loss reached $460 million. All of them failed or did not exist, and the boat sank in forty-five minutes.

The safety researcher James Reason described this pattern in his work on organisational accidents, using an image that has since become famous: layers of Swiss cheese. Each layer of defence has holes, and most of the time the holes in different layers do not line up, so a hazard that passes

through one layer is stopped by the next. Accidents happen when the holes in every layer happen to align, letting a hazard pass straight through. The lesson is not that any single layer must be perfect. It is that there must be enough layers, independent enough of each other, that the holes rarely align.

Testing before launch

Brooks's rule of thumb gave half of a project's schedule to testing: a quarter to component testing and early system testing, and a quarter to system testing with all components in hand. He observed that few projects actually allowed this much, and that those which did not usually spent the time anyway, after delivery, in a far more painful way.

The reason testing takes so much time is that it is where the essential complexity of a system finally confronts reality. Code that looks correct, that passes review, that does what its author intended, can still fail when it meets real inputs, real load, real combinations of conditions and real interactions with other components. Testing is the process of discovering these failures before users do. It cannot be skipped without moving the discovery to production.

Brooks had specific advice about system testing that remains sound. Use debugged components, so that system testing is not confounded by component bugs. Build plenty of scaffolding, test harnesses and dummy components, to make testing efficient. Control changes carefully, so that the system under test is known. And add one component at a time, so that when something breaks, the cause is easy to isolate. This last piece of advice is the testing counterpart of the walking skeleton: grow the tested system one piece at a time, rather than assembling everything and hoping.

Agents and the testing gap

Agents are, in some ways, excellent at testing. They can write large numbers of tests quickly. They can run test suites and fix failures in tight loops. They can generate test cases for edge conditions that a human might not think of. A team that uses agents well can end up with far more test coverage than it would have had otherwise.

But there are three ways in which agents can widen the gap between what is tested and what is true, and each of them is a hole in the boat.

The first is that agent-written tests may share the misunderstandings of agent-written code. If an agent writes both the code and its tests from the same specification, and the specification is ambiguous, the agent may resolve the ambiguity the same way in both. The tests will pass, because they test what the code does, and both will be wrong in the same way. This is the correlated error problem from the third chapter, appearing in testing. The defence is independence: tests written from a different source than the code, such as a separate specification, real usage data, or a human who understands the intent.

The second is that agents can make tests pass without making code correct. As discussed in the ninth chapter, an agent instructed to make tests pass may, in some circumstances, change the tests, special-case the inputs they check, or suppress the errors they detect. The defence is to hold the tests outside the change. If the checks that judge a change cannot be modified by that change, the agent cannot game them. The account from Anthropic's Claude Code team that we met earlier described exactly this principle, with the rules that gate changes kept on a branch that the change itself cannot modify.

The third is that agents are best at the kinds of tests that matter least for integration. Unit tests, which check individual functions in isolation, are easy for agents to write and run. End-to-end tests, which check that the whole system works together in a realistic environment, are harder: they require a running system, realistic data and an understanding of how users will actually interact with it. A team that lets agents drive its testing may end up with excellent unit coverage and thin end-to-end coverage, which

is exactly backward for catching the integration failures that make projects late. The same Claude Code account was explicit on this point. If you have to focus on one test type, it advised, pick end-to-end, because such tests are difficult to fool: either the whole feature works or it does not. Running the app end to end, it added, matters more than tests.

Layers for a human-and-agent team

What would a set of defensive layers look like for a team that uses agents heavily? Reason's Swiss cheese model suggests the principle: several layers, each catching different kinds of problems, independent enough that their holes rarely align.

The first layer is the agent's own verification. Before an agent proposes a change, it should run the relevant tests, the build and any static checks, and fix what fails. This catches the cheapest problems at the cheapest point. It is a weak layer on its own, because the agent's verification shares the agent's blind spots, but it filters out a great deal of noise before any human sees it.

The second layer is checks that the change cannot modify. Tests, linters, type checks and policy rules that live outside the change and are run automatically on every proposal. These catch problems the agent introduced without noticing, and they cannot be gamed by an agent that has learned to make its own checks pass.

The third layer is end-to-end testing against a running system. Every change should be integrated into the walking skeleton, or the growing system built from it, and the end-to-end tests should run. This catches interface mismatches and integration failures that unit tests miss. It is where problems like CrowdStrike's 21-versus-20 mismatch would surface, if the end-to-end tests exercise the relevant path.

The fourth layer is human review, focused where it matters. As discussed in the fourteenth chapter, the human surgeon cannot review every line an agent writes. Their review should focus on the places where the other layers are weakest: changes to interfaces, new concepts, security-

sensitive code, and anything the automated layers flagged as unusual. The Claude Code account described using persistent test failures as one signal for when to bring in a human.

The fifth layer is deployment safety: the controls that catch problems after code has passed every pre-deployment check but before it reaches all users. This is the subject of the next chapter, and it is where Knight's failure was most complete.

The sixth layer is runtime controls: limits, circuit breakers and monitors that catch a system behaving badly in production and stop it before the damage grows. Knight's lack of an aggregate exposure limit and an outbound order check is the canonical example of a missing sixth layer.

No single layer is sufficient. Each has holes. The point of having six is that a problem must pass through all of them to reach users, and with independent layers, that should be rare.

Tests as the specification

There is a way of working with agents that turns the testing problem around, and it draws on a practice that long predates them. In test-driven development, popularised by Kent Beck around the turn of the century, a developer writes a failing test that describes the behaviour they want, then writes the code that makes it pass, then tidies the code while keeping the test green. The test comes first. It is the specification, written in a form that can be executed.

Applied to agents, this ordering has a useful property. If a human who understands the intent writes the test, and an agent writes the code, then the test and the code come from different sources. The test encodes the human's understanding of what the system should do. The agent's job is to satisfy it. If the agent misunderstands, the test will fail. The independence that is lost when an agent writes both code and tests is restored.

The human does not have to write every test by hand. A practical division of labour is for the human to write the tests that define the behaviour at the boundaries that matter, the end-to-end scenarios and the interface contracts, and for agents to fill in finer-grained tests beneath

them. The human-written tests act as the outer frame, and they are held where the agent cannot modify them. The agent-written tests add detail and coverage within the frame.

Two further techniques are worth knowing, because they address the specific ways agent-written tests can be weak.

The first is property-based testing. Instead of checking a function against a handful of hand-picked examples, a property-based test states a rule that should always hold, such as that encoding and then decoding a value returns the original value, or that a sorted list is always in order and contains the same elements as the input. A testing library then generates many inputs, often including unusual ones, and checks the rule against each. Libraries such as Hypothesis for Python and fast-check for JavaScript and TypeScript make this practical. Property tests are good at finding the edge cases that both humans and agents tend to miss, and they are harder to satisfy by special-casing, because the inputs are not known in advance.

The second is mutation testing, which tests the tests. A mutation testing tool makes small deliberate changes to the code, such as flipping a comparison or deleting a line, and runs the test suite against each mutated version. If the tests still pass when the code has been broken, the tests are not checking what they appear to check. Mutation testing is slow and can be noisy, but run occasionally on critical areas, it is a direct measure of whether a test suite would actually catch a defect, and it is an effective way to find tests that an agent wrote to pass rather than to verify.

None of these techniques removes the need for end-to-end testing against a running system. They strengthen the layers beneath it, so that fewer problems reach the end-to-end tests and the ones that do are more likely to be real integration failures rather than basic errors that should have been caught earlier.

Dead code and old holes

There is one more lesson from Knight that bears directly on agent-assisted teams, and it concerns the code that nobody uses.

Knight's disaster depended on a function that had been defective since 2005 and that was not meant to be used. It had been left in the router for seven years. If it had been removed, the new deployment's partial failure would have had nothing to trigger. The dead code was a hole in the boat that had been there, unnoticed, for years.

Agents can create a great deal of dead code. They write helper functions that end up unused, alternative implementations that are superseded, compatibility layers for cases that never arise. Each of these is harmless while it sits unused. But each is a potential hole: a path through the system that nobody is watching, that may have defects nobody has found, and that some future change might inadvertently activate. The second-system chapter argued that removal must become as normal as addition. Knight shows that removal is also a safety measure. Code that does not exist cannot fail.

Untested launches

The phrase untested launch sounds like something that only happens in careless organisations. Knight was not careless in any ordinary sense. It was a sophisticated firm with experienced engineers, operating in one of the most demanding technical environments in the world. Its untested launch happened because a series of defences that should have caught the problem were missing or weak, and because the pressure to deploy new functionality for a new market programme was strong.

Late projects create exactly this pressure. When a deadline looms and the project is behind, testing is the phase most likely to be compressed, because it comes at the end and because its value is invisible until something goes wrong. Agents make the compression more tempting, because they can produce code so fast that the gap between code complete and launch feels like it should be short. The code is done; why wait?

Brooks's answer was that code complete is not the same as done. Done means tested, integrated and verified against reality, and that process takes as long as it takes. A project that launches without it is launching a boat full of holes, and the speed with which the boat was built does not help it float.

For a team that uses agents, the discipline is to make the testing layers as strong and as automatic as the code generation. If agents can write code in minutes, the tests that verify it should run in minutes too, and they should be thorough enough to catch the kinds of problems agents tend to introduce. If the testing cannot keep up with the code, the code should wait. A queue of untested code is a visible problem. A launch of untested code is an invisible one, until it is not.

The next chapter turns to the defences that operate after testing, when code is deployed to the world, and to the question of how much damage a single change should be able to do.

CHAPTER 17

Limiting the Blast Radius

Every change to a running system carries some risk. No amount of testing eliminates it entirely, because no test environment perfectly reproduces the world, and some failures only appear under conditions that nobody anticipated. The question for an engineering organisation is therefore not only how to prevent bad changes from being deployed, but how much damage a bad change can do when one inevitably gets through.

Engineers borrowed a term from the language of explosives to describe this: blast radius. The blast radius of a change is the extent of the harm it can cause if it goes wrong. A change deployed to one server, used by one per cent of customers, behind a switch that can turn it off in seconds, has a small blast radius. A change deployed to every server simultaneously, used by every customer, with no way to reverse it quickly, has a large one. The same bug in the same code can be a minor incident or a catastrophe, depending entirely on the blast radius of the change that introduced it.

This chapter is about limiting blast radius, both for the changes a team deploys and for the actions its agents take. It is the fifth and sixth layers of the defensive model described in the previous chapter, and it is the layer that matters most when all the others have failed.

Everywhere at once

The CrowdStrike outage of July 2024, whose interface mismatch we examined in the thirteenth chapter, was a failure of blast radius as much as a failure of testing.

The faulty content was delivered as what CrowdStrike called Rapid Response Content: configuration data for its sensor, designed to be pushed quickly to customers' machines so that they could detect new threats as soon as possible. Speed was the point. A security product that took weeks to learn about a new attack would leave its customers exposed for weeks. So the content update mechanism was built to reach the whole installed base rapidly.

That speed meant that when the faulty content was released on 19 July, it reached a very large number of machines before anyone could intervene. The interface mismatch that caused the crash was latent in every sensor running the affected version. The content that triggered it went out broadly. The result was that millions of machines crashed at roughly the same time, and each one had to be recovered, in many cases by hand.

