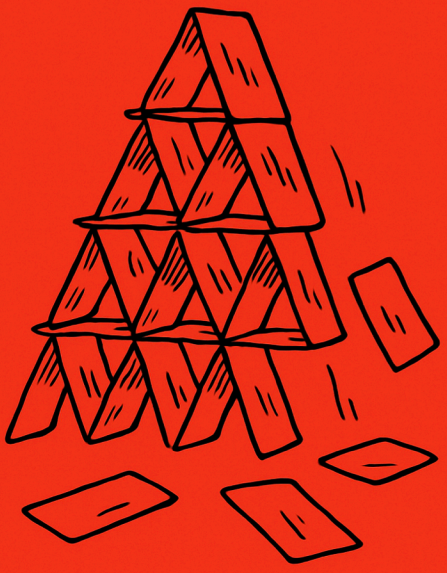
how software fails



the hidden laws of complex systems

by Engin Yöyen

How Software Fails

The Hidden Laws of Complex Systems

Engin Yöyen

This book is available at https://leanpub.com/how-software-fails

This version was published on 2025-09-08

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

Contents

Preliminaries . . . . . **i**
Errata & Suggestion i

Part I: Foundation 1

Chapter 1: When Cosmic Rays Attack **2**
What Do a Video Game, an Election, and an Airplane Have in Common? 2
The Bit That Changed Everything 3
The Cosmic Connection 6
The Scale Problem: When Rare Becomes Routine 7
Making Sense of the Impossible 10
Chapter 1 Key Takeaways 11

Chapter 2: The Rules of the Game **13**
How Faults Become Failures 13
What Makes a System "Complex"? 13
How Complex Systems Actually Fail 13
Designing for Inevitable Failure 14
Your Diagnostic Toolkit: Using Cook's Rules 15
Chapter 2 Key Takeaways 15

Part II: Failure Patterns 16

Chapter 3: The Blame Game **17**
Before the Fall: The Choices That Mattered 17
The Unraveling: When Simple Solutions Create Complex Problems . 17
The Detective Work: Following the Trail 17
The Pattern: How Complex Systems Fail 17
The Aftermath: Learning the Wrong Lessons 17

The Broader Truth: Why Root Cause Analysis Falls Short 17
Chapter 3 Key Takeaways 18

Chapter 4: Death by a Thousand Cuts 19
The Drift: When Reasonable Becomes Catastrophic 19
The Aftermath: Learning from Drift 19
The Psychology Behind Drift: How Normalization Enables Failure . . 19
The Pre-Mortem: Imagining Failure to Prevent It 19
Chapter 4 Key Takeaways 19

Chapter 5: When Humans and Machines Have Communication Problems 21
The Human Cost of Interface Failure 21
The Critical Development Failures That Created Catastrophe 21
The Unraveling: When "Normal" Means "Catastrophic" 21
The Design Problem: When Context Gets Lost in Translation 21
Chapter 5 Key Takeaways 22

Chapter 6: The Butterfly Effect Has Trust Issues 23
When Everything Connects to Everything 23
When Redundancy Becomes a Single Point of Failure 26
The Anatomy of a $370 Million Failure 27
The Pattern: How Tight Coupling Amplifies Small Failures 29
A Warmer Example: When Your House Becomes Too Smart 30
The Ripple Effect: How Local Failures Become Global Problems . . . 31
The False Promise of Redundancy 32
The Uncomfortable Truth About Connection 34
Chapter 6 Key Takeaways 35

Chapter 7: The Potemkin Village of Code 37
The Hidden Vulnerability: How Excellence Concealed Danger 37
The Performance Paradox: When Excellence Creates Vulnerability . 37
The Pattern: How Industries Optimize Into Vulnerability 37
Why We Can't See What's Hidden 37
Chapter 7 Key Takeaways 38

Chapter 8: Murphy's Law Meets Moore's Law 39
The Mathematics of Scale Amplification 39
The CrowdStrike Cascade: Anatomy of Global Failure 39
The Pattern: Scale as the Ultimate Amplifier 39
Why Scale Makes Recovery Impossible 39

Designing for Scale Resilience 40
Chapter 8 Key Takeaways 40

Part III: Making Peace with Chaos 42

Chapter 9: Breaking Things to Fix Them 43
The Setup: The Traditional Approach to Not Breaking 43
The Problem with Perfect Components 43
The Chaos Engineering Philosophy 43
The Unraveling That Works 43
Learning from Controlled Chaos 43
The Pattern: Antifragile Design 43
From Fighting Cook's Rules to Embracing Them 44
The Psychological Safety Requirement 44
Chapter 9 Key Takeaways 44

Chapter 10: The Internet Routes Around Everything 46
The Setup: From Network Research to Robust Design 46
The Routing Philosophy: Simple Rules, Complex Behavior 46
How the Internet Healed Itself 46
Lessons for Building Adaptive Systems 46
The Routing Paradox 46
Chapter 10 Key Takeaways 46

Chapter 11: Failing Fast, Failing Forward, Failing Better 48
The Setup: Building for Breakdown 48
The Philosophy of Graceful Degradation 48
The Unraveling That Wasn't 48
The Fast Failure Philosophy 48
The Recovery Time Objective 48
The Culture of Resilient Failure 48
The Resilience Investment Paradox 49
Chapter 11 Key Takeaways 49

Chapter 12: When Good Enough Isn't 50
Who Decides What Level of Broken Is Acceptable? 50
The Invisible Moral Layer 50
The Degraded Mode Dilemma 50
The Gambler's Ethics 50
Making the Invisible Visible 50

Chapter 12 Key Takeaways 50

Epilogue: Living with Inevitable Failure 52
The Pattern Beneath the Patterns 52
The Moral Thread 52
Building for the World We Actually Live In 52
The Future of Failure 52
The Wisdom of Inevitable Failure 52

Technical Appendix 53
Therac-25 Race Condition 53
Spectre Attack 55
CrowdStrike Channel File 291 Incident 56
Border Gateway Protocol (BGP) Technical Details 57

Preliminaries

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

Errata & Suggestion

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

Part I: Foundation

Part I establishes the foundational understanding that complex systems operate in a universe governed by probability rather than certainty, where perfect systems are mathematically impossible and failure is not a deviation from normal operation but an integral part of it. These two chapters draw on some of the work of safety researcher and physician Richard Cook, along with insights from other thinkers in complexity and system safety, showing how his principles explain system behavior and why the search for simple root causes in complex systems consistently leads us astray. **Software failures are not accidents, they are inevitabilities**.

Chapter 1: When Cosmic Rays Attack

Everything fails, eventually.

Given enough time, every system, whether natural or human-made, will eventually break down, degrade, or become obsolete. Software accumulates bugs. Hardware wears out. Aging biological organisms suffer molecular and cellular damage. Financial systems collapse, maybe because greed is in our nature.

