SDPF

SOFTWARE DEVELOPMENT PROMPTING FRAMEWORK

Governed AI Execution:

The Specification as the Governing Contract

WHITE PAPER

Hamza Abdullah
Creator and Principal Author, SDPF

2026

Executive Summary

The problem is not the AI model. The problem is the specification.

Generative AI is producing software at scale. That software is also failing at scale — not because the models are defective, but because the instructions given to them are incomplete. When a specification leaves gaps, an AI model fills them with its best guess. The guess is plausible. It is not what was intended. What the industry calls hallucination is, in the overwhelming majority of practical cases, specification-induced speculation.

SDPF — the Software Development Prompting Framework — is the formal answer to this problem. It defines how to write specifications that give AI models nothing to guess about, how to verify that every technical fact in a specification is true before any code is generated, how to produce a complete, signed, tamper-evident audit trail from business problem through requirement through implementation through verification, and how to enforce the specification's authority not only at generation time but throughout the operational lifetime of the system.

SDPF is not a prompt template. It is not a workflow tool. It is a formal specification language grounded in a foundational theorem — the Bounded Stochasticity Theorem — which establishes the conditions under which a deterministic specification can govern a stochastic generator. That theorem is the load-bearing element of the entire framework. Everything else derives from it.

What SDPF Provides	For Whom
A formal specification language with 17 normative dialects	Developers and architects building AI-generated software
Mandatory problem definition before any specification begins	Teams who need requirements to trace to real business problems
Technical Verification Gate — every fact verified before execution	Anyone who has been burned by AI confidently asserting wrong facts
Signed, tamper-evident evidence packages from every run	Regulated organisations needing auditable AI development
A runtime architecture for continuous post-deployment governance	Organisations deploying AI in safety-critical or regulated domains
A formal standard suitable for certification and adoption	Standards bodies seeking a governance framework for AI-generated software

01

The Problem

Why AI-assisted development fails — and why the industry has misidentified the cause

The software industry has spent the last three years arguing about AI models. Which model is more accurate. Which model hallucinates less. Which benchmark best predicts real-world performance. These are not the wrong questions — but they are the wrong level of analysis for the problem that actually occurs in practice.

The problem that occurs in practice is this: a developer gives an AI model an incomplete instruction, the model fills the gaps with its best guess, and the result does not match what was intended. This is attributed to hallucination. It is not hallucination. Hallucination is the confident assertion of facts that are not true. What happens in most AI-assisted software development is something different and more tractable: the model is given an instruction that does not fully specify the intended behavior, and it completes the instruction as instructed — faithfully, plausibly, and incorrectly.

> *A better model makes better guesses. It is still guessing. The correct intervention is not a better model. It is a complete specification — one where there is nothing to guess.*

This is not a new problem. It is the oldest problem in software engineering: the gap between intent and implementation. AI makes it faster and more visible. The gap that used to take weeks to open now opens in seconds, at scale, across every generated artifact.

The Five Failure Modes

SDPF identifies five structural failure modes in AI-assisted software development. These are not edge cases. They are the default state of a development process that has no enforced contract between what a system is supposed to do and what it does.

Failure Mode	What Happens
Requirements in someone's head	The developer starts building. Requirements exist in a Slack message or a verbal conversation. No one has formally defined what the system must accept, process, or return. The AI guesses the rest.
Tests after code	Tests are written to match the code that exists, not to verify the behavior that was required. Bugs in the original implementation are preserved in the tests that were supposed to catch them.

Changes without tracing	A requirement changes. The code is updated. No one identifies which other parts of the system depended on the old requirement. The change propagates silently.
Untraced code	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. Everywhere else it is technical debt waiting to become a crisis.
No governance after deployment	The system passes verification. It is deployed. Three weeks later a dependency update or a database migration silently moves the system outside its specification. Nobody knows.

Each of these failures has a structural cause — the absence of an enforced contract between specification and implementation. SDPF closes each one.

02

The Foundational Theorem

Why SDPF works — and the formal result that makes it defensible

Every serious framework rests on a formal property that makes it defensible to people who were not already convinced. Codd's relational model rested on his theorems about normal forms. TCP/IP rested on the end-to-end argument. Hoare logic underpins design by contract. Without the formal property, a framework is one methodology among several. With it, the framework becomes the justified choice.

SDPF rests on the Bounded Stochasticity Theorem. This theorem resolves what the industry had been treating as an irresolvable contradiction: AI models are stochastic — they produce different outputs from the same input — yet SDPF claims that deterministic specifications can govern them. How?

THE BOUNDED STOCHASTICITY THEOREM

Let S be a complete SDPF specification defining structural invariants I. Let G be a generative AI output produced from S. G satisfies S if and only if G satisfies all invariants in I. Stochastic variation in G is bounded by I — any variation satisfying I is a conforming realisation of S.