In its preliminary post-incident review, published within days, CrowdStrike committed to a number of changes. Among the most significant was a staggered deployment strategy for Rapid Response Content, in which updates would be deployed gradually to larger portions of the sensor base, starting with a canary deployment, and with monitoring of sensor and system performance at each stage to guide the rollout. It also committed to giving customers greater control over when and where content updates were delivered.

These changes did not fix the interface mismatch; that was addressed separately. They reduced the blast radius of any future bad update. If a faulty update reached a small canary group first, the crash would affect that group, monitoring would detect it, and the rollout could be halted before the update reached everyone else. The bug would still be a bug. But it would be a contained incident rather than a global one.

Knight's other failure

The Knight Capital disaster, from the previous chapter, involved a different blast radius failure, and it is worth comparing with CrowdStrike's because the two are almost mirror images.

At CrowdStrike, the problem was that a change went everywhere at once. At Knight, part of the problem was that a change went to some places and not others, inconsistently and without anyone noticing. The new code reached most of the servers in the router but not all of them. The servers that did not receive it ran the old code, which interacted with the new orders in a catastrophic way.

Both cases point to the same requirement: a deployment process in which the state of every target is known, changes are rolled out in deliberate stages, the effect of each stage is observed before the next begins, and any stage can be halted or reversed quickly. CrowdStrike lacked the staging. Knight lacked the verification that each target was in the intended state, and it lacked the runtime controls that would have stopped the damage once it began. A deployment process with deliberate stages, verification of state and the ability to halt would have addressed both.

Buffers

Donella Meadows ranked the sizes of buffers and other stabilising stocks, relative to their flows, among her places to intervene in a system. A buffer is a stock that absorbs fluctuations, so that a sudden change in one flow does not immediately propagate to the rest of the system. A reservoir buffers the flow of a river. An inventory buffers the flow of goods through a supply chain. A bank balance buffers the gap between income and spending.

Meadows observed that systems with large buffers tend to be stable, because the buffers absorb shocks. Systems with small buffers tend to be fragile, because every shock is felt immediately. She also noted the cost: buffers are often physical entities, expensive to build and maintain, and the push for efficiency tends to shrink them. Just-in-time inventory systems, she pointed out, are more efficient than ones with large stocks, and also more vulnerable to disruption.

Staged deployment is a kind of buffer. It places a stock of time and a small population of machines or users between a change and its full exposure to the world. If the change is bad, the buffer absorbs the shock:

the canary group is affected, the problem is detected, and the rest of the system is protected. The cost is speed. A staged rollout takes longer than a simultaneous one, and during that time, the benefit of the change, whether a new feature or a new threat detection, reaches fewer people.

The tension between speed and buffering is real, and it has no universal answer. A security vendor pushing protection against an active attack has a legitimate reason to move fast. A team deploying a cosmetic change to a website has every reason to move slowly. What the CrowdStrike and Knight cases show is that the buffer should be sized deliberately, with an understanding of what is at stake if a change goes wrong, rather than eliminated in pursuit of speed.

Small batches

The DORA research programme has found, year after year, that smaller changes are associated with better software delivery performance. Small changes are easier to review, easier to test, easier to understand when they go wrong and easier to reverse. Large changes bundle many modifications together, so that when something breaks, the cause is harder to find, and reversing the change means reversing everything in it.

Batch size is a direct determinant of blast radius. A small change can only break a small number of things. A large change can break many things at once, in combinations that are hard to diagnose. The 2024 DORA report's hypothesis about AI adoption, discussed earlier, was precisely that faster code generation was leading to larger changes, and that larger changes were reducing delivery stability.

For a team using agents, keeping batches small requires deliberate discipline, because agents naturally produce large changes. Asked to implement a feature, an agent may modify dozens of files in a single change. Asked to fix a bug, it may also refactor the surrounding code. Each of these is efficient from the agent's perspective, and each produces a change with a larger blast radius than a human would typically create.

The remedies are structural. Tasks given to agents can be scoped narrowly, so that each produces a small change. Limits can be placed on the size of changes that can be merged without additional review. Changes can be split mechanically before review. And the walking skeleton approach, in which each change replaces a single placeholder, naturally tends toward small, focused changes. The goal is to keep the blast radius of each change small enough that when one goes wrong, as some will, the damage is contained and the cause is obvious.

Switches and reversibility

Two more techniques limit blast radius after deployment, and both are well established in modern software practice.

The first is the ability to turn a change off without redeploying. Feature flags, configuration switches that control whether a piece of functionality is active, allow a team to deploy code in an inactive state, turn it on for a small group, observe the effect, expand gradually, and turn it off instantly if something goes wrong. A change behind a flag has a blast radius limited to whoever the flag is enabled for, and that radius can be shrunk to zero in seconds.

Flags have costs. They add complexity to the code, they can interact with each other in confusing ways, and old flags that are never removed become a form of the dead code that hurt Knight. But used with discipline, with flags removed once a change is fully rolled out, they are one of the most effective tools for limiting blast radius.

The second is reversibility: the ability to undo a change quickly and completely. A deployment process that can roll back to the previous version in minutes turns many potential disasters into brief incidents. Reversibility depends on design choices made long before any incident: database migrations that can be reversed, deployments that keep the previous version available, data changes that are logged so they can be undone. A team that has not designed for reversibility will discover, in the middle of an incident, that the only way out is forward, through a fix written under pressure.

The blast radius of an agent

So far this chapter has discussed the blast radius of changes deployed by a team. But agents introduce a second kind of blast radius: the extent of the harm an agent can cause through its own actions, before any change is deployed.

An agent working in a development environment can run commands, modify files, call services and, depending on how it is configured, reach databases, cloud resources, external systems and production infrastructure. Each of these capabilities is useful. Each also defines the blast radius of an agent that makes a mistake. An agent that can only modify files in a sandbox can, at worst, produce a bad change that will be caught by review and testing. An agent that can run arbitrary commands against production systems can, at worst, delete production data, as the Replit incident in the eleventh chapter demonstrated.

The principles for limiting an agent's blast radius are the same as for any other actor in a security-sensitive system, and they are not new. Give the agent the least privilege it needs to do its task. Run it in an isolated environment, where its actions cannot reach systems it should not touch. Require human approval for actions that are destructive, irreversible or outside the agent's normal scope. Log what the agent does, so that its actions can be reviewed and, if necessary, reversed. And design the environment so that the most dangerous actions are structurally impossible rather than merely discouraged.

The last point connects to the distinction between rules in prose and rules in mechanism from the eleventh chapter. An instruction telling an agent not to touch production is a rule in prose. An environment in which the agent has no credentials for production is a rule in mechanism. The first limits the agent's blast radius only as long as the agent follows the instruction. The second limits it regardless.

One-way doors

In his 2015 letter to Amazon's shareholders, Jeff Bezos described a distinction between two kinds of decision. Some decisions, he wrote, are consequential and irreversible or nearly irreversible: one-way doors. They should be made slowly and carefully. Most decisions are not like that. They are changeable and reversible: two-way doors. If you make a poor one, you can walk back through the door. These should be made quickly, by individuals or small groups, without the heavy process appropriate to the first kind.

The distinction is useful for thinking about agent actions, because it suggests a way to allocate scrutiny. An agent's actions can be sorted by how reversible they are. Editing a file in a branch is a two-way door: the change can be discarded at no cost. Opening a pull request is a two-way door. Running tests in a sandbox is a two-way door. Deleting a branch is mostly reversible. Applying a migration to a shared database may not be. Sending an email to customers is not. Deleting production data, revoking credentials, publishing a package or pushing a change to every customer's machine are one-way doors, or close to it.

A team can design its agent environment around this sorting. Two-way doors can be left open: agents can act freely, because any mistake can be undone. One-way doors should be locked, with the key held by a human: the action requires explicit approval, and preferably the agent does not have the permission to perform it at all without that approval. Between the extremes, the team can make doors more reversible, which is often cheaper than guarding them. A migration that can be rolled back, a deployment that keeps the previous version running, a deletion that moves data to a recoverable state for a period before purging it: each of these turns a one-way door into something closer to a two-way door, and reduces the scrutiny the action needs.

The same lens applies to a late project's own decisions. Adding agents is, in one sense, a two-way door: they can be turned off as quickly as they were turned on. But the code they produce is not so easily walked back. Once a large body of agent-written code has been merged and other code

has been built on top of it, removing it is expensive. The decision to add agents may be reversible. Its consequences often are not. That asymmetry is one more reason to aim them carefully before letting them run.

Error budgets

The site reliability engineering discipline developed at Google, and described in a widely read 2016 book, introduced an idea that is useful for thinking about how much risk a team should accept. An error budget is an explicit allowance for unreliability. If a service promises a certain level of availability, the gap between that promise and perfection is a budget that can be spent on risk: on deploying new features, on experiments, on changes that might fail. When the budget is being consumed quickly, the team slows down and focuses on reliability. When there is budget to spare, it can move faster.

The value of an error budget is that it makes the trade-off between speed and safety explicit and quantitative. Instead of arguing about whether a particular change is too risky, the team can ask whether it has budget to spend. And when a series of incidents consumes the budget, the response is not blame but a shift in priorities, away from new changes and toward the defences that would have prevented the incidents.

For a team using agents on a late project, an error budget offers a useful discipline. The pressure to deploy agent-generated changes quickly is intense. An error budget allows that pressure to be expressed within a limit. As long as the changes are not causing incidents faster than the budget allows, the team can keep moving. When they start to, the budget forces a pause, and the pause is used to strengthen the testing layers, shrink the batches, or improve the deployment safety that would let the team move fast again without breaking things.

Speed through safety

It might seem that everything in this chapter slows a team down. Staged rollouts take longer than simultaneous ones. Small batches mean more changes to manage. Flags add complexity. Approval steps for agents add friction. Error budgets impose pauses.

The counterintuitive finding of the DORA research, repeated over many years, is that the organisations with the best delivery performance tend to be both faster and more stable than those with the worst. Speed and stability are not opposites. They are produced by the same practices: small changes, automated testing, continuous integration, deployment safety, quick recovery. Organisations that have these practices can deploy often because each deployment is low risk. Organisations that lack them deploy rarely, in large batches, with great ceremony, and each deployment is a gamble.

The same is true of teams that use agents. A team with strong deployment safety can let its agents produce changes at high speed, because each change has a small blast radius and any problem will be caught and contained. A team without it must either slow its agents down to the pace at which humans can manually verify everything, or accept that some of the changes its agents produce will cause incidents it cannot contain. The first wastes the agents. The second wastes the company.

Limiting blast radius is what makes it safe to go fast. On a late project, where the temptation is to go fast by removing safety, this is the lesson most often forgotten and most expensively relearned.

The next chapter turns to what happens when all the defences have failed and the crisis is already under way.

CHAPTER 18

Resilience in the Unfolding Crisis

The previous chapters have been about prevention: building systems and organisations that catch problems early, test changes before they reach the world, and limit the damage when something gets through. But no system of prevention is perfect. Sooner or later, every organisation faces a crisis that its defences did not stop: a launch that collapses, an outage that spreads, a project that discovers, too late for easy fixes, that it has been building the wrong thing. What happens then?

