SDPF

Why Engineering Intent Becomes

Structurally Impossible to Ignore

How SDPF embeds original engineering intent into every artifact, gate, and gate check — making drift structurally impossible rather than merely discouraged.

Hamza Abdullah

Software Development Prompting Framework · Version 2.2 · 2026

The Problem With Every Other Approach

In conventional software development, engineering intent lives in documents — requirements specifications, wiki pages, Jira tickets, architecture decision records. These documents describe what the system should do. The code describes what the system actually does. The relationship between the two is maintained entirely by human discipline and memory.

Human discipline fails. Memory fades. The document that described the intent gets out of date. The developer who understood the original requirement leaves the team. The ticket gets closed and forgotten. Six months later, nobody can confidently say whether the current implementation matches the original intent — because the original intent and the implementation live in completely separate places, connected by nothing structural.

This is not a failure of individual developers or teams. It is a structural failure of the methodology. When the connection between intent and implementation depends on human effort to maintain, the connection will eventually break. It always does. Under deadline pressure, through staff turnover, across version boundaries, through scope changes — the gap opens slowly, silently, until it becomes a crisis.

THE PROBLEM

Intent and implementation live in separate places, connected by human discipline alone. Human discipline fails. The gap is inevitable under every conventional methodology.

SDPF does something fundamentally different. It does not ask people to maintain the connection between intent and implementation. It makes that connection structural — embedded in the artifacts themselves, enforced by the tool, and impossible to sever without breaking the process entirely.

This document explains exactly how that structural embedding works. There are five distinct mechanisms. Each one independently constrains the ability to drift from intent. Together they make drift structurally impossible rather than merely discouraged.

1 MECHANISM

The Specification Is Locked Before Anything Else Exists

Intent is the cause, not a record of the effect

In most methodologies, requirements documentation is produced either before development (and then ignored) or after development (as a description of what was built). In both cases the documentation is separate from the causal chain that produced the system. It describes the system but does not cause it.

SDPF inverts this relationship. The specification is not a description of the intended system. It is the prerequisite gate that causes the system to exist.

Conventional Methodology	SDPF Methodology
Intent documented (often ignored)	Intent locked — gates blocked until complete
Code written based on interpretation	Tests generated from locked intent
Documentation updated to match code	Implementation generated from locked tests
Intent and code drift apart over time	Intent is the root — everything derives from it
Documentation describes what exists	Specification causes what will exist

The SDPF tool enforces this causality mechanically. The Generate Tests button raises a hard error if no specification is locked. The Generate Implementation button raises a hard error if tests are not locked. The causal arrow runs in one direction only: intent → tests → implementation. There is no path from implementation back to undocumented intent.

This means the engineering intent is not recorded after the fact — it is recorded first, before any code exists, and everything that follows derives from it. The intent is not a description of the system. It is the foundation the system is built on.

WHY THIS MATTERS

When intent is the cause rather than the record, it cannot be ignored without stopping the process entirely. You cannot proceed without it. It is not optional documentation — it is the prerequisite for everything else.

2
MECHANISM

Every Requirement Receives a Permanent Identity

REQ-IDs create an unbreakable chain from intent to artifact

When a specification is locked, SDPF parses every tagged requirement line and assigns it a permanent identifier. R-001, R-002, R-003. These identifiers are not labels — they are anchors in a chain that runs through every downstream artifact.

Artifact	How the REQ-ID Appears
Test Suite	Every test body includes # Trace: R-xxx
Implementation	Every function docstring and comment includes R-xxx
Traceability Matrix	REQ-ID → Test ID → Implementation Hash
Evidence Package	Full traceability rows in signed JSON

The consequence is that every piece of the system can be traced back to the exact requirement that justified its existence. Not approximately. Not based on memory or inference. Exactly.

If a function exists in the implementation but has no REQ-ID, that function is untraced — it was added outside the specification-governed process. The traceability matrix makes this visible immediately. An untraced function is a visible deviation from the original engineering intent.

If a requirement has no tests, the verification gate fails the traceability_complete check. The run cannot be marked as verified. The incompleteness is not a warning — it is a hard block on the process.

```
def req_r_007_critical_all_write_operations_produce_audit_entries(payload:
dict) -> dict:
    """Trace: R-007 | CRITICAL"""
    # Requirement: All write operations produce immutable audit entries
    # Style Engine Impl: compliance_trace_engine
    ...
```

WHY THIS MATTERS

The REQ-ID makes the intent permanently visible in every artifact it produces. Intent and artifact are not two separate things maintained by human effort — they are causally linked by a persistent identifier that survives refactoring, staff changes, and version upgrades.

3
MECHANISM

The Gate Sequence Is Enforced in Code, Not Policy

Structural impossibility replaces disciplinary requirement

This is the most important mechanism. It is the one that makes SDPF fundamentally different from every methodology that came before it.

Most methodologies define a gate sequence as a process rule. "Requirements must be approved before development begins." "Code must pass review before release." "Tests must exist before code is shipped." These rules are correct. They are also violated constantly, under every real-world pressure that software teams face. They are violated because the tool does not prevent the violation.