We rely on software to do everything from controlling the speed of trains to managing hospital equipment. Our hardware powers life-critical systems like pacemakers and aircraft control. And we, as humans, are biological organisms made up of an enormous number of interacting molecules, organs, tissues, neurons, and more.

These are all examples of complex systems, networks of interconnected parts whose overall behavior emerges from their relationships and interactions. These systems evolve over time, often adapt to changing environments, and inevitably develop hidden points of failure. A strange feature of complex systems is that they can fail flawlessly: every component can function exactly as intended, every specification met, every test passed, yet the system as a whole can still produce outcomes no one predicted or desired.

Understanding why requires accepting an uncomfortable truth: our most critical systems, digital, biological, mechanical, and social, all operate in a universe that is constantly working against them. And sometimes, the universe wins in unexpected ways.

What Do a Video Game, an Election, and an Airplane Have in Common?

Let's consider three seemingly unrelated incidents that occurred in different times, different countries, and different domains:

Belgium, 2003: Electronic voting machines in a local election mysteriously added exactly 4,096 votes to one candidate. The number was oddly

specific, not a round number like 1,000 or 5,000, but precisely 4,096. The issue was noticed because the machine gave the candidate more votes than were possible. Election officials were baffled. The machines had been tested extensively and were working normally in all other respects[1].

Super Mario 64, 2013: A speedrunner (someone who tries to complete video games in record time) named DOTA_Teabag was attempting a world record in Super Mario 64 when something impossible happened. Mario spontaneously jumped upward without any controller input. The character launched himself into the air as if catapulted by an invisible force. The speedrunning community was baffled, such behavior shouldn't have been possible in this thoroughly analyzed[2]. A YouTuber even offered a $1000 bounty to anyone who can recreate the issue.

Qantas Flight 72, 2008: An Airbus A330 flying over the Indian Ocean suddenly pitched downward without warning. Over one-third of the 303 passengers and most of the crew sustained injuries, with a dozen people seriously hurt and nearly 40 requiring hospitalization. The aircraft's flight computers had received erroneous data and initiated an emergency descent. Investigators found no mechanical problems with the plane's systems[3].

What connects these three incidents across different technologies, different times, and different countries? The leading theory involves something so small it's invisible: a single **bit flip**.

The Bit That Changed Everything

Most people today know that computers run on a binary system, data is stored and processed using just two symbols: 0 and 1. These binary digits, or bits, may seem simple on their own, but when combined and interpreted correctly, they form the foundation of everything from emails and video calls to financial transactions and medical diagnostics.

---

[1] "Electronic voting in Belgium." Wikipedia, Wikimedia Foundation, 13 May 2025, https://en.wikipedia.org/wiki/Electronic_voting_in_Belgium.

[2] UncommentatedPannen. "TTC Upwarp Highlight" YouTube, March 2024. https://www.youtube.com/watch?v=bhBf5crp0i8&ab_channel=UncommentatedPannen

[3] Australian Transport Safety Bureau. (2011). In-flight upset 154 km west of Learmonth, WA, 7 October 2008, VH-QPA, Airbus A330-303. ATSB Transport Safety Report AO-2008-070. https://www.atsb.gov.au/sites/default/files/media/3532398/ao2008070.pdf

Why do computers use 0s and 1s? While binary mathematics and Boolean logic existed long before computers, George Boole developed Boolean algebra in the 1840s[4] (the name *Boolean algebra* was suggested later and not by George Boole), the reason computers adopted binary lies in physics. Electronic systems can most reliably distinguish between two distinct states: on or off, high voltage or low voltage, charged or uncharged. This makes binary the most practical way to implement mathematical operations in physical hardware. Binary systems are also more resilient against electrical noise than systems with multiple voltage levels. But they're not perfect.

A **bit flip** occurs when a single binary digit in a computer's memory unexpectedly changes from a 0 to a 1, or vice versa. This might sound trivial, but in some cases, the consequences can be dramatic.

Consider the Belgian voting incident. Investigators discovered a puzzling anomaly: a candidate mysteriously received 4,096 extra votes. There was no sign of fraud or software error. The leading theory? A cosmic ray struck the system's memory, flipping the 12th bit in the vote counter. That's all it took. Now you might wonder, how could flipping just one bit, from 0 to 1, suddenly add exactly 4,096 votes?

Here is a visual illustration of how this theory suggests it might have happened. The first diagram shows the candidate's vote count before the bit flip: 514 votes, represented in binary as `0000001000000010`. In this 16-bit representation (meaning 16 digits, each either 0 or 1), only two bits are set to 1: the 9th bit (representing 512) and the 1st bit (representing 2) giving a total of 512 + 2 = 514 votes.

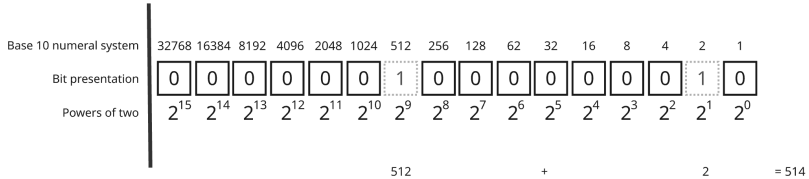


Figure 1. Base-10 and Bit Representation

The second diagram shows what would happen if a cosmic ray flipped the 12th bit from 0 to 1, changing the binary number to `0001001000000010`. This

[4] Boole, G. (1854). An Investigation of the Laws of Thought: On Which are Founded the Mathematical Theories of Logic and Probabilities. Walton and Maberly.

would add 4,096 to the vote count (since 2^12 = 4,096). Now we would have three bits set to 1: the 12th bit (4,096), the 9th bit (512), and the 1st bit (2), giving us 4,096 + 512 + 2 = 4,610 votes, exactly matching the mysterious 4,096 vote increase that baffled election officials.

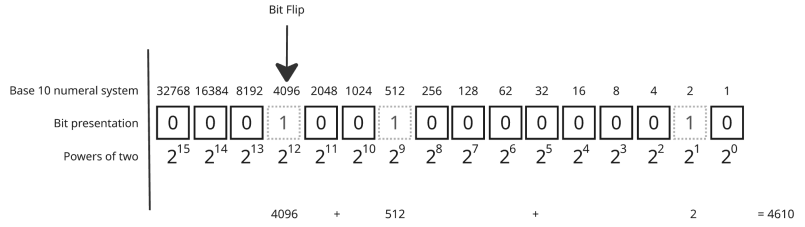


Figure 2. Bit Flip

In Mario's impossible jump, the leading theory is that a cosmic ray flipped a bit in the memory location storing his vertical velocity, changing it from zero to a positive value. In the Qantas incident, investigators suspect a similar bit flip occurred in the aircraft's air data computer, causing it to send false information to the flight control system.

What this shows is that when software is running, it often stores important state information (e.g. positions, speeds, or system statuses) as numeric values in memory. If one of these values is unexpectedly altered by a bit flip, the next time the software reads it, it may behave in ways that were never intended or anticipated.