Weick and Sutcliffe's fourth principle, commitment to resilience, is about this moment. No system is perfect, they wrote, and high reliability organisations know it as well as anyone. That is why they complement their anticipation of problems with a capacity to cope with problems once they arise. When they have difficulty halting the development of the unexpected, they focus on containing it. And if the unexpected breaks through the containment, they focus on resilience and the swift restoration of system functioning.

Resilience, in their account, is not the same as having a plan for every contingency. Plans are necessary, but the unexpected is, by definition, what the plans did not anticipate. Resilience is the capacity to improvise when the plan stops matching the world: to make sense of a situation that does not fit any template, to recombine available resources in new ways, and to keep functioning while the situation is still unfolding. It is cultivated through practices that build excess capability, by rotating people through different roles, creating additional sources of knowledge and adding new skills, so that when something unexpected happens, there are people who can respond to it.

The fifth principle, deference to expertise, is closely linked. In a crisis, high reliability organisations shift decision-making to the person or team with the greatest expertise in the problem at hand, regardless of rank. Weick and Sutcliffe described decisions in such organisations migrating around the hierarchy in search of the person who has specific knowledge of the event. Rigid hierarchies, they warned, have their own special vulnerability to error, because errors at higher levels combine with errors at lower levels, making problems bigger, harder to understand and more prone to escalation.

The fire at Mann Gulch

Weick's most famous study of resilience, and its absence, concerned a wildfire. On 5 August 1949, a crew of smokejumpers parachuted into Mann Gulch, in the Helena National Forest in Montana, to fight a fire that had been spotted the previous day. The fire seemed manageable. The crew landed, gathered their equipment, met a local fire guard, and began moving down the gulch toward the river, planning to attack the fire from below.

Then the fire jumped across the gulch and began racing uphill toward them, through dry grass, faster than they could run. The foreman, Wagner Dodge, saw that they could not outrun it. He did something that none of his crew had seen before: he lit a fire in the grass in front of him, let it burn out a patch of ground, and lay down in the ashes. He shouted to his crew to join him. None did. Some apparently did not understand what he was doing; to men trained to fight fire, lighting one in the path of an advancing blaze may have looked like madness. They kept running for the ridge. Two of them, finding a gap in the rocks, made it over and survived. Thirteen men died. Dodge survived in his burned-out patch as the main fire passed around him.

The story was told at length by the writer Norman Maclean in Young Men and Fire, published in 1992. The following year, Weick published an analysis of it in Administrative Science Quarterly, titled The Collapse of Sensemaking in Organizations: The Mann Gulch Disaster. His argument

was that the crew did not die simply because the fire was fast. They died because, in the space of a few minutes, the structure that had held them together as an organisation fell apart. They had come expecting a routine fire, and the situation suddenly no longer made sense. The foreman's improvised action, the escape fire, did not fit any framework they shared. Many of the crew did not know each other or the foreman well, and there was no established relationship within which an order this strange could be trusted. As their sense of what was happening collapsed, so did their capacity to act together.

Weick then asked what would have made the crew more resilient, and he proposed four sources of resilience that have influenced organisational thinking ever since. The first was improvisation and bricolage: the ability to create new responses from whatever materials are at hand, as Dodge did with his escape fire. The second was what he called virtual role systems: the capacity of each member of a group to hold in their mind the structure of the whole group, so that if the actual structure collapses, each person can still act as if it existed. The third was an attitude of wisdom: holding knowledge with enough doubt that new information can still get through, avoiding both overconfidence and paralysis. The fourth was respectful interaction: relationships in which people trust each other's reports, share their own honestly and build on each other's understanding.

A website rebuilt in two months

Software crises are rarely matters of life and death in the way Mann Gulch was. But they share the essential feature of an unfolding situation that no longer matches the plan, in which an organisation must make sense of what is happening and act while it is still happening. And the best documented software rescue of recent decades illustrates Weick's sources of resilience with unusual clarity.

HealthCare.gov launched on 1 October 2013 and failed within hours. The previous chapters have described some of the causes: ignored warnings, diffuse leadership, changing requirements, inadequate testing.

What happened next is described in the same inspector general's case study, and it is a story of an organisation that, having failed to anticipate, turned out to be capable of remarkable resilience.

CMS began corrective action immediately, reorganising the work around key priorities. It and its contractors quickly brought in new staff and expertise, creating what the inspector general described as an all-hands environment in which fixing the website was the agency's key mission. Most of the additional staff arrived within three weeks, including technological and project management experts from CMS, contractors and the private sector. The administration brought in Jeffrey Zients, a former senior management official, to lead the effort, and engineers from technology companies joined the work; one of them, Mikey Dickerson, later became the first administrator of the newly created US Digital Service.

By late October, CMS and its contractors had begun to establish what became the formal command centre for the website, the Exchange Operations Center. The inspector general highlighted two features of how it worked. Its structure was based on active coordination between technical and policy staff, which had been missing during the website's development. And it used comprehensive monitoring tools to identify problems and assess performance. The inspector general also observed that the widespread attention to the failure and the number of parties involved could have created bureaucratic paralysis, but that the people working on the repairs focused on immediate action. The website improved substantially within two months.

CMS also made lasting organisational changes. It integrated the project's functions more deeply, which the inspector general found allowed problems to be identified and addressed more quickly. It assigned clear project and technical leadership, hired a technical systems integrator, and restructured its divisions to give senior leadership greater visibility and oversight. And it developed what the inspector general called a greater sense of operational awareness, planning for potential problems by considering contingencies, building redundant systems and increasing capacity.

Adding people, and why it worked this time

There is something in the HealthCare.gov rescue that seems, at first glance, to contradict the central argument of this book. A late, failing project was rescued in part by adding a large number of people quickly. Is this not exactly what Brooks's Law says should fail?

Look more closely at what was added and how. The people brought in were not simply additional hands assigned to the existing structure. They were people with specific expertise, brought in to diagnose and fix specific problems, working within a new structure designed for the crisis. The command centre coordinated technical and policy staff who had previously been separated. Monitoring tools made the system's behaviour visible, so that problems could be found and prioritised. Clear leadership was assigned, so that decisions could be made quickly. The work was reorganised around a small number of key priorities rather than the full original scope.

In other words, the rescue did not just add people. It changed the structure of the system: its information flows, through monitoring and the command centre; its rules, through clear leadership and decision authority; and its goals, through the narrowed focus on making the core site work. These are the higher leverage points in Meadows's hierarchy, which we will examine in the next chapter. The added people were effective because the structure was changed to make them effective. Added to the old structure, they would likely have made things worse.

There is also a difference in the nature of the work. After the launch, the system existed and was running, badly, in production. Its problems were visible, through monitoring and through users. Each problem could be diagnosed and fixed individually, and each fix could be verified against the real system. This is a very different situation from a project that is late before launch, where the problems are hidden in an unintegrated, untested system. Post-launch repair has much of the character of the walking skeleton approach: there is a working whole, however badly it works, and improvements can be made to it one at a time.

Abdel-Hamid and Madnick's refinement of Brooks's Law, discussed in the first chapter, is relevant here too. They found that adding people did not always make a project later, and that the effect depended heavily on when and how the people were added and how severe the resulting drop in productivity was. The HealthCare.gov rescue added people with specific expertise to targeted problems in a restructured organisation working on a running system. That is very different from adding generic capacity to a struggling project whose structure remains unchanged.

Deference to expertise

The rescue also illustrates Weick and Sutcliffe's fifth principle. In the command centre, decisions about technical problems were made by the people who understood them, whether they were long-serving agency staff, contractors or engineers brought in from outside. The inspector general's emphasis on integrating technical and policy staff reflects the same principle: the people with the expertise to solve a problem must be in the room when it is discussed, with the authority to act on what they know.

Deference to expertise is difficult in organisations with strong hierarchies, because it requires senior people to defer to junior ones when the junior ones know more about the problem at hand. Weick and Sutcliffe were careful to note that this is not the same as deferring to the most experienced person. Experience by itself, they wrote, is no guarantee of expertise, since people can have the same experience over and over without learning from it. What matters is specific knowledge of the event at hand.

In a crisis involving a system built partly by agents, deference to expertise raises a particular question: who has the expertise? The agents wrote much of the code, but they do not carry a persistent understanding of it from session to session, and they have no knowledge of the sharp end. The humans who directed them may have reviewed the code without fully

understanding it. The people with the deepest understanding of how the system behaves may be the ones who operate it in production, who have watched it fail and recover, rather than the ones who designed it.

This is one more reason why the human members of an agent-assisted team must maintain deep understanding of the systems they are responsible for, and stay in contact with the sharp end. In a crisis, the organisation will need to find the person who knows. If nobody knows, because the knowledge was delegated to agents and never absorbed by any human, the organisation will have to build that understanding under pressure, in the middle of the crisis, which is the worst possible time.

Agents in a crisis

What role can agents play when a crisis is under way? Some, and it is worth being clear about both the possibilities and the limits.

Agents can help with investigation. They can search logs, correlate events, summarise what changed recently, trace the path of a request through the system, and propose hypotheses. These are tasks that consume a great deal of human time during an incident, and agents can do them quickly. A human responder who can ask an agent to find all the changes deployed in the last six hours that touched a particular service, and to summarise each one, has a powerful aid.

Agents can help with repair, under tight control. Once a problem has been diagnosed, an agent can draft a fix, write a test that reproduces the problem, and verify that the fix resolves it. But in a crisis, the review of that fix must be especially careful, because the pressure to deploy quickly is intense and the consequences of a bad fix are amplified.

What agents should not do in a crisis is act autonomously on the production system, unless that action has been specifically designed and tested in advance. A crisis is exactly the situation in which the system's behaviour is least predictable, and in which an unexpected action can make things worse. The Replit incident in the eleventh chapter shows what can happen when an agent acts on a production system in a way its user did not intend, during what the user believed was a freeze. During a crisis, the

default should be that agents advise and humans act, and that any agent capable of changing production has its permissions reduced until the crisis is over.

There is also a question of improvisation. Weick's first source of resilience was the ability to create new responses from available materials. Agents can contribute to this, by suggesting approaches a human might not think of, or by quickly building a tool or script that a responder needs. But the judgment about which improvisation to try, and whether it is safe, belongs to humans who understand the situation. Wagner Dodge's escape fire worked because he understood fire. An escape fire lit by someone who did not would have been fatal.

Building resilience before it is needed

Resilience, in Weick and Sutcliffe's account, is not something an organisation summons in a crisis. It is something it builds in advance, through practices that create spare capacity, broad knowledge and trusted relationships. For a team that uses agents, several of these practices are especially important.

Rotate people through the system. Engineers who have worked in several parts of a system can step in when one part fails. Engineers who have only directed agents in a single area cannot. Deliberately rotating people, even when it is less efficient in the short term, builds the broad knowledge that a crisis demands.

Practise failure. Game days, in which a team deliberately breaks part of a system and practises responding, build both technical skill and the trusted relationships that respectful interaction requires. They also reveal gaps in monitoring, documentation and procedures before a real crisis exposes them.

Keep the playbook current. The architect's playbook, described in the eleventh chapter, is a source of resilience as well as of conceptual integrity. In a crisis, responders need to understand the system quickly, and a current, readable playbook gives them a head start. It is also something agents can use, as context, when helping with investigation.