SDPF's gate sequence is not a process rule. It is enforced by the application itself, in code, as hard errors that cannot be bypassed.

```
def generate_tests(self) -> List[TestCase]:
    if self.state.stage not in {Stage.SPEC_LOCKED, Stage.TESTS_GENERATED}:
        raise RuntimeError("Tests can only be generated after the specification
is locked.")

def generate_implementation(self, module_name: str = "sdpf_module") -> str:
    if self.state.stage != Stage.TESTS_LOCKED:
        raise RuntimeError("Implementation can only be generated after tests
are locked.")

def verify(self) -> VerificationEvidence:
    if self.state.stage != Stage.CODE_GENERATED:
        raise RuntimeError("Verification is only available after implementation
generation.")
```

These are not warnings. They are not advisory messages. They are RuntimeErrors that stop execution. The UI surfaces them as categorised error dialogs that explain the required sequence. The buttons are disabled when the stage does not permit the action.

There is no override. There is no "skip this step" option. There is no admin mode that bypasses the gates. The only way to perform an action is to satisfy the prerequisites — which means completing the upstream steps that derive from the original engineering intent.

What Process Rules Prevent	What Code Enforcement Prevents
Deviations when everyone is disciplined	Deviations regardless of discipline
Violations that are noticed and corrected	Violations that are structurally impossible
Drift that accumulates slowly	Drift that cannot begin without breaking the tool

What Process Rules Prevent	What Code Enforcement Prevents
Non-compliance that must be audited for	Non-compliance that produces tool errors, not audit findings

WHY THIS MATTERS

When the gate sequence is enforced in code, the engineering intent cannot be ignored under pressure, through laziness, or by mistake. It cannot be ignored at all — because ignoring it means the tool stops working. The process and the tool are the same thing.

4
MECHANISM

Eleven Verification Checks Block Release Until Intent Is Satisfied

Completeness is enforced, not assumed

Even after a developer has followed every step — locked the specification, generated tests, locked tests, generated implementation — the verification gate runs eleven checks before the run can advance to VERIFIED status. Every check must pass. A single failure holds the stage at CODE_GENERATED and blocks evidence export.

Check	What It Enforces About Engineering Intent
spec_exists	The intent was formally captured — a specification exists
style_defined	A style was chosen — the intent was classified into a known domain
tests_exist	The intent produced tests — it was not specified and then bypassed
tests_locked	Tests were frozen before implementation — intent governed the test suite
implementation_exists	Implementation was produced after the intent was locked
traceability_complete	Every requirement has at least one test — no intent line was skipped
style_constraints_present	The style's requirement marker appears in the spec — style was applied correctly
style_N_engine_verified	Style-specific engine tags appear in tests and code — style governed the artifacts
policy_engine_passed	Specification content satisfies the semantic rules for the chosen style
ci_gate_ready	A CI workflow was generated — intent can be enforced in automated pipelines
provenance_signing_ready	A signing key is configured — the evidence can be tamper-proofed

The most significant of these for engineering intent is traceability_complete. This check answers the question: is every requirement that was specified actually covered by a test? If R-012 was specified but no test references R-012, the check fails. The intent expressed in R-012 is not considered satisfied. The run is not complete.

The policy_engine_passed check adds a content layer. For Style 10 (Compliance-Driven), the engine checks whether the specification contains the keywords required for that domain — regulatory, audit, and related terms. A specification that claims to be compliance-driven but

contains no compliance language fails this check. The intent must match the style, not just the label.

WHY THIS MATTERS

Verification does not assume completeness — it proves it. Every gap between intent and artifact is a failed check. Every failed check blocks the run. The engineering intent must be fully satisfied before the process is considered complete.

5
MECHANISM

The Signed Evidence Package Makes Intent Tamper-Evident

The original intent cannot be altered retroactively

The final mechanism operates at the boundary between the SDPF process and the external world — clients, auditors, regulators, future developers. When a run is complete and evidence is exported, the entire record of the run is packaged into a single JSON document and signed with HMAC-SHA256.

The signature is computed over the canonical payload — the specification, requirements, tests, implementation hash, verification checks, traceability matrix, event log, AI results, and model comparison — before the signature itself is added. The verification procedure is documented inside the package so anyone can verify it independently.

```
"provenance": {
  "algorithm": "HMAC-SHA256",
  "key_id": "env:SDPF_SIGNING_KEY:v1",
  "signed_at": "2026-04-10T14:23:00Z",
  "signature": "a3f8c2d19b4e...",
  "verification_procedure": "Recompute HMAC-SHA256 over canonical payload
    excluding provenance field and compare signatures."
}
```

The consequence of this signature is precise and important: the original engineering intent, exactly as it was specified before any code was written, is permanently recorded in a tamper-evident document. Any change to any field — a requirement reworded, a failed check altered to pass, a test added retroactively, an AI result inserted after the fact — breaks the signature.