These were not software bugs in the traditional sense. The code was working correctly. The hardware was functioning as designed. The problem came from the fundamental physics of information storage: when you represent information as electrical charges in microscopic silicon structures, those charges can be disturbed by forces as exotic as cosmic radiation.

Computer memory stores data using tiny electrical charges. A charged state might represent a "1" and an uncharged state a "0." But these charges exist in a universe filled with high-energy radiation that can, occasionally, deposit enough energy to flip the state of individual memory cells.

The phenomenon has been documented since the 1970s, when NASA first noticed mysterious computer issues in spacecraft beyond Earth's protective atmosphere. The aviation industry routinely accounts for cosmic ray inter-

ference[5]. The semiconductor industry builds error-correction systems into memory chips to combat bit flips from various sources, including manufacturing defects, thermal noise, and radiation effects like cosmic rays[6].

But for most of us, cosmic ray interference remains invisible, until Mario jumps when he shouldn't, or votes get miscounted, or planes dive unexpectedly.

The Cosmic Connection

The cosmic rays bombarding Earth originate from some of the most violent events in the universe: exploding stars, colliding galaxies, and black holes accelerating particles to nearly the speed of light. These high-energy particles travel across interstellar space for millions of years before striking our atmosphere.

When high energy cosmic rays, mostly originating from outside our solar system, hit atoms in Earth's upper atmosphere, they are usually blocked or deflected by the atmosphere and our planet's magnetic field. But sometimes these collisions produce cascades of secondary particles, including neutrons, that can travel all the way to ground level. Neutrons, being electrically neutral, are difficult to detect directly, but they can still collide with atoms inside electronic components, depositing enough energy to flip the state of individual memory cells

The probability of any specific bit being flipped by a cosmic ray is extraordinarily small, roughly one in several billion per bit per year. But scale matters. Modern computers process billions of bits every second. Data centers contain millions of memory chips. The global digital infrastructure handles quintillions of bits daily.

At this scale, the extremely improbable becomes inevitable. Somewhere in the world, cosmic rays are flipping bits right now, in smartphones, servers, voting machines, aircraft computers, and game consoles. Most of these bit flips are harmless or caught by error-correction systems. But occasionally, a cosmic particle from a dying star flips exactly the right bit at exactly the right

[5]D. Binder, E. C. Smith and A. B. Holman, "Satellite Anomalies from Galactic Cosmic Rays," in IEEE Transactions on Nuclear Science, vol. 22, no. 6, pp. 2675-2680, Dec. 1975, https://ieeexplore.ieee.org/document/4328188

[6]Dell, T. J. (1997). A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division.

moment to cause exactly the kind of "impossible" failure that reminds us how little control we actually have over the complex systems we've built.

Meanwhile, in risk theory...

What these incidents illustrate has a name in risk theory: *normal accidents*. Sociologist Charles Perrow coined this term to describe failures that are inevitable in complex, tightly coupled systems, even when all components work correctly and all operators perform competently.

Normal accidents occur because complex systems exhibit *emergent properties*, behaviors that arise from the interactions between components rather than from the components themselves. These emergent behaviors can't be predicted by analyzing individual components, no matter how thoroughly[7].

The cosmic ray incidents didn't break anything. They simply changed one bit of information at exactly the right moment to create emergent behaviors that emerged from the interaction between cosmic physics and digital logic.

Nassim Taleb's *Black Swan* theory complements this perspective by highlighting how cosmic ray failures represent events that are rare, have extreme impact, and are retrospectively predictable but prospectively difficult to forecast. Like black swans, cosmic ray incidents seem impossible until they happen, then we construct explanations that make them appear inevitable. The combination of normal accidents (systematic vulnerabilities in complex systems) and black swan events (rare but high-impact occurrences) explains why cosmic ray failures are both theoretically predictable and practically surprising[8].

[7]Perrow, Charles. Normal Accidents: Living with High Risk Technologies. Princeton University Press, 1999.

[8]Taleb, N. N. (2010). The Black Swan: The Impact of the Highly Improbable. Random House.

The Scale Problem: When Rare Becomes Routine

One of the most counterintuitive aspects of complex system failures is how extremely rare events become common when systems operate at sufficient scale.

Consider electric vehicles: based on IBM's research on cosmic ray-induced bit flips, which found approximately one error per month per 256 MiB of RAM[9], a critical memory error in a single car's control system might have a FIT (failures per billion hours) exposure on the order of 1,000–10,000, depending on memory size and system criticality. For any individual vehicle, this probability is extremely low. But consider what happens when we scale up:

- **Electric vehicles on roads**: Approximately 58 million electric vehicles worldwide as of 2024[10]
- **Hours of operation**: If each vehicle operates 290 hours annually (based on US average of 293 hours driving time), that represents nearly 17 billion hours of total operation[11]
- **Expected cosmic ray events**: Using a conservative estimate of 1,000–10,000 FIT per critical system, with 17 billion hours of operation, we would expect approximately 17,000–168,000 cosmic ray-induced failures across all electric vehicles annually

Suddenly, an event that seems impossibly rare for any individual driver becomes virtually certain to occur somewhere in the global electric vehicle fleet.

This is precisely why error-correcting codes (ECC) and other defensive mechanisms become critical at scale. ECC memory can detect and automatically correct single-bit errors, reducing the effective failure rate by orders of magnitude. Without ECC, every one of those 17,000–168,000 bit flips could

[9] J. F. Ziegler et al., "IBM experiments in soft fails in computer electronics (1978–1994)," in IBM Journal of Research and Development, vol. 40, no. 1, pp. 3-18, Jan. 1996, doi: 10.1147/rd.401.0003.

[10] International Energy Agency (2025). Global EV Outlook 2025. IEA Publications. Retrieved from https://www.iea.org/reports/global-ev-outlook-2025/trends-in-electric-car-markets-2

[11] AAA Foundation for Traffic Safety. Americans Spend 293 Hours Driving Each Year. Retrieved from https://www.automotive-fleet.com/136735/americans-spend-an-average-of-17-600-minutes-driving-annually

potentially cause system malfunctions. With ECC protection, only the rare multi-bit errors or errors in unprotected memory regions pose serious risks.

However, ECC isn't universally deployed. Many consumer systems, including smartphones, tablets, and lower-end automotive components (e.g., window regulators, infotainment), still rely on non-ECC memory to reduce costs. This creates a concerning gap: as we deploy billions of devices without cosmic ray protection, we're essentially gambling that rare events won't happen often enough to matter. But as our calculations show, at sufficient scale, rare events become routine occurrences.

The semiconductor industry learned this lesson the hard way. Early spacecraft failures due to cosmic ray interference drove the development of radiation-hardened electronics and ECC memory systems. Today, virtually all servers in major data centers use ECC memory, not because individual failures are likely, but because at the scale of millions of servers processing exabytes of data, cosmic ray interference is a predictable operational expense rather than a theoretical concern.