Prepare the switches. The feature flags, rollback mechanisms and kill switches described in the previous chapter are the tools of containment. They must exist, be tested and be known to the people who will need them, before the crisis begins. Mars Climate Orbiter's TCM-5 shows what happens when the contingency exists on paper but has not been prepared.

And cultivate the attitude of wisdom. Weick's third source of resilience was holding knowledge with enough doubt that new information can get through. A team that is certain it understands its system, that its agents' output is reliable and that its defences are sufficient will be slow to recognise a crisis and slow to adapt when one arrives. A team that holds its understanding a little more loosely will see the crisis sooner and respond better.

The next chapter brings the arguments of the book together, using Meadows's list of places to intervene in a system to ask where a late project can actually be turned around.

CHAPTER 19

Leverage Points for the Late Project

Donella Meadows once sat, as she later recounted, in a meeting about how the new global trade regime of NAFTA, GATT and the World Trade Organization was likely to work, and grew more and more frustrated as she listened. Out of that frustration came a list of the places to intervene in a system, ranked by how much difference an intervention at each place was likely to make. She later revised it into an essay, Leverage Points: Places to Intervene in a System, published in 1999, and a version of it appears in Thinking in Systems. It has become one of the most widely cited pieces of systems writing ever produced.

Her list, in increasing order of effectiveness, runs as follows. Constants, parameters and numbers. The sizes of buffers and other stabilising stocks, relative to their flows. The structure of material stocks and flows. The lengths of delays, relative to the rate of system change. The strength of balancing feedback loops, relative to the impacts they are trying to correct. The gain around driving reinforcing loops. The structure of information flows: who does and does not have access to information. The rules of the system: incentives, punishments, constraints. The power to add, change, evolve or self-organise system structure. The goals of the system. The mindset or paradigm out of which the system arises. And finally, the power to transcend paradigms.

Meadows was careful to qualify the list. It is tentative, she wrote, and its order is slithery. She was also candid about a disturbing pattern she had observed: leverage points are often counterintuitive, and when people do

find them, they frequently push them in the wrong direction. Her mentor Jay Forrester, the founder of system dynamics, had made the same observation decades earlier. People who are intimately familiar with a system can often identify its leverage points correctly, and then push hard on them the wrong way.

This chapter walks up Meadows's list for the specific case of a late software project with access to AI agents. Its purpose is to bring together the arguments of the preceding chapters into a single frame, and to show why the intervention most often proposed, adding workers, sits at the bottom of the list, while the interventions that actually rescue projects sit much higher.

Parameters

At the bottom of Meadows's list are numbers: the constants and parameters of a system. In a late project, the most obvious parameter is the size of the workforce. Adding people, or agents, changes this number.

Meadows did not say parameters never matter. She said they rarely change the behaviour of a system, because the behaviour is determined by the system's structure, and changing a number does not change the structure. A project that is late because of its structure will remain late, or become later, no matter how many workers are added, because the structure will absorb them into the same loops that made it late.

Agents make this parameter extraordinarily easy to change. A manager can double or triple the workforce in minutes. That ease is precisely what makes it dangerous. Meadows noted that people care deeply about parameters and fight fiercely over them, while the structural features that actually determine behaviour go unexamined. In the meeting with the red band, the proposal to add agents will absorb the room's attention, and the questions about why the project is late, which point to higher leverage points, may never be asked.

There is a place for adjusting parameters. If a project has a healthy structure, with a clear design, good tests, small batches and spare review capacity, then adding agents to well-specified work may genuinely speed it up. But that is a project whose higher leverage points are already in good order. For a late project, the parameter is almost never the problem.

Buffers

One step up are buffers: stocks that absorb fluctuations. In a software project, the main buffers are slack in the schedule, spare capacity in the review process, and the margins built into the system's design, such as headroom for load or tolerance for unexpected inputs.

Late projects have usually consumed their buffers. The schedule slack is gone, which is what it means to be late. The review capacity is saturated. The design margins have been trimmed in the pursuit of features. A late project is a system with no shock absorbers, in which every surprise is felt immediately.

Restoring buffers is a real intervention, and it can make a difference. The most important buffer for a project using agents is review capacity. If the humans who understand the design have time to review agent output carefully, the stock of undiscovered rework grows slowly. If they do not, it grows fast. A manager who protects the review capacity of their surgeons, by limiting the number of agents each one supervises, shielding them from meetings and keeping their queues within a work in progress limit, is rebuilding a buffer.

Schedule slack is harder to restore, because it usually requires either extending the deadline or reducing scope, both of which involve admitting that the original plan was not achievable. But a project that has no slack at all will be blown off course by every surprise, and a modest buffer, honestly acknowledged, is often worth more than any amount of additional capacity.

Structure of stocks and flows

Next is the physical structure of the system: how its stocks and flows are arranged. In a software project, this corresponds to the architecture of the system being built and the structure of the team building it.

Meadows placed this relatively low on her list because physical structure is usually slow and expensive to change. A city's road network cannot be redesigned overnight. A software system's architecture cannot be rebuilt in the middle of a crisis. The right time to get the structure right is at the beginning, when it is cheap.

But some structural changes are possible even in a late project, and some are powerful. Building a walking skeleton through an existing set of components, as described in the fifteenth chapter, is a structural change: it rearranges the flow of work from build-then-integrate to grow-continuously. Reorganising the team into surgical units, as described in the fourteenth chapter, changes the structure of communication. Defining interfaces precisely, as described in the thirteenth, changes the structure of dependencies between parts of the system. Each of these is harder than adding agents, and each changes how the system behaves in a way that adding agents cannot.

Delays

Meadows ranked the lengths of delays, relative to the rate of change in the system, above physical structure. Delays in feedback loops, she argued, are critical determinants of system behaviour, because they determine how quickly a system can respond to its own state.

The most important delays in a late project are the ones between doing work and learning whether it was right. The delay between writing code and discovering its defects. The delay between building a component and integrating it. The delay between building a feature and putting it in front of users. When these delays are long, the project accumulates hidden problems faster than it can discover them, and the manager acts on information that is out of date.

Shortening these delays is one of the most effective things a late project can do. Continuous integration shortens the delay between writing code and discovering integration failures. A walking skeleton shortens the delay between building components and seeing whether they work together. Early contact with users shortens the delay between building features and learning whether they are needed. End-to-end tests that run on every change shorten the delay between a change and the discovery that it broke something.

Meadows warned that shortening a delay is not always beneficial, and that changing the speed of a loop can destabilise a system. Her example was financial markets, where faster information transfer could produce wild swings. The relevant warning for agent-assisted projects is the one developed in the second chapter: agents shorten the delay in the hiring loop dramatically, making it possible to add workers in minutes rather than weeks. That is a delay that should arguably be made longer, not shorter, because it gives the manager time to see the effects of their last decision before making the next one. The delays that should be shortened are the ones that carry information about the true state of the project. The delays that should be preserved, or lengthened, are the ones that slow down decisions made on bad information.

Balancing loops

Next is the strength of balancing feedback loops: the corrective mechanisms that push a system back toward its goals. In a software project, these include the quality gates that stop bad changes from being merged, the review processes that catch design problems, and the work in progress limits that stop queues from growing without limit.

Late projects tend to weaken their balancing loops. Under schedule pressure, reviews get shallower, quality gates get bypassed, and limits get raised or ignored. Each of these weakens the system's ability to correct itself, and each lets problems accumulate that a stronger loop would have caught.

Strengthening balancing loops is a powerful intervention. Holding the line on test requirements, so that no change is merged without passing end-to-end tests. Keeping review queues within a work in progress limit, so that agents stop producing when humans cannot keep up. Making the checks that judge a change unmodifiable by that change, so that the loop cannot be short-circuited. Each of these strengthens the system's ability to correct itself, and each slows the project down in the short term in order to keep it from falling apart in the long term.

Reinforcing loops

Above balancing loops in Meadows's ranking is the gain around reinforcing loops: how strongly a self-amplifying process amplifies itself. A reinforcing loop with high gain grows explosively. One with low gain grows slowly, giving balancing loops time to take hold.

A late project is typically in the grip of two reinforcing loops, described in the second chapter. The staffing loop, in which lateness leads to more workers, which leads to more overhead, which leads to more lateness. And the pressure loop, in which lateness leads to shortcuts, which lead to defects, which lead to more lateness. Meadows argued that reducing the gain around a reinforcing loop is usually a more powerful intervention than strengthening the balancing loops that oppose it, because it attacks the source of the runaway behaviour rather than trying to contain it.

How can the gain be reduced? For the staffing loop, the most direct way is to stop adding workers in response to lateness, or at least to slow the rate at which they are added, so that the overhead of each addition can be seen and absorbed before the next. For the pressure loop, the most direct way is to reduce the pressure, by restoring some slack or reducing scope, so that the team is not forced into shortcuts. Both of these require the manager to resist the most natural response to a late project, which is to push harder. Meadows's observation that people tend to push leverage points in the wrong direction applies with particular force here.

Information flows

Higher still is the structure of information flows: who has access to what information. Meadows considered this a very high leverage point, because missing information is one of the most common causes of system malfunction, and adding or restoring information can change behaviour without changing anything else.

In a late project, the critical missing information is usually about the hidden stocks: undiscovered rework, unreviewed changes, unresolved design questions, and the gap between work as imagined and work as done at the sharp end. The HealthCare.gov warnings existed but did not reach anyone who could see them all together. The Mars Climate Orbiter navigation anomalies were discussed but not escalated through the channel designed for them. In both cases, the information existed in the system, but the flows that should have carried it to decision-makers were broken.

Restoring these flows is one of the most powerful interventions available. The progress ledger described in the ninth chapter is an information flow: it carries the state of the hidden stocks to the people managing the project. A single point where warnings converge, held by the architect, is an information flow. Monitoring that makes the system's real behaviour visible, as the HealthCare.gov command centre did, is an information flow. Regular contact between the humans directing agents and the sharp end is an information flow. Each of these changes what decision-makers know, and changing what they know changes what they do.

Rules

Above information flows are the rules of the system: its incentives, punishments and constraints. Rules determine what behaviour is possible and what behaviour is rewarded, and changing them can transform a system.

For a project using agents, the most important rules are the ones that constrain what agents and humans can do. A rule that no change can modify its own checks. A rule that agents have no access to production. A

rule that no change larger than a certain size is merged without additional review. A rule that features must fit within the architect's complexity budget. A rule that no feature work begins until the walking skeleton runs end to end. These rules shape the system's behaviour more powerfully than any number of instructions, because they determine what is possible rather than what is encouraged.

The eleventh chapter's distinction between rules in prose and rules in mechanism matters here. A rule that exists only in a document is weaker than one enforced by the environment. The most effective rules are the ones built into the structure of the system, so that violating them is not merely discouraged but impossible.

Self-organisation

Near the top of Meadows's list is the power of a system to add, change or evolve its own structure. Systems that can reorganise themselves in response to new conditions are resilient. Systems that cannot are brittle.

In a software project, self-organisation shows up as the team's capacity to change how it works when it learns something. The architect's playbook, refined continuously by feedback from reviews, incidents and surprises, is a mechanism of self-organisation: it lets the team change its own conventions and constraints as it learns. A team that holds regular retrospectives and acts on what it finds is self-organising. A team that can restructure itself, as CMS did after the HealthCare.gov launch, when the existing structure is failing, is self-organising.

