Chapter 1 - The Problem

Systems That Drift

1.1 Development Speed and Loss of Control

Software development was long constrained by the cost of production. Writing code, producing documentation, creating diagrams, tests, and architectural artifacts all required time, effort, and coordination. As a result, teams often made a practical compromise: if speed was needed, some structure was sacrificed. If clarity was needed, development slowed down.

With the rise of AI tools, that balance changes. The cost of production drops sharply. Code, specifications, tests, refactoring proposals, documentation, and diagrams can now be produced far faster than before. At first glance, this looks like a pure gain. And in many situations, it is.

But this is exactly where the new problem begins. When production becomes cheaper, the system's ability to remain controlled becomes more expensive and more important. It is no longer hard to produce a large volume of changes. The hard part is ensuring those changes remain aligned with the law of the system, its architecture, ownership model, and long-term sustainability.

In other words: speed is no longer the scarcest resource. Control is.

Once development becomes fast enough, a system can begin changing faster than a team can understand the consequences of those changes. Local decisions become permanent patterns. Temporary solutions become the architectural norm. Explanations that were never formalized become the basis for future modifications. The system does not break immediately. It continues to work. But it begins to lose its shape.

That is why this book does not begin with the question of how to make development even faster. It begins with the question of how to keep a system correct when speed is no longer the problem.

1.2 Implicit Knowledge as a Hidden Risk

Many systems survive for a long time thanks to knowledge that is never clearly written down anywhere. Some developer knows why something was done in exactly that way. Someone on the team remembers which layer truly owns a particular responsibility, even though it was never formally explained. Someone knows that a certain workaround must not be touched, even though there is no invariant anywhere that says so explicitly.

This implicit knowledge often looks like a practical advantage. It allows a team to move faster, without formalizing every decision. In small systems and stable teams, that can sometimes work.

But implicit knowledge has one serious problem: the system depends on it without actually possessing it.

As soon as there is meaningful growth, new team members, a shift in context, more parallel changes, or the introduction of AI agents, implicit knowledge stops being an advantage and becomes a risk. It is no longer a shortcut to understanding, but a source of uncertainty. A new actor in the system - whether human or agent - can no longer reliably distinguish:
- what is a rule
- what is an accidental decision
- what is a temporary workaround
- what is an invariant
- what is actually the source of truth

When that is unclear, the system begins to change on the basis of assumptions.

That is precisely why implicit knowledge is one of the most expensive forms of debt. It does not look like a technical problem until the system starts growing under pressure. But once that pressure arrives, it becomes obvious that the system does not contain enough of its own truth within itself; instead, it depends on people carrying that truth in their heads.

This book starts from the opposite idea: a system must possess enough of its own structure, laws, and traces of decisions to remain understandable under change, under growth, and under automation.

1.3 AI as an Amplifier of Existing Problems

AI in development is not primarily a problem because it is unreliable. Its greatest problem is that it very efficiently amplifies what already exists.

If a system has a clear specification, good architecture, clear boundaries, controlled context, and a good harness, AI can significantly accelerate work without sacrificing discipline. But if the system already rests on implicit knowledge, unclear ownership, duplicated logic, local shortcuts, and weak verifiability, then AI does not solve those problems. It multiplies them.

That is the real shift. In the past, a bad decision could remain limited because its spread was slow. Today, one unclear pattern can be:
- propagated across multiple files
- turned into a new norm
- documented as if it were correct

- further reinforced by tests that verify the wrong thing
- all faster than a human can recognize that the direction is wrong

AI is therefore not just a new tool. It is a multiplier. It amplifies good structure, but it also amplifies bad structure.

That means the question is no longer only whether a model can produce code. The question is: in what kind of system is it allowed to act?

That is where the core thesis of this book begins. If we want AI to be useful, we must not drop it into a chaotic system and expect it to magically clean things up. We first have to build a system that knows what is correct, where each responsibility belongs, how change is controlled, and how deviation is recognized. Only within such a framework does AI stop being a threat to structure and become an amplifier of discipline.

1.4 Local Correctness vs. System Correctness

One of the most dangerous illusions in development is the belief that something is correct if it works locally.

Local correctness means that a particular part of the system produces the expected result in a specific case. A screen renders. An API call returns a value. A queue looks up to date. A test passes. At the level of the immediate task, everything seems fine.

But system correctness demands more. It asks:
- did the change happen in the right layer
- did ownership remain clear
- was the source of truth preserved
- does the invariant still hold
- is the flow still understandable
- did the repair solve the cause rather than just the symptom
- will the next change be easier or harder because of this solution

Many systems lose control precisely because they confuse these two levels. A locally successful solution gets accepted as systemically correct even though it may have been introduced:
- in the wrong place
- under the wrong owner
- with no trace in the specification
- without protecting the invariant
- and with no mechanism that could later recognize that the system was weakened

At that point the system continues to work, but its correctness becomes more and more of an illusion.

This is one reason the book is not only about code. Code is often the place where a consequence becomes visible, but it is not always the place where the problem originated. For a system to remain correct, it is not enough for a local solution to work. It must remain within the law, the architecture, and a verifiable discipline of change.

1.5 Degradation as a Process

Systems rarely fail because of a single catastrophic decision. Much more often, they degrade through a sequence of small decisions that each look reasonable on their own.

One workaround. One duplicated logic path. One temporary exception. One local fix that bypasses the rightful owner. One insufficiently clear contract. One change that was never reflected back into the specification. One flow that nobody can clearly explain anymore. Each of these things seems small in isolation. Together, they change the system.

That is degradation as a process.

The danger of degradation is not only that things become uglier or less elegant. Its real danger is that the system loses the ability to remain under control. Over time, it becomes harder and harder to:
- understand where things belong
- distinguish cause from consequence
- introduce new functionality safely
- fix an issue without creating a new structural problem
- bring in a new person or agent without multiplying confusion

Degradation is therefore not the opposite of “good code.” It is the opposite of a controlled system.

That is the starting point of this book. If degradation is a process, then resistance to degradation must also be a process. Good intentions, talented people, or powerful tools are not enough. What is needed is a model in which the system:
- knows its own law
- has an architectural space
- has control over context
- has a harness that enforces discipline
- has agents that act within rules
- and has a way to turn failure into repair, and change into evolution, without losing identity

That is the problem this book wants to solve.