At these scales, events with probabilities of one in a trillion become daily occurrences. Events with probabilities of one in a quadrillion happen multiple times per year. The "impossible" becomes not just possible, but predictable.

This brings us to a fundamental principle that governs complex systems operating in a probabilistic universe: ***Anything that can go wrong will go wrong***[12], [13].

The principle is named after Edward A. Murphy Jr., an American aerospace engineer who was working on rocket sled tests at US Air Force Base in 1949. When strain gauges were wired incorrectly during a critical experiment, Murphy observed that if there were multiple ways to do something and one of those ways could result in disaster, that's inevitably the way it would be done. What started as an frustrated engineer's observation about human fallibility became a fundamental law of complex systems.

Murphy's Law isn't just a cynical joke, it's a statistical certainty when you combine the scale, complexity, and fragility of modern computing with the countless ways things can fail. From hardware faults and software bugs to random environmental interference, failure isn't just possible, it's inevitable.

In honor of Edward A. Murphy Jr., each chapter includes a "Murphy's

[12] Matthews, Robert A. J. "The Science of Murphy's Law." *Scientific American* 276, no. 4 (April 1997): 88-91. https://doi.org/10.1038/scientificamerican0497-88.

[13] "Murphy's Law." Wikipedia. https://en.wikipedia.org/wiki/Murphy%27s_law

Postcard", a single observation that captures the ironic wisdom of an engineer who discovered that the universe has a sense of humor about our best-laid plans.

Murphy's Postcard

In a universe with billions of cosmic rays and billions of computers, the impossible happens several times before lunch.

Making Sense of the Impossible

The cosmic ray incidents reveal a fundamental truth: in a universe governed by probability rather than certainty, perfect systems are impossible. At scale, the extremely improbable becomes inevitable.

This reality challenges our faith in comprehensive testing. For decades, software engineering has relied on testing as our primary defense against failure, the systematic process of running code through various scenarios to identify and fix problems before users encounter them. As computer scientist Edsger Dijkstra observed in 1969: "Testing shows the presence, not the absence of bugs"[14]. Testing is crucial, it helps us find and fix many problems, and it gives us confidence in our systems. But it cannot guarantee perfection, especially when dealing with events so rare they may never occur during testing yet become inevitable at scale. In the coming chapters, we'll see how even extensively tested systems can fail in ways their creators never imagined.

But that's not the end of the story, it's the beginning. If we accept that certain kinds of failures are mathematical certainties, we can stop wasting energy trying to prevent the inevitable and start designing systems that work despite it.

The cosmic rays are still coming. They've been coming since the formation of the universe, and they'll keep coming long after our civilization has passed

[14] Dijkstra, E. W. (1969). In J.N. Buxton and B. Randell, eds, Software Engineering Techniques, Report on a NATO Science Committee conference, Rome, Italy, 27–31 October 1969, p. 16.

into history. We can't stop them, but we can learn to build systems that work despite them.

If outer space can break your Super Mario, what chance does your code have?

The answer is: *exactly the chance you design for it to have*.

But understanding how to design for inevitable failure requires us to think differently about the systems we build. Before we start investigating specific disasters, we need to understand the rules that govern how complex systems actually behave when they encounter the impossible.

Chapter 1 Key Takeaways

Failure is inevitable in complex systems, not exceptional

In a universe governed by probability rather than certainty, cosmic ray interference, quantum effects, and countless other physical phenomena make perfect systems impossible. When systems process millions of operations per day, events with probabilities of one in a billion become daily occurrences. The Belgian voting machines, Mario's impossible jump, and Qantas Flight 72 all demonstrate that at sufficient scale, the extremely improbable becomes mathematically certain.

Complex systems fail perfectly through emergent behaviors

All components can function exactly as designed, every specification met, every test passed, yet the system as a whole can still produce unintended outcomes. These "normal accidents" arise from interactions between components rather than component failures themselves. Perfect testing cannot reveal all failure modes because some are too rare to reproduce reliably yet become inevitable at scale.

Scale transforms the impossible into the inevitable

With 58 million electric vehicles operating 17 billion hours annually, cosmic ray-induced failures that seem impossibly rare for individual drivers become predictable fleet-wide events. At sufficient scale, Murphy's Law becomes a statistical certainty rather than pessimistic observation.

Graceful degradation beats the pursuit of perfection

Complex systems always run in degraded mode, your system is already more broken than it appears. Accept that something, somewhere, will go wrong, and design for adaptability rather than trying to prevent the inevitable.

* * *

Next: Chapter 2 establishes the foundational framework for understanding complex system failures through Richard Cook's principles.

Chapter 2: The Rules of the Game

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

How Faults Become Failures

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

What Makes a System “Complex”?

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

How Complex Systems Actually Fail

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

Rule 1: Complex Systems Are Intrinsically Hazardous

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

Rule 2: Complex Systems Run in Degraded Mode

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

Rule 3: Catastrophe Requires Multiple Failures

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

Rule 4: Complex Systems Contain Changing Mixtures of Failures

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

Rule 5: There Is No Single "Root Cause"

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

Rule 6: Change Creates New Failure Modes

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

Rule 7: System Interactions Create Unexpected Failure Paths

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

Rule 8: Humans Are the Adaptive Element

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

Rule 9: Learning Requires Experience With Failure

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

Rule 10: All Practitioner Actions are Gambles

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

Rule 11: People Continuously Create Safety

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

Designing for Inevitable Failure

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

Your Diagnostic Toolkit: Using Cook's Rules

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

Chapter 2 Key Takeaways

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

System failures follow the fault-error-failure chain

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

Emergent behavior means complex systems can fail perfectly

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

Effective failure investigation requires detective thinking

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

Understanding complexity patterns enables building resilient systems

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

Part II: Failure Patterns

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

Chapter 3: The Blame Game

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

Before the Fall: The Choices That Mattered

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

The Unraveling: When Simple Solutions Create Complex Problems

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

The Detective Work: Following the Trail

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

The Pattern: How Complex Systems Fail

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

The Aftermath: Learning the Wrong Lessons

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

The Broader Truth: Why Root Cause Analysis Falls Short

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

Chapter 3 Key Takeaways

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

Complex system failures have multiple parents, never single “root causes”

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

Well-intentioned decisions by competent people collectively create catastrophe

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

The search for blame obscures systemic understanding

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

Move beyond root cause analysis to systems thinking

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

Chapter 4: Death by a Thousand Cuts

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

The Drift: When Reasonable Becomes Catastrophic

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

The Aftermath: Learning from Drift

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

The Psychology Behind Drift: How Normalization Enables Failure

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

The Pre-Mortem: Imagining Failure to Prevent It

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

Chapter 4 Key Takeaways

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

Multiple system failures create cascading crises

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

Normalization of deviance enables drift

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

Technical failures have profound human consequences

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

Drift into failure follows predictable patterns

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

Governance-expertise gaps amplify drift into failure

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

Preventing drift requires psychological safety

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

Chapter 5: When Humans and Machines Have Communication Problems

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

The Human Cost of Interface Failure

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

The Critical Development Failures That Created Catastrophe

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

The Unraveling: When “Normal” Means “Catastrophic”

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

The Technical Root Cause: Software Faults in Safety-Critical Systems

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

The Design Problem: When Context Gets Lost in Translation

Chapter 5 Key Takeaways

Semantic gaps between interfaces and reality create deadly translation failures

Cognitive heuristics make humans vulnerable to systematic interface deception

Mental model mismatches turn operator expertise into liability

Critical systems must provide meaningful feedback loops and error representation

Design constraints that prevent dangerous configurations, not operator vigilance

Chapter 6: The Butterfly Effect Has Trust Issues

Backup systems are supposed to save you. Sometimes, they're the ones that kill you.