Late projects tend to lose this capacity, because every hour spent reflecting and reorganising feels like an hour taken from delivery. But a team that cannot change how it works is a team that will keep failing in the same way. Protecting the capacity for self-organisation, by keeping the playbook current and making time to act on lessons, is a high-leverage investment.

Goals

Second from the top is the goal of the system. Meadows argued that changing a system's goal changes everything below it: its rules, its information flows, its loops and its parameters will all reorganise around the new goal.

The goal of most late projects is to deliver a fixed scope by a fixed date. This goal, held rigidly, generates much of the behaviour that makes the project later: the pressure loop, the reluctance to cut scope, the temptation to add workers rather than question the plan. A project whose goal is instead to deliver the most valuable working system possible by the date behaves very differently. It cuts scope readily. It builds the skeleton first. It tests against reality early. It protects quality, because a system that does not work has no value regardless of how many features it has.

Changing a project's goal requires the agreement of people outside the project, its sponsors and stakeholders, which is why it is hard. But it is often the single most powerful intervention available, and it is frequently the one that rescued projects actually made.

Paradigm

At the top of the list, below only the power to transcend paradigms, is the mindset out of which a system arises: the shared assumptions about what the system is and how it works.

The paradigm behind the proposal to add agents to a late project is that software is a quantity of output, produced by workers, and that more workers produce more output. This is the man-month in a new costume, and it is the paradigm that Brooks spent his book dismantling. The alternative paradigm, the one this book has argued for, is that software is a coherent design, realised through a process of discovery and integration, whose essential difficulty is in deciding what to build and making the parts fit together. In this paradigm, workers are valuable to the extent that they help realise the design, and harmful to the extent that they fragment it.

A manager who holds the first paradigm will reach for agents whenever the project is late. A manager who holds the second will reach for the design, the skeleton, the tests, the information flows and the goal. The second manager may use agents too, and use them well. But they will use them in service of a coherent design, rather than as a substitute for one.

Meadows's final leverage point, the power to transcend paradigms, is the recognition that no paradigm is the final truth, including this one. The tools will change. The economics of software will change. Some of what this book has argued will turn out to be wrong. The best defence is the attitude of wisdom that Weick described: holding our models loosely enough that the evidence can still change them.

CHAPTER 20

A Rescue in Practice

The preceding chapters have made an argument. This one tries to turn the argument into a sequence of actions that an engineering leader could take, starting on a Monday morning, when handed a project that is late and already full of agents.

It is offered as a composite, drawn from the principles developed in the book rather than from any single real project, and it should be read as a sketch rather than a recipe. Every late project is late in its own way, and the order and emphasis of the steps below will vary. But the shape of a rescue is surprisingly consistent, and it is worth seeing the pieces assembled in one place.

To keep it concrete, imagine a project of a kind that is increasingly common. A company is rebuilding the system its customers use to manage their accounts: billing, plans, usage, invoices, payment methods. The project has a team of twelve engineers and a product manager. Six months ago, it was due to launch in three months. Three months ago, it was due in six weeks. It is now late by an amount nobody will state with confidence. Over the past two months, in an effort to catch up, the team has dramatically increased its use of coding agents. Each engineer now runs several in parallel. The dashboard shows record numbers of merged changes. The project is still late.

You have just been asked to lead it.

Week one: stop and look

The first and hardest step is to stop pulling the lever that the organisation has been pulling. If the response to every slip has been more agents, the first thing to do is to stop adding them, and probably to reduce the number running, until you understand what they are producing.

This will feel wrong, and it will look wrong to the people who asked you to rescue the project. They will see the dashboard numbers fall. Explain, as plainly as you can, that the dashboard measures activity and that you need to know whether the activity is progress. Explain that adding workers to a late project has a long history of making it later, and that you need a week to find out whether that is happening here.

Then spend the week looking at the hidden stocks.

Count the unreviewed changes and note how old the oldest ones are. If agents have been producing faster than humans could review, there will be a backlog, and its size and age tell you how far behind the review process has fallen.

Look at the reopen rate. How many changes merged in the last month have since been reverted, fixed or substantially rewritten? A high rate means that much of the apparent progress was illusory, and that the stock of undiscovered rework has been filling.

List the open design questions. Sit with the senior engineers and ask them what they do not yet know about how the system should work. Write each question down with the date it was first raised. You will probably find more than anyone expected, some of them months old, and some of them affecting large amounts of code already written.

Find out which end-to-end capabilities actually work. Not which features are implemented, but which user journeys can be completed from start to finish in a realistic environment. Can a customer change their plan and see the change reflected in their next invoice? Can they add a payment method and have it charged correctly? It is common, on a late project, to find that many features are implemented and very few journeys work.

And talk to the sharp end. Find the people who support the current system, the ones who answer customers' questions and handle their complaints. Ask them what goes wrong today, and whether the new system addresses it. Find out whether anyone has shown the new system to a real customer.

By the end of the week, you should have a picture of the project as it is, rather than as the dashboard shows it. It will probably be sobering. That is the point.

Draw the loop

Before deciding what to do, gather the team and draw the system that produced the current situation. Use Meadows's vocabulary: stocks, flows, loops, delays.

The picture that emerges is usually some version of the one described in the second chapter. A visible stock of remaining work, shrinking more slowly than planned. An invisible stock of undiscovered rework, growing faster than anyone knew. A review process that became a bottleneck as agent output grew. Reinforcing loops from staffing and from schedule pressure. Long delays between writing code and discovering whether it integrates.

Drawing the loop together does two things. It gives the team a shared understanding of why the project is late that does not depend on blaming anyone, because the behaviour came from the structure. And it points directly at the interventions, because each element of the structure suggests a place where it could be changed.

Name the architect

If the project does not have a single person, or a very small group, holding the overall design, establish one. This is the most important structural decision of the rescue, for all the reasons laid out in Part Three.

The architect should be the engineer who best understands the system as a whole, who has the judgment to make design decisions and the standing to have them respected. They should be relieved of most other

duties. Their job, for the rest of the project, is to own the design: to resolve the open questions, to define the interfaces, to maintain the playbook, to review for coherence and to say no to features that do not fit.

Give the architect a copilot, someone who knows the design well enough to take over if necessary and who can act as an independent check. And identify, for each major area of the system, a surgeon: an engineer who holds the design for that area and who will supervise the agents working in it.

Find the thinnest working path

Next, identify the thinnest end-to-end path through the system that would deliver real value to customers, and make getting that path working the team's single top priority.

For the account management system, it might be something like this: a customer can log in, see their current plan and their most recent invoice, and update their payment method. That is a small fraction of the planned scope. But it exercises authentication, the account data model, the billing integration, the payment provider integration and the user interface, which is to say most of the system's major components and most of its interfaces.

Getting this path working, in a realistic environment with real integrations, will surface the integration problems the project has been deferring. Some of them will be serious. Some will require design changes that affect code already written. This is painful, and it is exactly what the project needs, because those problems were going to surface eventually, and surfacing them now, on a thin path, is far cheaper than surfacing them at launch across the whole system.

Once the thin path works, it becomes the walking skeleton. Every subsequent piece of work extends it. End-to-end tests are written for it and run on every change. The project has shifted from building pieces and hoping they fit to growing a working system one piece at a time.

Renegotiate the goal

With a realistic picture of the project and a working skeleton, go back to the sponsors and renegotiate the goal.

The original goal was probably a fixed scope by a fixed date, and the evidence now shows that it cannot be met. Present the evidence plainly: the capabilities that work, the open questions, the hidden rework. Then propose a different goal: the most valuable working subset of the system by a date that can be met, with the rest of the scope to follow in later releases.

This conversation is uncomfortable, and it is the highest-leverage intervention available. Meadows ranked goals second only to paradigms in her list of places to intervene, because changing the goal reorganises everything below it. A project whose goal is a valuable working subset will make different decisions about scope, quality and testing than one whose goal is the full original specification, and those decisions will make it far more likely to ship something that works.

Sponsors often accept this more readily than project leaders expect, especially when it is presented with evidence and with a credible plan. What they cannot accept is continued uncertainty, and a project that has been late repeatedly has usually exhausted their patience with optimistic dates. A smaller commitment that is kept is worth more to them than a larger one that slips again.

Turn interfaces into contracts

With the architect in place and the skeleton running, define the interfaces between the major components precisely, in forms that can be checked mechanically. Where components were built by different engineers or agents working from loose descriptions, there will be mismatches. Find them now, through contract tests or schema validation, and fix them.

This is the city plan from the thirteenth chapter. It is what makes it safe to work on different components in parallel, because any divergence at an interface will be caught as soon as it is introduced. It is also what makes it

safe to bring agents back in, because each agent working on a component will have a precise, checked definition of what its component must do at its boundaries.

Put the rules in mechanism

Review the environment the agents work in, and move the important rules from prose to mechanism.

Make sure agents cannot reach production systems or production data. Make sure the tests and checks that gate changes are held outside the changes, where agents cannot modify them. Cap how large a change may be before it needs a second reviewer. Set a work in progress limit on each surgeon's review queue, so that agents pause when the queue is full. Require that every change pass the skeleton's end-to-end tests before it is merged.

Each of these will slow the agents down in some way. Each will also make their output more trustworthy, so that the humans reviewing it can spend their attention on design and fit rather than on catching basic errors.

Write the playbook

While the skeleton is being built and the interfaces defined, the architect should be writing, or rewriting, the playbook.

On a late project, the playbook usually either does not exist or is badly out of date. The conventions the team follows have drifted. Different areas of the code follow different patterns, often because different agents, given different context, made different choices. The architect's job is to decide what the conventions should be, write them down with their rationale, and make sure every agent session and every human engineer has access to them.

This is also the moment to start the feedback loop that keeps the playbook current. Every review comment that reflects a design principle, every surprise in agent output, every incident during testing, should be

considered as a possible playbook update. Within a few weeks, the playbook will have absorbed a great deal of the team's design knowledge, and agent output will have become noticeably more consistent.

Bring the agents back, in the right places

Now, and only now, bring the agents back to full use. But aim them deliberately.

Agents are most valuable on accidental work in areas where the design is settled. Once the skeleton runs and the interfaces are defined, much of the remaining work in a project like the account management system fits this description: implementing the remaining screens within established patterns, writing tests for well-understood behaviour, migrating data according to defined rules, generating documentation from code. Assign agents to these, under surgeons who can absorb their output.

Agents are also valuable for simplification. Direct some of their capacity to removing dead code, consolidating duplicated logic and bringing inconsistent areas into line with the playbook. On a project that has had heavy agent use without a strong architect, there will be a great deal of this work, and doing it reduces the stock of accidental complexity that slows every future change.

Agents are least valuable, and potentially harmful, in areas where the design is still unsettled. Keep them away from those areas until the architect and the relevant surgeon have resolved the open questions. Code written before the design is settled will have to be rewritten once it is.

Keep looking

Finally, establish the habits that will keep the project honest for the rest of its life.

Maintain the progress ledger from the ninth chapter: demonstrated capabilities, open design questions, the age of the review queue, the reopen rate, and an honest sentence about the biggest risk. Review it weekly in a status meeting where information is shared without immediate action, so that the people maintaining it have no reason to shade it.

Keep a lightweight record of small surprises, in the spirit of JPL's Incident, Surprise, Anomaly process, and have the architect review it regularly for patterns.

