Problem Definition

The Problem SDPF Solves

The Core Problem

Software is built backwards.

In the overwhelming majority of software projects — from internal tools to enterprise platforms — code is written before requirements are complete, tests are written after code exists, changes are made informally, and no one can trace a defect back to the requirement that was supposed to prevent it.

This is not a skills problem. The developers are competent. The tools are mature. The problem is structural. There is no enforced contract between what a system is supposed to do and what it actually does. The gap between intent and implementation is filled with assumption, tribal knowledge, and hope.

The Five Failure Modes

1. Requirements exist only in someone's head.

A developer starts building. The requirements are in a Slack message, a verbal conversation, or a Jira ticket written in three sentences. No one has formally defined what inputs the system accepts, what it must do with them, what it must return, or what happens when something goes wrong. Two weeks later, the system does something no one expected — and no one can point to the document that says it should not.

2. Tests are an afterthought.

Testing is treated as something that happens after the code is written. This means tests are written to match the code that exists, not to verify the behaviour that was required. Bugs that exist in the original implementation are often preserved in the tests that were supposed to catch them. The test suite proves the code does what it does — not that it does what it should.

3. Changes are made without consequence tracking.

A requirement changes. Someone updates the code. No one identifies which other parts of the system were built against the old requirement. No one re-verifies the affected components. The change propagates silently. Six months later, something breaks in a way that cannot be explained without reading git history from two versions ago.

4. Code has no traceable justification.

A function exists. It does something. No one can say which requirement it was written to satisfy. In regulated environments, this is an audit failure. In any environment, it means that when the function needs to change, no one knows what it is safe to touch and what it is not. Refactoring becomes archaeology.

5. Technical facts are asserted but never verified.

The specification says the system uses Python 3.11. It does not. The specification says the API endpoint is at `/v1/users`. It was renamed in the last release. The specification says the compression tool is installed at `C:\upx`. It is not on the build server. These errors are invisible until the build fails, the deployment breaks, or the integration test collapses in an environment that is not the developer's machine.

Who This Affects

This problem does not discriminate by company size, industry, or engineering maturity. It affects:

Individual developers and small teams who lose days debugging systems they built themselves because requirements were never written down and the original intent has faded.

Enterprise engineering teams who cannot answer a simple question — "show me every piece of code that was written to satisfy this requirement" — without a manual audit that takes weeks.

Regulated industries — healthcare, finance, legal, government — where the inability to produce an auditable chain from requirement to implementation to test is not just an engineering problem. It is a compliance failure with legal and contractual consequences.

Teams using AI-assisted development where an AI model confidently generates code based on a version number, a flag, or an API parameter that no longer exists. The process was followed correctly. The output is broken. There was no mechanism to catch the error before it was embedded in the codebase.

Why Existing Solutions Are Insufficient

The problem is not new. The industry has tried to solve it.

Issue trackers (Jira, Linear, GitHub Issues) capture requirements as tickets but provide no structural link between a ticket and the code written to satisfy it. Closing a ticket does not mean the requirement was correctly implemented. It means someone marked it done.

Documentation tools (Confluence, Notion, wikis) capture requirements as prose but have no mechanism to enforce that the code matches the document. Documentation drifts from reality immediately and silently.

API specification tools (Swagger, OpenAPI) define contracts for API surfaces but do not govern the internal logic, error handling, state invariants, or test coverage of the system behind the API.

Testing frameworks (pytest, Jest, JUnit) verify that code does what it does, but do not verify that code does what it was supposed to do. A test suite with 100% coverage can pass while the system violates every requirement it was built to satisfy, if the tests were written to match the implementation rather than the specification.

Linters and static analysis tools (ESLint, pylint, SonarQube) catch code quality issues and common error patterns but have no concept of requirements, traceability, or specification compliance.

None of these tools enforce the relationship between a requirement and its implementation. They operate on code in isolation. The gap between intent and implementation remains open.

The Specific Gap

What is missing is a methodology with teeth — one that:

- Forces requirements to be completely specified before implementation begins
- Names every requirement so it can be referenced, tracked, and traced
- Defines tests before code is written, not after
- Verifies every technical fact asserted in the specification against live reality before a build begins
- Classifies every change to a requirement by its impact and identifies every artifact that must be rebuilt
- Produces a machine-readable map from every requirement to its implementation to its test
- Refuses to allow a release until every verification item has a recorded result
- Enforces all of the above automatically, not through discipline or culture

The gap is not a missing tool. It is a missing contract — a formal, enforceable agreement between what a system is supposed to do and what it is permitted to do.

The Consequence of Not Solving It

In low-stakes environments, the consequence is wasted time — debugging, rework, and the slow accumulation of technical debt that eventually makes the system unmaintainable.

In high-stakes environments, the consequences are more severe. A healthcare system that cannot prove its behaviour matches its specification is a liability. A financial platform that cannot produce an audit trail from requirement to implementation cannot pass a regulatory review. A security-critical system with untraced code contains attack surface that no one knows exists.

Across all environments, the consequence is the same: software that cannot be trusted, cannot be audited, and cannot be safely changed.

The Solution

SDPF addresses this problem at its root by making specification a structural prerequisite — not a recommendation. It provides a complete methodology for capturing requirements, a formal prompt structure that encodes those requirements in a machine-readable and human-readable contract, a toolchain that enforces the contract automatically, and a platform that operationalises both at scale.

The result is software that is not just functional — but correct by construction, auditable by design, and safe to change because every change is traceable, classified, and verified.

The problem is not that developers do not care about requirements. The problem is that the industry has never built a system that makes it structurally impossible to ignore them.