We've already seen how multiple failures can combine to bring down a trading firm in seconds (Knight Capital), how organizations can drift into disaster over months or years (Birmingham), and how lethal energy at the human–machine interface can turn ordinary operator actions into fatal outcomes (Therac-25). Now we turn to a different danger: tightly coupled failures, where a safeguard can destroy the very system it's meant to protect.

On June 4, 1996, the European Space Agency launched flight 501 of the Ariane 5 rocket, their most advanced launch vehicle ever built. Thirty-seven seconds after liftoff, the rocket suddenly veered off course and self-destructed[1]. In forty seconds, ten years of work and $370 million in hardware became a firework display over French Guiana.

The cause wasn't a faulty engine or broken hardware, it was code that wasn't even supposed to be running during flight.

But here's the thing about tightly coupled systems: your fitness tracker notices you're not sleeping well, so it tells your smart thermostat to lower the temperature, which signals your coffee machine to brew stronger coffee, which alerts your car that you'll need extra navigation prompts, which overwhelms your garage door opener, and suddenly you're trapped in your own driveway. Welcome to the future, where asking for a better night's sleep can literally lock you out of your life.

This absurd chain demonstrates how well-intentioned automation can create unexpected dependencies and failure modes. Each system is trying to be helpful based on information from the previous one, but their collective "intelligence" produces outcomes nobody designed or wanted.

[1]Dowson, M. (1997). The Ariane 5 software failure. SIGSOFT Softw. Eng. Notes 22, 2 (March 1997), 84. https://doi.org/10.1145/251880.251992

When Everything Connects to Everything

To understand how a backup system can destroy the thing it's supposed to protect, you need to understand what researchers in complex systems call "tight coupling", the degree to which components in a system are connected and dependent on each other.

In loosely coupled systems, components operate relatively independently. When one part fails, the others continue working normally. Think of a traditional car: if the radio breaks, you can still drive. If the air conditioning fails, the engine keeps running. Each system has clear boundaries and limited interactions with other systems.

But in tightly coupled systems, components are interconnected in ways where one part's behavior immediately affects many others. However, tight coupling manifests in different patterns, each creating distinct types of failure cascades.

Three Types of Tight Coupling

Sequential coupling creates failure chains where components are arranged in a straight line, with each depending on the previous one. Think of a traditional assembly line: "material shortage > production halt > quality control backup > shipping delay > customer cancellation > revenue loss". Each failure immediately causes the next, like dominoes falling in a predetermined order. When one node fails, everything after it in the sequence stops working.

Hierarchical coupling creates tree-like dependency structures where components are arranged in layers, with higher-level components controlling multiple lower-level ones. A failure at a higher level can affect entire branches of the system below it, like an organizational chart where a department head's absence affects all subordinate teams.

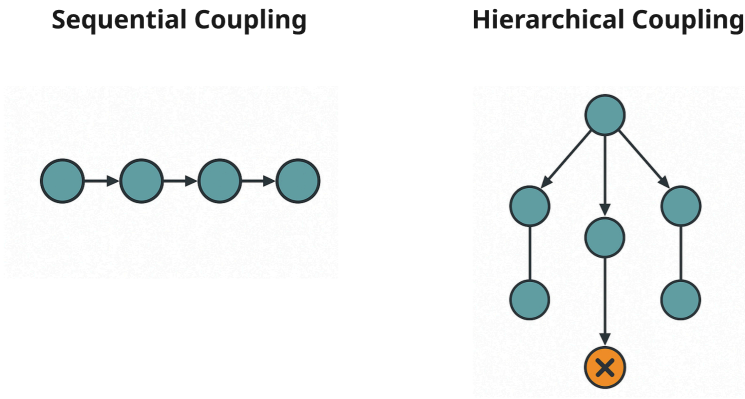


Figure 3. Sequential vs Hierarchical Coupling

The diagram above shows both patterns. On the left, sequential coupling arranges components in a chain where failure propagates step by step. On the right, hierarchical coupling creates a tree structure where a failure at one level (shown in orange) can impact multiple branches below it.

Networked coupling creates failure webs where one component's failure simultaneously affects multiple other components. This pattern appears in power grids, computer networks, and distributed systems. When one node fails, the failure spreads through multiple pathways to affect many connected components at once.

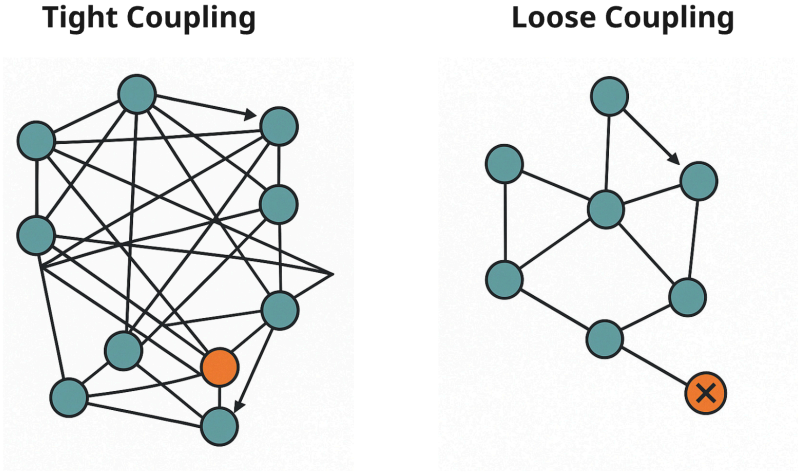


Figure 4. Tight vs Loosely Coupling

The networked coupling diagram above illustrates the difference from loose coupling. On the right, loosely coupled systems have sparse connections, when one component fails (marked with X), the failure stays contained. On the left, tightly coupled systems have dense interconnections, a single component failure (shown in orange) rapidly spreads to many other components simultaneously through multiple dependency pathways.