Keep the engineers in contact with the sharp end. Put the thin path in front of real customers as early as possible, and keep putting each new capability in front of them as it is added. Keep the engineers on call for whatever parts of the system are running.

And keep resisting the lever. There will be moments, later in the project, when it slips again and someone proposes adding more agents. Sometimes that will be the right call, if the slip is in accidental work with a settled design and the review capacity exists to absorb more output. Often it will not be. The question to ask, every time, is the one this book has tried to equip you to answer: why is the project late, and is that the kind of lateness that more workers can fix?

Signs that it is working

A rescue of this kind takes weeks to show results, and the early weeks can look worse than what came before, because the dashboard numbers fall while the team looks at the hidden stocks and builds the skeleton. It helps to know what signs to look for, so that the rescue is judged on the right evidence.

The first sign is that the open design questions start to close. In the first week, the list will probably grow, as the team surfaces questions that had been deferred or unspoken. Within a few weeks, if the architect is doing their job, it should begin to shrink. Questions should be resolved faster than new ones appear, and the oldest ones should disappear first.

The second sign is that the number of working end-to-end capabilities rises steadily. The skeleton's first path is the hardest. Subsequent capabilities, built on top of it, should come more easily, and the ledger should show them being demonstrated at a regular pace.

The third sign is that the review queue stabilises. Its size should stay within the work in progress limit, and its oldest item should be days old, not weeks. If it keeps growing, the agents are still outpacing the surgeons, and either the agents should be reduced or the surgeons' load should be lightened.

The fourth sign is that the reopen rate falls. As the playbook takes hold, the interfaces are checked and the skeleton's end-to-end tests catch integration failures early, fewer merged changes should need to be reverted or rewritten. A falling reopen rate means the stock of undiscovered rework is filling more slowly, which means the project's visible progress is becoming more trustworthy.

The fifth sign is qualitative, and in some ways the most important. The team's conversations change. Instead of discussing how to go faster, they discuss what to build next and how it should fit. Instead of surprises at integration, there are small, manageable surprises every day. Instead of a vague sense that the project is somewhere near done, there is a shared and specific picture of what works, what does not, and what remains. That shift, from motion to understanding, is what it feels like when a project stops getting later.

What the rescue does not promise

It is important to be honest about what this kind of rescue achieves. It does not make a late project early. It does not recover the time that was lost. It does not make essential complexity disappear, and it does not guarantee that the system will be finished by any particular date.

What it does is change the trajectory. A project in the grip of a regenerative disaster gets later with every intervention, because each intervention feeds the loops that are making it late. A project that has been restructured around an architect, a skeleton, honest measurement and a realistic goal stops getting later. It starts to converge. The remaining work becomes visible and finite, the integration problems surface one at a time, and the team can see the end. That is not a miracle. It is what it looks like when a system's structure is changed so that its behaviour changes.

CONCLUSION

What Can Rescue a Late Project

Frederick Brooks died in November 2022, at the age of ninety-one. He lived long enough to see the first wave of AI coding assistants, though not the agents that followed. It is tempting to wonder what he would have made of them, and the honest answer is that we cannot know. But his book gives us a reasonable guess about the questions he would have asked.

He would have asked what the agents actually do, and whether it is essential or accidental work. He would have asked what they cost, not in money but in the time of the people who understand the system. He would have asked who holds the design, and how it is communicated to every worker. He would have asked how the team knows where it really stands, and whether its milestones have sharp edges. And he would have asked, with the wry humility of someone who managed OS/360, whether the organisation proposing to add them had looked honestly at why its project was late.

This book has tried to answer those questions, with help from two other thinkers who would have recognised them. Donella Meadows gave us a way to see the late project as a system, with stocks that remember its history, flows that fill and drain them, loops that drive its behaviour and delays that hide that behaviour from the people inside it. Karl Weick and Kathleen Sutcliffe gave us a way to understand why organisations fail to see what is happening to them, and what the organisations that do see it have in common.

The argument can be stated briefly.

Why agents cannot rescue a late project

Brooks's Law holds that putting more people onto a software project that is already late will delay it further. Brooks gave three reasons: new workers must be trained, the work must be repartitioned to give them something to do, and the communication needed to keep everyone coordinated grows faster than the number of workers. Each of these is a cost of understanding, borne mostly by the people who already understand the system.

AI agents change the form of these costs without removing them. Training becomes context provision, paid not once per worker but, to some degree, once per task. Repartitioning becomes specification, done at finer grain and greater volume. Communication becomes review, in which every line an agent writes must be read, understood and judged by someone who holds the design. All three costs concentrate on the same scarce people, and all three grow as more agents are added.

Seen through Meadows's eyes, Brooks's Law is a reinforcing feedback loop wrapped around an invisible stock. Lateness prompts the addition of workers, the workers add overhead, the overhead slows progress, and the slower progress prompts more workers. Meanwhile, a stock of undiscovered rework fills silently, hidden from the burndown chart by the delay between making an error and finding it. Agents make the loop spin faster, because workers can now be added in minutes rather than weeks, and they make the invisible stock fill faster, because they produce code faster than humans can review it carefully.

Brooks's distinction between essential and accidental complexity explains why. Agents are a powerful attack on accidental complexity: the work of representing a settled idea in code. But late projects are rarely late because of accidental complexity. They are late because the requirements were not understood, the design did not fit, the components did not integrate, or the external constraints were not grasped. These are essential difficulties, and agents do not solve them. Set to work before the essential problems are solved, agents produce code that embodies the wrong concepts, quickly and fluently, adding to the pile that must be revised.

And Weick and Sutcliffe explain why organisations keep reaching for the wrong remedy. Lateness accumulates through small slips, each of which is easy to explain away. Organisations tell themselves reassuring stories about early warnings, attach comfortable labels to small failures, lose touch with the sharp end and measure activity rather than progress. By the time the evidence is undeniable, they are primed for a remedy that does not require anyone to admit that the plan was wrong. Adding workers is that kind of remedy, and agents make it cheaper and faster than ever.

What can

The other half of the argument is more hopeful, and it matters more.

Late projects can be rescued, and projects can be kept from becoming late, by the same things that have always worked, applied with more discipline because cheap workers have removed some of the natural brakes.

The first is conceptual integrity: a single coherent design, held by an architect or a very small group, empowered to defend it. Brooks ranked it above every other property of a system, and its value rises as implementation gets cheaper, since every cheap pair of hands is another force pulling the design in its own direction. A team in which each engineer steers a fleet of agents has, in effect, a dozen architects, and without one design above them the code will show it.

The second is a living playbook and a precise plan of interfaces, which carry the design from the architect to every worker. Agents cannot absorb a team's culture through conversation. Everything they know about the design, they learn from what is written down. The playbook must capture not just rules but reasons, so that workers can extend the design to cases it does not cover. The interfaces must be defined in forms that can be checked mechanically, so that independent workers cannot diverge without being caught. And the most important rules must be enforced by the environment rather than by instruction, because a rule in prose holds only as long as it is followed.

The third is a walking skeleton: a thin, working version of the whole system, built first and grown continuously. It drags the integration problems into daylight at the start of the project, when they are cheapest to fix. It gives agents a concrete structure to work within and end-to-end tests to work against. It narrows the cone of uncertainty quickly and provides the only honest measure of progress there is: what actually works, end to end, in a realistic environment.

The fourth is testing that comes before launch, and checks that the change cannot modify. Agents can write excellent tests and can also make tests pass for the wrong reasons. The defence is layered: the agent's own verification, checks held outside the change, end-to-end tests against the running skeleton, human review focused where the other layers are weakest, staged deployment that limits the blast radius of any change, and runtime controls that stop a misbehaving system before the damage grows.

The fifth is the habit, built into the organisation, of hearing bad news before it becomes expensive. That means preoccupation with small failures, including the small failures of agents that are easy to dismiss as quirks. It means reluctance to accept labels, hallucination and bad prompt as much as operator error, that close inquiry before it has found the cause. It means sensitivity to the sharp end, maintained deliberately by the humans on a team, because the agents contribute nothing to it. It means honest measures of progress, and status meetings in which bad news can be shared without triggering panic. And it means building resilience in advance, through rotation, practice and the attitude of wisdom, so that when the crisis comes, as it eventually will, the organisation can improvise its way through.

None of these is new. All of them would have been familiar to Brooks, to Meadows and to Weick and Sutcliffe. What is new is that agents make all of them harder to practise and more costly to neglect. A team with cheap workers and no architect will produce an incoherent system faster than ever. A team with cheap workers and no skeleton will reach the ninety percent mark faster than ever and then stall. A team with cheap

workers and weak testing will launch boats full of holes at unprecedented speed. And a team with cheap workers and no capacity to hear bad news will drive its project into the ground with a dashboard full of green.

Cheap hands, costly coherence

If there is a single idea at the heart of this book, it is this: when workers become cheap, coherence becomes the scarce resource. For most of the history of software, the limiting factor on what a team could build was the number of skilled hands available to build it. That constraint is loosening. What remains is the constraint that Brooks identified as the most important all along: the number of minds that can hold a design together, and the means by which they communicate it to everyone else.

Organisations that understand this will invest in their architects and surgeons, protect their time, give them copilots and keep them in contact with the sharp end. They will build skeletons first and features second. They will write playbooks and define interfaces before adding workers. They will put their most important rules into mechanism. They will measure what has been demonstrated rather than what has been done. And they will use agents generously, in the places where agents help, under the supervision of people who can absorb what they produce.

Organisations that do not understand it will treat agents as a substitute for all of this. They will add them whenever a project is late, measure their activity and call it progress, and discover, at integration or at launch, that they have built a large, incoherent system faster than anyone thought possible.

Brooks ended the essay on the mythical man-month with a warning about the man-month itself, describing it as a dangerous and deceptive myth, because it implies that people and months are interchangeable. The agent-month is the same myth in a new form. It is more seductive, because agents really are more interchangeable than people, and because their cost really is low. But the work they are applied to has not changed. It still consists, at its heart, of deciding what to build and making the pieces fit together. That work does not divide among many hands, human or

machine. It belongs to the few minds that can hold the whole, and the best thing an organisation can do for a late project is to give those minds the time, the authority and the information to do it.

The chart in the meeting will still have a red band across the top. Somebody will still propose spinning up twenty agents. The answer, now as in 1975, is to ask first why the project is late. If the answer is that there is a large body of well-understood work and not enough hands, agents may help, and they should be used. If the answer is anything else, and it usually is, the project needs something agents cannot supply: a coherent design, a working skeleton, honest testing and an organisation willing to hear what the evidence is saying. Those are what rescue late projects. They always have been.

Notes on Sources

This book draws on three primary works and on a body of published investigations, audits, regulatory findings and research. The notes below identify, chapter by chapter, where the facts and quotations come from. Quotations from Brooks, Meadows, Weick and Sutcliffe are taken from the editions and documents listed in the References; elsewhere their arguments are paraphrased. Figures from investigations and audits are reported as the issuing body stated them. Where the public record is contested or incomplete, the text says so or avoids the point.

