The Technical Verification Gate

Where the Framework Meets Reality

Hamza Abdullah

2026

The Problem It Solves

Generative AI models, when asked to produce or assist with a specification, assert technical facts with confidence. They name version numbers. They reference API endpoints. They cite command-line flags. They describe environment variables. They do this fluently, and the fluency is the trap.

The model is not lying. It is pattern-matching from training data that has drifted out of date — and sometimes from training data that was wrong to begin with. The version number it produces was correct eighteen months ago. The flag exists in a neighbouring tool but not in the one being specified. The endpoint was deprecated two releases back. The environment variable has been renamed. The model cannot know this, and in the absence of a mechanism that forces it to check, the model will assert the stale fact as confidently as the current one.

The result is a specification that looks complete, passes every structural check, survives review, and produces code that fails on contact with reality. The specification was internally consistent. The code was faithfully derived from the specification. The failure came from a factual assumption embedded in both, which neither the specification discipline nor the implementation discipline was positioned to catch.

The first version of SDPF had no defence against this. The framework constrained process but not facts. A prompt could follow every SDPF rule correctly and still embed a wrong version number, a broken flag syntax, a stale endpoint, or a non-existent library. The build would fail for reasons that had nothing to do with the specification's structure — and the framework would appear to have worked correctly while producing broken software.

The Technical Verification Gate is the mechanism that closes this gap. It is the framework's answer to knowledge hallucination — the failure mode that becomes dominant the moment generative AI enters the specification loop.

What the TVG Is

The TVG is a mandatory section in every SDPF specification. It lists every technical fact the specification depends on, and it forces each one to be verified against live output before the specification can be locked.

Every asserted technical fact appears as a TVG entry with four components: the asset name and its asserted value, a verification command that can be executed against the live environment, a pass condition defining what the output must contain, and a HALT rule specifying what happens on failure.

The structure is deliberately mechanical. Here is an entry for a Python version assertion:

```
Tool: Python

Asserted Version: 3.11.15

Verification: python --version

Pass Condition: output contains "Python 3.11.15"

HALT: Do not proceed under any other version
```

When the tool runs the verification command and the output does not match the pass condition, the specification cannot be locked. The build does not start. Work halts until the specification is corrected to reflect what is actually present in the environment — or the environment is corrected to match what the specification requires. Either resolution is acceptable. What is not acceptable is proceeding with a mismatch.

This applies to every asserted technical fact, not just version numbers: tool versions, library versions, API endpoints, flags, file paths, environment variables, authentication mechanisms, protocol versions. Anything the specification asserts about the world the code will run in must appear in the TVG, and each entry must pass before the specification is valid.

The HALT Discipline

The most important rule in the TVG is not about what to check. It is about what to do when a check fails.

> *If any TVG check fails, the specification shall be updated before implementation proceeds. Work shall not continue around a TVG failure.*

This is a policy stance, not a technical constraint. Nothing prevents a team from ignoring a failed TVG check. The discipline is that a conforming tool surfaces the failure as a blocking error, and a conforming practitioner treats it as a specification problem rather than a build problem.

The distinction matters. In most engineering cultures, a version mismatch is handled at the build level: the team patches around it, updates the dependency, adds a compatibility shim, or simply lowers the asserted version to match what is installed. The specification is treated as advisory, the implementation as authoritative. Whatever it takes to get the build green is considered success.

The TVG reverses this relationship. The specification is authoritative. If it asserts Python 3.11.15 and the environment has Python 3.11.12, one of those is wrong — and the fix is to decide which one, then update that one, rather than quietly papering over the gap. If Python 3.11.15 is genuinely required, the environment must be upgraded. If 3.11.12 is acceptable, the specification must be changed to say so. Either way, the specification and the reality it describes are brought into alignment before any code is written against the mismatch.

This discipline prevents the most common failure mode in AI-assisted development: the model produces plausible-but-wrong technical assertions, the human accepts them because they look right, and the discrepancy surfaces much later when something expensive breaks. By the time the failure is visible, the specification has been locked, tests have been written, code has been generated, and the cost of correction is proportional to the depth of the stack built on the false assumption. The TVG makes that depth zero.

Where It Sits in the Lifecycle

The TVG is positioned as a prerequisite for locking the specification, not as a check performed after implementation. This placement is structural and intentional.

In the SDPF lifecycle, a specification moves through six stages: draft, specification locked, tests generated, tests locked, code generated, verified. The transition from draft to locked has three preconditions: the specification validates structurally, there are no unresolved conflicts between critical requirements, and every TVG entry has passed. If the TVG fails, the specification stays in draft. Nothing downstream can begin.

This is the difference between a gate and a check. A check runs after the fact and reports findings, which the team then decides how to handle. A gate runs before the fact and blocks progress until the condition is met. The TVG is structurally a gate — positioned before tests are generated, before implementation begins, before the AI is asked to produce code. The hallucination, if it exists, is caught at the specification layer, where correction is cheap, rather than at the deployment layer, where correction is expensive.

The Language Specification states this normatively: a specification shall not be locked until all TVG entries have been verified against live output and have passed. A conforming tool shall surface TVG failures as blocking errors. These are not suggestions. They are requirements a tool must satisfy to be considered SDPF-conforming.