Modern cars demonstrate both types: sequential coupling in the drive train (engine > transmission > wheels) and networked coupling in the infotainment systems (one central computer failure can simultaneously affect radio, navigation, climate controls, and backup camera display).

Now that we understand how tight coupling works in theory, let's see how it destroyed a $370 million rocket in practice.

When Redundancy Becomes a Single Point of Failure

The Ariane 5 was a tightly coupled system par excellence. Its Inertial Reference System (SRI) didn't just provide navigation data, it was connected to the main flight computer, which controlled the engine nozzles, which

determined the rocket's trajectory, which triggered the self-destruct system if anything went wrong.

The SRI itself was actually two computers running identical software, designed to provide redundancy. If one computer failed, the other would seamlessly take over. This had worked brilliantly on the Ariane 4, where the same SRI design had operated successfully for fifteen years without a single failure.

But the Ariane 5 flew a different trajectory profile that created flight conditions the Ariane 4 had never encountered.

The engineers knew this, of course. They had extensively tested the new rocket's primary systems, upgraded the engines, redesigned the fuel tanks, and verified the main flight software. But the SRI was a proven component. It had worked perfectly for fifteen years. Why reinvent something that wasn't broken?

The Overconfidence Effect

The Ariane 5 failure may illustrate what psychologists call the overconfidence effect. Fifteen years of flawless Ariane 4 performance could have encouraged engineers to believe the Inertial Reference System's reliability would carry over to a radically different launch vehicle. As Daniel Kahneman notes, past success can make us overestimate our ability to predict and control outcomes, especially in novel situations[2].

The Anatomy of a $370 Million Failure

The Ariane 5 was a different beast from its predecessor. Bigger, faster, more powerful, it experienced higher acceleration forces and flew a trajectory the Ariane 4 had never encountered.

Thirty-six seconds after launch, the Ariane 5 was traveling faster and at a steeper angle than any Ariane 4 had ever achieved. Inside the SRI, software was tracking horizontal velocity using a 16-bit signed integer that could only

[2]Kahneman, D. (2011). Thinking, fast and slow. Farrar, Straus and Giroux.

represent values up to 32,767 cm/s. The Ariane 4 had never exceeded this limit, but the Ariane 5's horizontal velocity reached 37,000 cm/s.

When the software tried to store this larger value, it caused an overflow exception[3], the number simply didn't fit. The SRI, following good programming practice, detected this exception and shut down safely rather than operating with corrupted data. But the backup SRI ran identical software and failed 72 milliseconds later for the same reason. Both navigation computers crashed simultaneously, turning intended redundancy into synchronized failure.

The main flight computer, suddenly without navigation data, interpreted the last message from the SRI, actually an error diagnostic, as genuine navigation data. Believing the rocket was catastrophically off course, it commanded aggressive steering corrections. The rocket, which had been flying perfectly, lurched sideways under these extreme commands until aerodynamic forces broke it apart. The self-destruct system then did exactly what it was designed to do[4].

The failure wasn't in any single component, it was in how the components were connected. Each system behaved exactly as designed, but their interaction created an emergent behavior nobody had predicted. This exemplifies Cook's Rule 3: catastrophic failures require multiple failures working in combination, not single-point failures.

The investigation revealed four interconnected stories:

The Software Story: The fatal horizontal velocity calculation wasn't even needed during flight, it was diagnostic code left over from ground testing that continued running "just in case" it might be useful for later analysis.

The Reuse Story: The SRI software was tested extensively, but only for Ariane 4 flight profiles. The Inquiry Board noted that Ariane 5 trajectory data was deliberately excluded from SRI requirements because the calculation wasn't considered critical to flight operations. This demonstrates Cook's Rule 6: change creates new failure modes, reusing proven components in new contexts introduces unexpected vulnerabilities.

[3]An overflow exception occurs when a program tries to store a number that is too large for the allocated memory space. For example, trying to put the value 50,000 into a container that can only hold numbers up to 32,767 causes an overflow. The computer detects this error and typically stops the program to prevent data corruption.

[4]European Space Agency. "Ariane 501 - Presentation of Inquiry Board report." ESA Press Release, July 19, 1996. https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report

The Redundancy Story: The dual SRI computers weren't truly redundant, they were identical, running the same software and encountering the same failure conditions. True redundancy requires independence; this was synchronized vulnerability.

The Integration Story: Tight coupling between navigation and flight control meant no isolation, no graceful degradation. The system was designed to be either perfectly functional or completely destroyed, with nothing in between.

Meanwhile, in systems theory...

What the Ariane 5 experienced has a name in systems theory: "common mode failure." This occurs when multiple redundant components fail simultaneously due to a shared vulnerability that affects all of them equally[5]**.**

In tightly coupled systems, common mode failures are particularly dangerous because the coupling amplifies and accelerates the failure propagation. A failure that starts in one component doesn't stay contained, it cascades through the connected systems faster than human operators (or automated systems) can respond.

The mathematical term for this is "failure correlation", the degree to which component failures are statistically dependent rather than independent. In truly independent systems, the probability of total system failure is the product of individual component failure probabilities. But when failures are correlated, this math breaks down completely.

The Pattern: How Tight Coupling Amplifies Small Failures

The Ariane 5 disaster illustrates a fundamental characteristic of complex systems: tight coupling can transform minor local failures into major system

[5]Common mode failure occurs when multiple redundant components fail simultaneously due to a shared cause. In reliability engineering, this represents a failure of the independence assumption that underlies redundancy strategies.

catastrophes. This isn't a bug in system design, it's an inherent trade-off. The same coupling that enables sophisticated system behaviors also creates pathways for failure propagation.

This illustrates the central paradox of modern system design: the integration that makes systems powerful also makes them fragile, as Cook's Rule 1 observes, complex systems are intrinsically hazardous, with the same properties that enable capabilities also creating cascade failure vulnerabilities.

A Warmer Example: When Your House Becomes Too Smart

While rocket explosions grab headlines, the same tight coupling problems affect the mundane systems we interact with daily. Consider what happened to thousands of Nest thermostat users during the winter of 2016.

Nest thermostats are sophisticated devices designed to learn your schedule, optimize energy usage, and respond to remote control through smartphone apps. They're part of the "Internet of Things" revolution that's making our homes smarter and more efficient.

But on January 12, 2016, something went wrong with a software update. Thousands of Nest users woke up to cold houses and dead thermostats. The devices had received an automatic firmware update that contained a bug causing excessive battery drain. The thermostats' batteries died overnight, leaving homes without heat during one of the coldest weeks of the year[6].

This wasn't just an inconvenience, it was a perfect example of how tight coupling in modern systems can amplify the impact of seemingly minor failures.

The Connectivity Story: Traditional thermostats are simple devices with minimal external dependencies. They read temperature, compare it to a setpoint, and turn heating systems on or off. If they fail, they usually fail in a predictable way that homeowners can work around.