Introduction

Brooks's Law is stated in the second essay of The Mythical Man-Month (1975; anniversary edition 1995). The METR randomised controlled trial is Becker and colleagues (2025): sixteen experienced developers, 246 tasks, a measured 19 percent increase in completion time with AI tools allowed, against a forecast 24 percent reduction and a post hoc estimate of 20 percent. METR has since published continuation results for newer tools, and its authors cautioned against generalising the 2025 finding. The DORA figures (a 1.5 percent reduction in delivery throughput and a 7.2 percent reduction in delivery stability for every 25 percent increase in AI adoption) and the small-batch hypothesis are from the 2024 Accelerate State of DevOps report. The 18 documented warnings and the phrase about desensitisation to bad news are from the HHS Office of Inspector General's 2016 case study of HealthCare.gov. The CrowdStrike 21-versus-20 input mismatch is from the company's external technical root cause analysis of August 2024. The walking skeleton is Alistair Cockburn's term; Gall's law is from John Gall's Systemantics.

Chapter 1

OS/360 history, Brooks's description of its strengths and its lateness, memory use and cost overruns, and his acceptance of responsibility for the 1964 to 1965 design flaws are from the preface to The Mythical Man-Month. The child-bearing illustration, the n(n-1)/2 intercommunication argument, the scheduling rule of thumb (one third planning, one sixth coding, one quarter component and early system test, one quarter system test), the worked example of the regenerative schedule disaster, and the statement of the law with its qualifier are from the essay The Mythical Man-Month. The

observation that all programmers are optimists is from the same essay. Abdel-Hamid and Madnick's system dynamics findings, including that adding people made medium-sized application projects more costly but not always later, are from their MIT working papers and their 1991 book Software Project Dynamics. Brooks left IBM in 1965 to found the computer science department at the University of North Carolina at Chapel Hill.

Chapter 2

Meadows's bathtub, her definition of a stock as a physical memory of the history of the system, her discussion of delays as determinants of oscillation, and her warning about reducing delays in financial markets are from Thinking in Systems (2008) and the essay Leverage Points (1999). The system trap she called shifting the burden to the intervenor is from Thinking in Systems. The rework cycle is a structure long used by system dynamics practitioners modelling large projects; no specific study is relied on here. The DORA interpretation is from the 2024 report.

Chapter 3

The SWE-bench Verified trend (from about 40 percent to more than 80 percent in one year) is as stated by Anthropic in its January 2026 article on evaluating AI agents. Amdahl's law is from Gene Amdahl's 1967 AFIPS conference paper; Amdahl was one of the principal architects of System/360. Little's law is from John D. C. Little's 1961 proof. The theory of constraints is popularised in Goldratt and Cox's The Goal (1984). The review practices attributed to the Claude Code team (using persistent test failures after three rounds to bring in a human, preferring one tuned reviewer over stacked reviewers, keeping checks out of reach of the changes they judge) are from an article on Anthropic's developer blog in November 2026. Weick and Sutcliffe's view of diversity as an instrument of perception is from Managing the Unexpected.

Chapter 4

No Silver Bullet was presented at the IFIP congress in 1986 and reprinted in IEEE Computer in 1987, then as chapter 16 of the anniversary edition. The quotations defining essential and accidental tasks and denying any single order-of-magnitude improvement, and the statement that deciding precisely what to build is the hardest single part of building a software system, are from that paper, as are the four essential properties (complexity, conformity, changeability, invisibility) and the four proposed attacks on the essence. The Ariane 5 Flight 501 account follows the inquiry board report chaired by Jacques-Louis Lions (July 1996).

Chapter 5

Hatching a Catastrophe is the fourteenth essay in The Mythical Man-Month and the source of the one-day-at-a-time observation, the advice on sharp milestones, and the distinction between status review and problem-action meetings. Weick and Sutcliffe's

account of weak signals and of recognition as a source of concern rather than comfort is from Managing the Unexpected (second edition, 2007). The HealthCare.gov material, including the five-times-expected traffic, the 18 documented warnings between July 2011 and July 2013, the diffuse leadership, the absence of contingency plans and the lack of serious discussion of delay, is from the HHS OIG case study (2016) and related testimony. Normalisation of deviance is Diane Vaughan's term from The Challenger Launch Decision (1996). Watermelon reporting is practitioner slang and is not attributed to any source.

Chapter 6

All Mars Climate Orbiter facts are from the NASA Mishap Investigation Board's Phase I report (November 1999) and its report on project management (2000), including the 170-kilometre trajectory error, the SM_FORCES units error and the factor of 4.45, the ten to fourteen times more frequent desaturation events and their cause, the four months of unusable files, the informal reporting of anomalies, the non-use of the Incident, Surprise, Anomaly process, and the account of TCM-5. The description of NASA officials' comments at the November 1999 press conference follows contemporaneous reporting in IEEE Spectrum. The premortem is Gary Klein's technique, described in Harvard Business Review (2007).

Chapter 7

The Therac-25 account follows Leveson and Turner (1993), including the eleven machines installed, the six known overdose accidents between June 1985 and January 1987, the removal of hardware interlocks, the early attribution of one accident to a microswitch failure, the cryptic error messages, the race condition affected by fast data entry, and the authors' quoted conclusions about human error, software error and the inevitability of another bug. Weick and Sutcliffe's sentence about less simplification allowing you to see more is from Managing the Unexpected.

Chapter 8

The NTSB findings, including the quoted comments of Chairman Robert Sumwalt and the content of recommendations A-19-10 to A-19-16, are from the board's September 2019 safety recommendation report and press release. Statements about MCAS activation on erroneous angle of attack data reflect widely published findings of the accident investigations; the text deliberately does not summarise contested causal questions. National Programme for IT facts are from the National Audit Office reports of May 2011 and May 2019; the six lessons are from the Wachter review (2016) as cited by the NAO. The onboarding example is a hypothetical illustration.

Chapter 9

The ninety-ninety rule is attributed to Tom Cargill of Bell Labs and was published in Jon Bentley's Programming Pearls column (1985). Brooks's remarks on fuzzy milestones and ninety percent complete coding are from Hatching a Catastrophe and The Mythical Man-Month. The cone of uncertainty follows Barry Boehm's Software Engineering Economics (1981); the name was popularised by Steve McConnell. Goodhart's law is from Charles Goodhart (1975), in the form later given by Marilyn Strathern (1997). DORA's rework rate is from the 2024 report. The progress ledger is the author's proposal.

Chapter 10

The Reims cathedral passage, the claim that conceptual integrity is the most important consideration in system design, the function-to-complexity ratio, the OS/360 and PDP-10 comparison, the separation of architecture from implementation, the clock example and the admission that the method failed through lack of application on OS/360 are from the fourth essay of The Mythical Man-Month, Aristocracy, Democracy, and System Design. Conway's law is from Melvin Conway's 1968 Datamation article. HealthCare.gov leadership and contracting findings are from the HHS OIG case study.

Chapter 11

Brooks's views on manuals and formal definitions are from the essay Passing the Word. Architecture decision records follow Michael Nygard's 2011 article. The Replit incident is reconstructed from Jason Lemkin's public posts as reported by The Register (21 July 2025), Fast Company (22 July 2025) and TechTarget (22 July 2025); the figures for deleted records are those the agent itself reported in screenshots Lemkin posted, as cited by Fast Company. Replit's response is as reported in those outlets. Single-loop and double-loop learning are Chris Argyris's terms.

Chapter 12

The second-system effect and Brooks's remedies are from the fifth essay of The Mythical Man-Month. Lehman's observation on increasing complexity is from his laws of software evolution. Scope growth on the NHS programme, HealthCare.gov and the Denver baggage system is from the Wachter review as cited by the NAO, the HHS OIG case study, and GAO report RCED-95-241FS respectively. You aren't gonna need it is a principle of extreme programming.

Chapter 13

The documentary hypothesis is the tenth essay of The Mythical Man-Month. The CrowdStrike account, including sensor version 7.11, the 21 defined input fields and the 20 supplied, wildcard matching in testing, the 19 July 2024 template instances, the

content validator's logic error and the remediations, is from CrowdStrike's external technical root cause analysis (6 August 2024). The estimate of about 8.5 million affected Windows devices is Microsoft's, published 20 July 2024. The Mars Polar Lander recommendation for an interface specification audit is from the MCO Phase I report.

Chapter 14

Mills's surgical team and its ten roles are as described in the third essay of The Mythical Man-Month. The study of programmer variation is Sackman, Erikson and Grant (1968), cited by Brooks. Brooks's proposals for growing great designers are from No Silver Bullet. Cognitive load as a design consideration for teams follows Skelton and Pais, Team Topologies (2019).

Chapter 15

The Denver baggage system account, including the city's initial decision against an airport-wide system, the two-year build and test window, the four opening delays from October 1993 to February 1995, the cost growth from about $195 million to more than $290 million, the February 1994 test failures and specific malfunctions, Logplan's recommendations and the capacity reduction from 65 to 30 bags per minute per line, is from GAO reports RCED-95-35BR (1994) and RCED-95-241FS (1995). Gall's law is from Systemantics. Plan to throw one away is the eleventh essay of The Mythical Man-Month; Brooks's later preference for incremental development and his description of growing software from a running system with dummy subprograms are from No Silver Bullet and the anniversary edition's added chapters. The walking skeleton is described in Cockburn's Crystal Clear (2004).

Chapter 16

All Knight Capital facts are from the SEC's order and press release of 16 October 2013, including the 2005 code change, the July 2012 deployment, the more than four million orders sent in forty-five minutes while trying to fill 212 customer orders, the more than 397 million shares traded, the loss of more than $460 million, the 97 automated emails and the list of control failures. The Swiss cheese model is James Reason's. Brooks's system-testing advice is from The Whole and the Parts. The end-to-end testing guidance attributed to the Claude Code team is from the November 2026 article cited for chapter 3. Test-driven development follows Kent Beck; Hypothesis and fast-check are named in that article as property-testing libraries.

Chapter 17

CrowdStrike's commitments to staggered deployment with canary stages and greater customer control are from its preliminary post-incident review (July 2024). Meadows's discussion of buffers is from the Leverage Points essay. DORA findings on batch size are

from successive Accelerate State of DevOps reports. Error budgets are described in Site Reliability Engineering (2016). One-way and two-way doors are from Jeff Bezos's 2015 letter to Amazon shareholders.

Chapter 18

Weick and Sutcliffe's principles of commitment to resilience and deference to expertise are from Managing the Unexpected. The Mann Gulch account follows Norman Maclean's Young Men and Fire (1992) and Weick's 1993 article in Administrative Science Quarterly, which also sets out the four sources of resilience. HealthCare.gov repair facts, including staff arrival within three weeks, the Exchange Operations Center, integrated technical and policy staff, monitoring tools and substantial improvement within two months, are from the HHS OIG case study. The roles of Jeffrey Zients and Mikey Dickerson are a matter of wide public record.

Chapter 19

The list of leverage points, their order, Meadows's account of the meeting that prompted it, her qualification that the list is tentative and its order slithery, and Jay Forrester's observation that people push leverage points in the wrong direction are from Leverage Points: Places to Intervene in a System (1999).

Chapter 20

The account management project is a composite illustration, not a description of any real organisation.

Conclusion

Brooks died on 17 November 2022. His description of the man-month as a dangerous and deceptive myth is from the essay The Mythical Man-Month.