This means the evidence package is not just a record of what happened. It is a proof that it could not have been altered after the fact. The intent captured in the specification is the same intent that governed the tests, that governed the implementation, that was verified by the eleven checks. The chain is unbroken and the proof of its unbroken state is the signature.

Without the Signature	With the Signature
Evidence is a document — alterable by anyone with file access	Evidence is tamper-evident — any change breaks verification
Requirements could be reworded retroactively to match code	Requirements are frozen at signing time — retroactive changes are detectable
Failed checks could be manually set to passed	Check results are part of the signed payload — unalterable
AI results could be added or removed	AI results are signed — the record of what was submitted is permanent

Without the Signature	With the Signature
The chain from intent to evidence relies on trust	The chain from intent to evidence is cryptographically enforced

WHY THIS MATTERS

The signature closes the loop. The engineering intent that started the process is the same engineering intent recorded in the evidence. No one can change the intent after the fact and claim the evidence still represents it. The original intent is not just preserved — it is cryptographically frozen.

Why Five Mechanisms and Not One

Each mechanism independently constrains the ability to drift from intent. Together they create a system of overlapping constraints where every path away from intent is blocked by at least one mechanism, and most paths are blocked by several.

Attempted Drift	Blocked By
Skipping specification entirely	Mechanism 1 — gates cannot open without a locked spec
Writing code that ignores a requirement	Mechanism 2 — untraced code is visible; Mechanism 4 — traceability check fails
Skipping tests to get to implementation faster	Mechanism 3 — implementation gate will not open without locked tests
Shipping without verification	Mechanism 3 — export requires VERIFIED stage; Mechanism 4 — 11 checks must pass
Altering requirements after implementation	Mechanism 5 — retroactive changes break the signature
Claiming compliance without compliance content	Mechanism 4 — policy engine checks spec content against style rules
Producing evidence for an incomplete run	Mechanisms 3 and 4 — stage must be VERIFIED; all checks must pass

Notice that no single mechanism is sufficient on its own. If only the gate sequence existed, someone could lock a trivial specification and proceed. If only traceability existed, someone could add trivial tests that reference every REQ-ID without actually covering the intent. If only the signature existed, someone could produce a signed package for a poorly specified, poorly traced run.

The five mechanisms work together precisely because each one closes a gap that the others leave open. The specification lock ensures intent exists. The REQ-IDs make intent traceable. The gate enforcement makes the sequence mandatory. The verification checks make completeness mandatory. The signature makes the record permanent. Remove any one and a gap opens. All five together and the gap closes.

The Deeper Reason This Matters

Intent Drift Is Invisible Until It Is a Crisis

In most projects the distance between what was intended and what was built grows continuously over time, silently. Requirements are forgotten. Scope expands informally. Edge cases are handled inconsistently across different parts of the system. Features are added that have no requirement backing them. Features that were required are partially implemented or quietly dropped.

By the time someone asks "does this system do what we specified," the honest answer is often "we are not sure." Not because anyone was negligent. But because the methodology provided no structural mechanism to maintain the connection between intent and implementation as the project evolved.

SDPF closes that distance at every step. Every change to the system requires a change to the specification first. The specification change is governed by the same gate sequence. The change is classified — BREAKING, SIGNIFICANT, or MINOR — and its downstream impact is identified before the change is made. Intent drift cannot accumulate silently because the process makes every departure from the original intent visible as a formal change event.

Systems Age Better When Intent Is Structural

The long-term value of SDPF is not visible on a single project. It is visible when maintaining, extending, or handing off a system that was built under SDPF.

A developer who joins a SDPF-governed project after the original team has left does not have to reconstruct the intent from code comments, git history, and institutional memory. The evidence package contains the locked specification — the original intent, exactly as it was when the system was built. The traceability matrix shows which requirement produced which code. The style selection shows what domain the system was built for.

They can answer the question "why does this work this way" by reading a signed document rather than reconstructing it from circumstantial evidence. The original engineering intent has not drifted because it was structurally prevented from drifting.

The AI Era Makes This More Important, Not Less

As AI-assisted development becomes standard practice, the gap between intent and implementation threatens to grow rather than shrink. An AI that receives a vague prompt produces a vague implementation. The developer who pastes the output into their codebase may not fully understand what the AI produced. The intent that was in the developer's head — the intent that was never formally specified — is never captured anywhere.

SDPF closes this gap for AI-assisted development specifically. The intent is locked before the AI sees it. The AI output is captured and checked against the REQ-IDs from the specification. The evidence package records which model produced the output, which requirements it covered, and how it compared to other models.

The result is that AI-assisted development, under SDPF, is more auditable than conventional human development without SDPF — because the intent is formally specified, the AI output is formally captured, and the relationship between them is formally recorded and signed.

THE RESULT

Under SDPF, the original engineering intent is not a document that describes a system. It is the root of a signed, traced, verified chain from which every artifact in the system derives. You cannot remove it without removing the system. You cannot change it without the change being recorded. You cannot ignore it without the tool stopping you. That is what structurally impossible to ignore means.

Specification first. Code second. Verification always.

SDPF — Software Development Prompting Framework · Hamza Abdullah · 2026