But smart thermostats are tightly coupled to cloud services, home wifi networks, smartphone apps, and automatic update systems. This coupling

[6]The New York Times. "Nest Thermostat Owners Report Losing Heat After Software Update." January 13, 2016. https://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html

enables sophisticated features like remote control, learning algorithms, and energy optimization. But it also means that a failure in any connected system can disable the entire heating system.

The Update Story: The problematic software update was pushed automatically to devices without user intervention. This automatic update capability is generally beneficial, it allows manufacturers to fix bugs, add features, and patch security vulnerabilities without requiring users to manually update their devices.

The same distribution mechanism that enabled rapid deployment of improvements became a vector for rapid deployment of failures. This pattern would later manifest globally when CrowdStrike's cybersecurity update system pushed a single faulty file to millions of computers worldwide in 2024, crashing 8.5 million Windows machines and triggering the largest IT outage in history (Chapter 8).

The Power Story: Traditional thermostats typically run on household electrical power with simple battery backup. If the power fails, the thermostat continues operating until power is restored. But smart thermostats require significant computational power for wifi connectivity, cloud communication, and user interface processing.

The tight coupling between software complexity and power consumption meant that a software bug causing excessive CPU usage could drain batteries faster than the charging system could replenish them. A pure software problem became a hardware failure.

The Dependency Story: The Nest thermostat's operation depended on a chain of external services: internet connectivity, cloud servers, mobile app infrastructure, and automatic update systems. Failure in any link of this chain could potentially impact device operation.

When users couldn't manually override their dead thermostats because the manual controls required the same systems that had failed, the tight coupling between different interface mechanisms turned a software update problem into a complete loss of heating control.

The Ripple Effect: How Local Failures Become Global Problems

What made both the Ariane 5 explosion and the Nest thermostat freeze particularly instructive is how they demonstrate "failure amplification" in tightly coupled systems. Small, localized problems cascade through system connections to create impacts far disproportionate to their original scope.

In the Ariane case, a single calculation that wasn't even necessary for flight operations brought down an approximately $370 million rocket. In the Nest case, a software bug in power management left thousands of homes without heat during a cold snap.

This amplification occurs because system-level weaknesses arise from component interactions rather than components themselves, illustrating Cook's Rule 7: system interactions create unexpected failure paths that emerge from connections between components.

Traditional reliability engineering focuses on making individual components more reliable, but in connected systems, reliability depends more on managing interactions than perfecting individual parts. You can't just make the parts better, you have to manage how the parts interact.

Think of failure propagation like an infection spreading through a population. The Ariane 5 demonstrated a high "reproduction number"[7], one failed calculation infected the backup computer within 72 milliseconds, then spread to flight control, structural systems, and self-destruct within 40 seconds. Each connection provided a pathway for failure to jump to the next component.

The False Promise of Redundancy

Both the Ariane 5 explosion and the Nest thermostat freeze illustrate a crucial misconception about redundancy in tightly coupled systems. Engineers often assume that adding backup systems increases reliability, but tight coupling can actually make redundancy counterproductive.

[7]The reproduction number (R_0) is a measure from epidemiology that indicates how many new infections are caused by each infected individual. In system failures, it represents how many additional system components fail as a result of each initial component failure.

True redundancy requires what engineers call “diversity”, backup systems sufficiently different from primary systems that they don’t share the same vulnerabilities. The Ariane 5’s dual SRI computers weren’t diverse; they were identical, creating “redundant vulnerability” where multiple components fail in exactly the same way simultaneously.

Diversity might involve different hardware architectures, software algorithms, data processing approaches, operating timescales, or even development teams. But diversity costs more and complicates integration. Most systems optimize for integration rather than resilience because integration benefits are immediate and obvious, while coupling costs only appear during rare failure events.

What True Redundancy Looks Like

Airbus A340 Flight Control System: The A340’s flight control architecture includes five self-checking computers: three of these run the primary flight control software, while the other two run separate monitoring software. The three primary computers operate in parallel and use a voting system (similar to triple modular redundancy, TMR) to mask hardware faults by selecting the majority output. The additional two computers perform continuous cross-checks on the outputs and the health of the system, improving fault detection and overall reliability[8].

This arrangement primarily addresses hardware faults through redundancy and cross-checking. However, because the primary computers run the same software version, this setup alone cannot fully mitigate software faults that affect all computers simultaneously. To handle software faults, techniques such as N-version programming, where multiple independently developed software versions run in parallel and their results compared, are needed[9].

[8]Sommerville, I. (2016) Software Engineering. 10th Edition, Pearson Education Limited, Boston.

[9]Mukwevho, M. A., & Celik, T. (2021). Toward a Smart Cloud: A Review of Fault-Tolerance Methods in Cloud Systems. IEEE Transactions on Services Computing, 14(2), 589– 605. https://doi.org/10.1109/TSC.2018.2816644

Murphy's Postcard

When your toaster can crash your rocket, maybe it's time to reconsider the home automation plan.

The Uncomfortable Truth About Connection

The most unsettling aspect of these failures is that they result from our best intentions. We connect systems to make them more capable, efficient, intelligent. Each connection solves real problems and provides genuine value.

Despite interconnected complexity, most coupled systems work most of the time because human operators constantly adapt, creating what Richard Cook calls "pathways for retreat or recovery from expected and unexpected faults." Air traffic controllers, power grid operators, software engineers, they all manage complexity no single mind can fully comprehend. This reflects Cook's Rule 8: humans are the adaptive element that makes complex systems resilient, and Cook's Rule 11: people continuously create safety through ongoing adjustments and workarounds.

This isn't an argument for disconnecting everything. Connected systems enable smart cities, coordinated healthcare, integrated transportation that solve real problems and save lives. Instead, it's an argument for designing with cascade failures in mind: building circuit breakers that isolate failures, creating diverse rather than identical redundancy, accepting that perfect reliability is impossible but controlled degradation is achievable.

In loosely coupled systems, system reliability follows from component reliability. But in connected systems, behavior emerges from component interactions, requiring what complexity theorists call "adaptive capacity", the ability to recognize and respond to unexpected system states.

Modern system failures seem surprising because we're building systems that exhibit behaviors no individual designer intended. The Ariane 5 explosion and Nest thermostat freeze weren't caused by incompetence, but by the fundamental challenge of predicting how interconnected systems behave under novel conditions. We've created a world where the most critical skill

isn't predicting failure, it's responding gracefully when the unpredictable happens.

Chapter 6 Key Takeaways

Connection enables capability but creates cascade pathways (Cook's Rules 1 & 7)

The Ariane 5's explosion began with diagnostic code that wasn't needed during flight, calculating horizontal velocity that exceeded a 16-bit integer limit. Connection between backup navigation and flight control meant one overflow exception destroyed the entire rocket within 40 seconds, integration transformed a minor calculation error into system-wide catastrophe. This demonstrates that complex systems are intrinsically hazardous (Rule 1), with system interactions creating unexpected failure paths (Rule 7).