Why It Is a Core Principle

SDPF has three core principles. Specification First. Technical Verification Gate. Verification Always.

The TVG is co-equal with the other two, and the elevation was deliberate. Earlier versions of the framework treated the TVG as a technique within the specification discipline. Version 1.1 promoted it to principle status because the framework's authors recognised that specification completeness alone is not sufficient. A complete specification that asserts false facts produces confident failure. The symptoms look exactly like correct operation — the code compiles, the tests pass, the documentation is coherent — until the system meets the reality the specification misrepresented, at which point it fails in ways that are hard to diagnose precisely because the specification looked so clean.

Specification First says nothing is built without a complete contract. Verification Always says nothing ships without passing verification. Neither of these covers the gap that the TVG addresses, which is the gap between what the contract asserts about the world and what the world actually is. Without the TVG, the other two principles can be followed perfectly and still produce broken software. With it, the framework has a principled answer to the question of why its specifications can be trusted to describe real systems rather than imagined ones.

What the TVG Catches That Nothing Else Does

Traditional testing catches wrong behaviour after the code exists. Static analysis catches certain classes of error in the code itself. Schema validation catches malformed specifications. Linting catches style violations and common bug patterns. Contract testing catches mismatches between services. Each of these is valuable and none of them catch what the TVG catches.

The TVG's territory is the gap between what the specification asserts about the world and what the world actually is. Every existing tool assumes the specification's factual claims are correct and operates within that assumption. The TVG is the mechanism that tests the assumption itself.

This is a narrow territory, but it is the territory where AI-assisted development fails most often. When a human writes a specification, factual errors tend to correlate with the author's unfamiliarity with the domain, and they are relatively bounded — a developer who has used Python 3.11 for a year is unlikely to assert a version that does not exist. When an AI writes or assists with a specification, factual errors are uncorrelated with anything legible. The model asserts whatever its training data suggests, with no internal signal distinguishing a current fact from one that was true in its training window. The TVG provides that signal externally, by requiring every assertion to be checked against live output before the specification can lock.

On Uniqueness

The TVG, like many of SDPF's contributions, is not conceptually novel at the level of underlying idea. Verifying assumptions against reality is something the software industry has been doing for decades, in various forms.

Infrastructure-as-code tools perform drift detection, comparing asserted state to actual state. Dependency resolvers verify declared versions against available ones. CI pipelines routinely include environment validation steps. Contract testing tools verify that asserted API shapes match real service behaviour. Doctest and similar mechanisms verify that documented examples still work. Each of these implements a piece of the underlying practice.

What is unique to the TVG is the combination of four properties that none of the prior art assembles together.

Specification-level positioning

Existing tools check assumptions at build time or deploy time — after the specification exists, while the code is being assembled or run. The TVG operates earlier, as a prerequisite for the specification itself being considered valid. This is a different location in the lifecycle than any existing tool occupies, and it is the location that matters when an AI is involved in producing the specification.

Normative mandatory status

Every existing mechanism is optional tooling that teams choose to adopt. The TVG is a normative requirement in a formal specification — not an add-on, not a best practice, but a conformance requirement. A specification without TVG entries for every asserted technical fact is not a valid SDPF specification. This elevates the practice from good engineering hygiene to formal conformance.

Specific targeting of AI hallucination

Prior tools were designed for human-authored systems where factual drift came from age, documentation lag, or environmental variance. The TVG targets a failure mode that only becomes dominant when an AI is generating or assisting with the specification: confident assertion of facts pattern-matched from outdated training data. No prior tool was designed with that specific failure mode in mind, because no prior tool operated in a context where the failure mode existed at scale.

The HALT discipline

Most existing tools report failures and leave the handling to the team. The TVG explicitly forbids working around a failure: the specification must be fixed, not the implementation. That policy — failures propagate back to the specification rather than being patched around downstream — is

a stance existing tools do not take, because existing tools do not treat the specification as the authoritative artefact.

Each property on its own is present in some prior tool. The combination is not. The TVG is the first mechanism that assembles specification-level positioning, normative status, AI-hallucination targeting, and a HALT discipline into a single named construct inside a formal framework's conformance requirements. That is a real contribution, and it is the contribution that makes SDPF workable on AI-assisted projects in a way that no existing tool chain supports.

The Short Version

The TVG is where the framework meets reality. Everything else in SDPF governs the relationship between intent and implementation. The TVG governs the relationship between the specification and the physical world the specification will run in.

Without it, a perfectly structured specification can produce broken software, because the specification's assertions about the world were never checked against the world. With it, the specification cannot lock until every technical assertion it depends on has been checked against live output — and the checking is mandatory, positioned before implementation, targeted at AI's specific failure mode, and structured so that failures propagate back to where they can be fixed cheaply.

That is the whole mechanism. It is the reason SDPF works on systems that connect to actual tools, actual APIs, and actual environments rather than abstract ideal ones. And it is the reason specifications produced with AI assistance can be trusted to describe the world the code will run in, rather than the world the model was trained to believe existed.