References

Abdel-Hamid, Tarek, and Stuart E. Madnick. Software Project Dynamics: An Integrated Approach. Prentice Hall, 1991.

Amdahl, Gene M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." AFIPS Spring Joint Computer Conference, 1967.

Anthropic. "Demystifying Evals for AI Agents." January 2026.

Becker, Joel, et al. "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity." METR, July 2025. arXiv:2507.09089.

Beck, Kent. Test-Driven Development: By Example. Addison-Wesley, 2002.

Bentley, Jon. "Programming Pearls." Communications of the ACM, 1985.

Bezos, Jeffrey P. Letter to Amazon Shareholders for 2015. Amazon.com, 2016.

Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, eds. Site Reliability Engineering. O'Reilly, 2016.

Boehm, Barry W. Software Engineering Economics. Prentice Hall, 1981.

Brooks, Frederick P., Jr. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1975. Anniversary edition, 1995.

Brooks, Frederick P., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering." Information Processing 86, IFIP, 1986; reprinted in IEEE Computer 20, no. 4 (1987).

Cockburn, Alistair. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley, 2004.

Conway, Melvin E. "How Do Committees Invent?" Datamation 14, no. 4 (1968).

CrowdStrike. Preliminary Post Incident Review: Content Configuration Update Impacting the Falcon Sensor and the Windows Operating System. July 2024.

CrowdStrike. External Technical Root Cause Analysis: Channel File 291. 6 August 2024.

DORA (Google Cloud). Accelerate State of DevOps Report 2024. 2024.

Gall, John. Systemantics. First published 1975 as General Systemantics; later editions published as The Systems Bible.

Goldratt, Eliyahu M., and Jeff Cox. The Goal. North River Press, 1984.

Klein, Gary. “Performing a Project Premortem.” Harvard Business Review, September 2007.

Lehman, M. M. “Programs, Life Cycles, and Laws of Software Evolution.” Proceedings of the IEEE 68, no. 9 (1980).

Leveson, Nancy G., and Clark S. Turner. “An Investigation of the Therac-25 Accidents.” IEEE Computer 26, no. 7 (1993): 18 to 41.

Lions, J. L., et al. Ariane 5 Flight 501 Failure: Report by the Inquiry Board. European Space Agency and CNES, July 1996.

Little, John D. C. “A Proof for the Queuing Formula: L = λW.” Operations Research 9, no. 3 (1961).

Maclean, Norman. Young Men and Fire. University of Chicago Press, 1992.

Meadows, Donella H. Leverage Points: Places to Intervene in a System. Sustainability Institute, 1999.

Meadows, Donella H. Thinking in Systems: A Primer. Edited by Diana Wright. Chelsea Green, 2008.

Microsoft (David Weston). “Helping Our Customers through the CrowdStrike Outage.” Official Microsoft Blog, 20 July 2024.

NASA Mars Climate Orbiter Mishap Investigation Board. Phase I Report. 10 November 1999.

NASA Mars Climate Orbiter Mishap Investigation Board. Report on Project Management in NASA. 2000.

National Audit Office. The National Programme for IT in the NHS: An Update on the Delivery of Detailed Care Records Systems. May 2011.

National Audit Office. Digital Transformation in the NHS. May 2019.

National Transportation Safety Board. Assumptions Used in the Safety Assessment Process and the Effects of Multiple Alerts and Indications on Pilot Performance. Safety Recommendation Report ASR-19-01, September 2019.

Nygard, Michael. “Documenting Architecture Decisions.” 2011.

Osmani, Addy. “How the Claude Code Team Thinks About Verification.” Anthropic developer blog, November 2026.

Reason, James. Managing the Risks of Organizational Accidents. Ashgate, 1997.

Sackman, H., W. J. Erikson, and E. E. Grant. “Exploratory Experimental Studies Comparing Online and Offline Programming Performance.” Communications of the ACM 11, no. 1 (1968).

Securities and Exchange Commission. In the Matter of Knight Capital Americas LLC. Order, 16 October 2013; Press Release 2013-222.

Skelton, Matthew, and Manuel Pais. Team Topologies. IT Revolution, 2019.

Strathern, Marilyn. "'Improving Ratings': Audit in the British University System." European Review 5, no. 3 (1997).

US Department of Health and Human Services, Office of Inspector General. Health-Care.gov: Case Study of CMS Management of the Federal Marketplace. February 2016.

US General Accounting Office. New Denver Airport: Impact of the Delayed Baggage System. RCED-95-35BR, October 1994.

US General Accounting Office. Denver International Airport: Baggage Handling, Contracting, and Other Issues. RCED-95-241FS, 1995.

Vaughan, Diane. The Challenger Launch Decision. University of Chicago Press, 1996.

Wachter, Robert, chair. Making IT Work: Harnessing the Power of Health Information Technology to Improve Care in England. National Advisory Group on Health Information Technology in England, 2016.

Weick, Karl E. "The Collapse of Sensemaking in Organizations: The Mann Gulch Disaster." Administrative Science Quarterly 38, no. 4 (1993).

Weick, Karl E., and Kathleen M. Sutcliffe. Managing the Unexpected. Jossey-Bass, 2001; second edition 2007; third edition, Wiley, 2015.

Press coverage of the Replit incident: Simon Sharwood, The Register, 21 July 2025; Mark Sullivan, Fast Company, 22 July 2025; Beth Pariseau, TechTarget, 22 July 2025.

Glossary

Accidental complexity. In Brooks's terms, the difficulties that attend the production of software with the tools of the day, such as expressing a design in a programming language, but that are not inherent in the problem. Better tools, including AI agents, can reduce it.

Agent-month. This book's name for the belief that compute spent on AI agents is a fungible unit of progress, so that twice the agents yield twice the result. It is the man-month fallacy in a new form.

Amdahl's law. The principle, from Gene Amdahl's 1967 paper, that the speedup available from parallel work is limited by the fraction of the work that must be done serially.

Architect. The person, or very small group, who holds a system's overall design, defines its concepts and interfaces, and is responsible for its conceptual integrity.

Balancing loop. A feedback structure that pushes a stock toward a goal, such as a thermostat. Also called a negative or correcting loop.

Blast radius. The extent of the harm a change, or an agent's action, can cause if it goes wrong.

Blunt end. The places where the conditions for work are set, such as planning, budgeting, design review and regulation, as opposed to the sharp end where the work meets the world.

Brooks's Law. Frederick Brooks's statement, which he called an outrageous oversimplification, that adding manpower to a late software project makes it later.

Buffer. A stock that absorbs fluctuations so that a sudden change in one flow does not immediately propagate through a system. Schedule slack, spare review capacity and staged deployment are buffers.

Canary deployment. Releasing a change first to a small subset of machines or users, observing its effect, and expanding only if it behaves well.

Conceptual integrity. The property of a system whose parts all reflect one coherent set of design ideas. Brooks considered it the most important consideration in system design.

Cone of uncertainty. The pattern in which estimates of a project's size and duration are most uncertain at the start and narrow as decisions are made and unknowns resolved.

Contract test. A test that checks that a component honours the expectations its consumers have of its interface, so that a change on either side of the interface cannot silently break the other.

Conway's law. Melvin Conway's 1968 observation that organisations which design systems produce designs that copy their own communication structures.

Copilot. In the surgical team, the surgeon's alter ego: a human who shares in the design, knows the code, provides an independent check and can take over if necessary.

Deference to expertise. Weick and Sutcliffe's fifth principle: in a crisis, decisions migrate to the people with the most relevant knowledge of the problem, regardless of rank.

Delay. The time between a cause and its visible effect in a system. Long delays in feedback cause overshoot, oscillation and decisions made on outdated information.

End-to-end test. A test that exercises a whole user-visible path through a running system, from input to output, rather than a single component in isolation.

Error budget. An explicit allowance for unreliability, used to decide how much risk a team may take with changes before it must slow down and invest in reliability.

Essential complexity. In Brooks's terms, the difficulty inherent in the conceptual structure of a software system: deciding what it must do, specifying it, and making its concepts fit together. Tools cannot remove it.

Feature flag. A configuration switch that turns a piece of functionality on or off without redeploying code, used to limit blast radius and allow gradual rollout.

Gall's law. John Gall's observation that a complex system that works is invariably found to have evolved from a simple system that worked.

Goodhart's law. The principle, in Marilyn Strathern's phrasing, that when a measure becomes a target it ceases to be a good measure.

High reliability organisation. An organisation, such as an aircraft carrier flight deck or a nuclear plant, that operates in hazardous conditions with far fewer serious failures than might be expected, studied by Weick and Sutcliffe.

Leverage point. In Meadows's terms, a place in a system where an intervention can change its behaviour. She ranked twelve, from parameters (weakest) to the power to transcend paradigms (strongest).

Little's law. The queueing result that the average number of items in a stable queue equals their arrival rate multiplied by the average time each spends in the queue.

Man-month. A unit of effort equal to one person working for one month. Brooks called it a dangerous and deceptive myth when used to measure the size of a job, because people and months are interchangeable only for work that needs no communication.

Mindful organising. Weick and Sutcliffe's term for the collective habits of attention that let high reliability organisations notice and contain the unexpected.

Mutation testing. Testing the tests by making small deliberate changes to the code and checking whether the test suite fails.

Normalisation of deviance. Diane Vaughan's term for the process by which repeated anomalies that do not cause disaster come to be redefined as acceptable.

Playbook. The living body of written design guidance, including rules, conventions, fragile areas and their rationale, that transmits a system's design to every human and agent who works on it, and that is refined by feedback.

Preoccupation with failure. Weick and Sutcliffe's first principle: treating every lapse or near miss as a possible symptom of a deeper problem in the system.

Progress ledger. This book's proposed record of a project's state in terms of demonstrated end-to-end capabilities, open design questions, review queue age, reopen rate and the leading risk.

Property-based testing. Testing that states a rule that should always hold and checks it against many generated inputs, rather than a few hand-picked examples.

Reinforcing loop. A feedback structure that amplifies whatever change is already happening. Brooks's regenerative schedule disaster is one.

Reluctance to simplify. Weick and Sutcliffe's second principle: resisting comfortable labels and explanations until they have been tested against the evidence.

Rework cycle. The structure in which work done incorrectly enters a hidden stock of undiscovered rework and later returns to the visible backlog when the errors are found.

Rule in mechanism. A constraint enforced by the structure of the environment, such as an agent having no credentials for production. Contrasted with a **rule in prose**, which is an instruction that holds only as long as it is followed.

Second-system effect. Brooks's warning that a designer's second system is the most dangerous, because it tends to be over-designed with every idea held back from the first.

Sensitivity to operations. Weick and Sutcliffe's third principle: close and continuous attention to the front line, where the real work is done.

Sharp end. The place where work meets the world: users, production systems, operators and on-call engineers.

Stock. In Meadows's terms, an accumulation in a system, such as a backlog, a body of code or a queue of unreviewed changes, which changes only through its inflows and outflows.

Surgical team. Harlan Mills's proposal, described by Brooks, for a team organised around one chief programmer supported by specialists, so that the design comes from one mind while the work is shared.

Walking skeleton. Alistair Cockburn's term for a tiny implementation of a system that performs a small end-to-end function and links together the main architectural components.

Work in progress limit. An explicit cap on the number of items allowed in a stage of a process at once, used to stop queues building in front of bottlenecks.