True redundancy requires diversity, not duplication (Cook's Rules 3 & 6)

Ariane 5 had two navigation computers for safety, but both ran identical software and failed simultaneously from the same overflow. Nest thermostats had identical software worldwide, each vulnerable to the same battery-draining bug. Redundancy without diversity creates synchronized failure rather than protection. This illustrates how catastrophic failures require multiple component failures working in combination (Rule 3), often triggered when system changes create new failure modes (Rule 6).

Failures propagate faster than human response can adapt (Cook's Rules 8 & 11)

When coupling works, emergent capabilities arise that no component could achieve alone. When coupling fails, disasters unfold faster than human adaptation, making prevention more critical than reaction. The Ariane 5's failure cascade, navigation to flight control to structural failure, occurred within 40 seconds. This highlights how humans serve as the adaptive element in complex systems (Rule 8), continuously creating safety through ongoing adjustments and workarounds (Rule 11), until failure speed exceeds human response capacity.

Shared pathways turn local failures into systemic disasters (Cook's Rules 1 & 7)

Nest thermostats' software coupling to power management meant one update bug disabled heating across thousands of homes. CrowdStrike's global

update system crashed 8.5 million computers from a single file. When components share failure pathways, local problems become systemic catastrophes. This reinforces that complex systems contain inherent hazards (Rule 1), where system interactions create unexpected failure paths that amplify small problems into large disasters (Rule 7).

* * *

Next: Chapter 7 explores how systems can appear robust and secure while harboring systematic vulnerabilities, the "Potemkin village" effect that makes dangerous systems look safe.

Chapter 7: The Potemkin Village of Code

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

The Hidden Vulnerability: How Excellence Concealed Danger

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

The Discovery: When Fast Code Revealed Hidden Secrets

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

The Performance Paradox: When Excellence Creates Vulnerability

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

The Pattern: How Industries Optimize Into Vulnerability

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

Why We Can't See What's Hidden

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

Chapter 7 Key Takeaways

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

Performance optimization created intrinsic hazards (Cook's Rules 1, 4)

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

Multiple factors aligned to hide systematic vulnerabilities (Cook's Rule 5)

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

Safety-creation processes systematically missed timing attacks (Cook's Rule 11)

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

Industry-wide optimization created emergent attack surfaces (Cook's Rule 7)

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

Adversarial thinking breaks optimization-vulnerability cycles

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

Chapter 8: Murphy's Law Meets Moore's Law

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

The Mathematics of Scale Amplification

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

The CrowdStrike Cascade: Anatomy of Global Failure

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

The Pattern: Scale as the Ultimate Amplifier

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

Why Scale Makes Recovery Impossible

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

Automated Failure, Manual Recovery

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

Testing at Impossible Scale

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

Hidden Dependencies Revealed

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

Designing for Scale Resilience

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

Chapter 8 Key Takeaways

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

Scale amplification follows power law mathematics where critical thresholds create global catastrophes

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

Cook's Rules become laws of physics at planetary scale (Cook's Rules 1-4)

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

Scale amplifies all previous failure patterns into civilization-level phenomena

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

Automated failure enables instant global destruction but forces sequential manual recovery

Planetary-scale systems cannot be tested at planetary scale

Part III: Making Peace with Chaos

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

Chapter 9: Breaking Things to Fix Them

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

The Setup: The Traditional Approach to Not Breaking

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

The Problem with Perfect Components

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

The Chaos Engineering Philosophy

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

The Unraveling That Works

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

Learning from Controlled Chaos

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

The Pattern: Antifragile Design

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

From Fighting Cook's Rules to Embracing Them

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

The Psychological Safety Requirement

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

Chapter 9 Key Takeaways

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

Chaos engineering improves system reliability

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

Antifragile systems become stronger through stress

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

Psychological safety is essential for chaos engineering

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

Focus on failure recovery metrics

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

Controlled failure injection creates learning opportunities

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

Chapter 10: The Internet Routes Around Everything

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

The Setup: From Network Research to Robust Design

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

The Routing Philosophy: Simple Rules, Complex Behavior

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

How the Internet Healed Itself

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

Lessons for Building Adaptive Systems

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

The Routing Paradox

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

Chapter 10 Key Takeaways

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

Distributed systems achieve global resilience through local decision-making

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

Simple rules enable emergent behaviors that exceed centralized control

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

Apparent inefficiency often represents resilience features

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

Trust-based protocols enable rapid adaptation

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

Learning from failure creates operational resilience

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

Chapter 11: Failing Fast, Failing Forward, Failing Better

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

The Setup: Building for Breakdown

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

The Philosophy of Graceful Degradation

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

The Unraveling That Wasn't

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

The Fast Failure Philosophy

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

The Recovery Time Objective

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

The Culture of Resilient Failure

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

The Resilience Investment Paradox

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

Chapter 11 Key Takeaways

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

Adaptive capacity is more valuable than prevention

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

Multi-layered graceful degradation enables mission resilience

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

Multiple failures create learning opportunities rather than disasters

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

Learning requires systematic exposure to failure

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

Design for degraded operation, not perfect performance

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

Chapter 12: When Good Enough Isn't

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

Who Decides What Level of Broken Is Acceptable?

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

The Invisible Moral Layer

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

The Degraded Mode Dilemma

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

The Gambler's Ethics

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

Making the Invisible Visible

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

Chapter 12 Key Takeaways

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

Technical design decisions always embed moral choices

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

Complex systems hide moral dimensions through abstraction

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

Degraded mode operation shifts moral responsibility without accountability

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

Apply the three-question framework to surface moral choices

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

Epilogue: Living with Inevitable Failure

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

The Pattern Beneath the Patterns

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

The Moral Thread

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

Building for the World We Actually Live In

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

The Future of Failure

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

The Wisdom of Inevitable Failure

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

Technical Appendix

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

Therac-25 Race Condition

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

The Software Architecture

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

The Race Condition: Timing-Critical Input Processing

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

The 8-Second Window Race Condition

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

The Turntable Position Overflow Bug

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

Why Traditional Testing Missed This

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

The Missing Safeguards

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

The Hardware Safety Elimination

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

Lessons for Concurrent Systems

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

Spectre Attack

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

How Speculative Execution Creates Attack Vectors

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

The Attack Mechanism

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

Why This Attack Works

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

The Scope of the Problem

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

CrowdStrike Channel File 291 Incident

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

The Technical Architecture

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

The Specific Failure: Channel File 291

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

Why This Caused Global Crashes

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

Border Gateway Protocol (BGP) Technical Details

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

BGP Message Types and Operation

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

Route Advertisement and Path Selection

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

BGP Path Attributes

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

BGP Path Selection Algorithm

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

iBGP vs eBGP Operation

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

BGP Security Vulnerabilities and Mitigations

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

Route Filtering and Policy Implementation

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

BGP Convergence and Stability

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