The theorem does not claim the AI is deterministic. It claims the acceptance criterion is. Determinism shifts from the generator to the boundary around the generator. A specification does not need to produce identical outputs — it needs to produce outputs that satisfy all formally defined invariants. Variation in surface form is acceptable. Variation that violates invariants is not.

What the Theorem Makes Possible

The theorem reopens a door the industry had closed. When AI models became the executor of software specifications, the formal methods tradition appeared to stop applying — a stochastic generator cannot, by the classical reading, satisfy a deterministic contract. The industry's reaction was to abandon specification in favour of evaluation: benchmarks, red-teaming, statistical acceptance.

The Bounded Stochasticity Theorem says this abandonment was premature. The contradiction between a deterministic methodology and a stochastic executor was never real. It was the product of specifying the wrong thing. Specify what the output must preserve

— structural invariants — not what the output must be, and the stochasticity becomes acceptable variation within bounded invariants.

Without the theorem	With the theorem
AI output is unverifiable in principle	AI output is verifiable against structural invariants
Governance is advisory (prompts, system messages)	Governance is structural (invariant satisfaction or not)
Audit trail cannot be produced	Complete, traceable, signed evidence package produced automatically
Regulated industries cannot adopt AI	Regulated industries have a defensible governance standard
The framework is one methodology among several	The framework is the justified choice for any organisation that needs defensible AI governance

The Scope Conditions

The theorem has explicit scope conditions. It does not hold when the specification is incomplete — which is why incompleteness is a first-class failure mode and Phase 0 problem definition is mandatory. It does not hold when technical facts in the specification are unverified — which is why the Technical Verification Gate is a hard gate, not a recommendation. It does not hold when the model lacks sufficient capability to satisfy the invariants — which means capability is a measurable precondition, not an assumption.

These scope conditions are features, not weaknesses. A theorem whose scope is everything is a theorem whose scope is nothing. These conditions tell any implementer exactly when they can rely on SDPF and exactly when they cannot.

03

The Framework

What SDPF defines and how it works

SDPF is a formal specification language. It defines the vocabulary, grammar, lifecycle, verification model, evidence standard, and conformance requirements for AI-governed software development. It is defined with sufficient precision that any practitioner, tool developer, certification body, or standards organisation can implement, evaluate, or adopt it without access to the original author.

Three Core Principles

Every SDPF-governed process is governed by three principles. They are not recommendations. They are hard gates. No stage may advance until the gate condition is satisfied.

Principle	What It Enforces
Specification First	No tests, no implementation, no AI submission until a complete, locked specification exists. The specification is the contract. Everything else executes it.
Facts Before Execution	Every technical fact the specification asserts — tool versions, API endpoints, database schemas, environment variables — must be verified by running a command in the actual target environment before implementation begins. If the fact is wrong, the specification is corrected. Nothing proceeds around a failed check.
Verification Always	Every run ends with a verification gate. All eleven structural invariant checks must pass. All tests must pass. CLOSURE STATUS must be COMPLETE. The run is not finished until they are.

Phase 0 — Problem Definition Before Specification

The most common cause of a failed specification is that the specification was written for the wrong problem. Someone decided they needed a login system and immediately began specifying the login system. Nobody asked: what problem does this solve? What is the measurable gap between the current state and the desired state?

SDPF makes Phase 0 mandatory. Before a single requirement is written, the practitioner must produce a validated problem statement containing four components: the observable current state, the measurable desired state, the quantified gap between them, and the

business impact of the gap's existence. Four validation tests must pass — the statement must be observable, bounded, cause-free, and solution-free. A problem owner must be named.

Only then may Phase 1 begin. The result is that every requirement in the specification can be traced to a real, validated problem — and the evidence package demonstrates that traceability.

The Technical Verification Gate

The TVG is the most important enforcement mechanism in SDPF. It exists because AI models confidently assert technical facts that are not true — the wrong library version, a non-existent API endpoint, a schema that was modified since the model's training data was collected. Specifications built on wrong facts generate code that does not work in the actual environment.

The TVG requires that every technical fact asserted in the specification has a corresponding verification entry: the assertion, the command to check it in the live environment, and the expected output. Every entry must be run and must pass before the specification can be locked. If any entry fails, the specification is corrected to match reality — or the environment is corrected to match the specification — and the check is re-run. Nothing proceeds around a failure.

> *A TVG failure is a specification error, not a build problem. The specification asserted a fact that is not true. Correct the specification. Then proceed.*

The Specification — 17 Normative Styles

SDPF defines 17 normative specification styles — formally defined dialects matched to system class. Style 1 governs deterministic APIs. Style 8 governs constraint-based embedded systems. Style 10 governs regulated, compliance-driven systems with mandatory regulatory mapping sections. Style 14 governs safety-critical systems where escalation is bound to runtime state. Each style changes the required sections, the AI framing, the semantic policy checks, the verification focus, and the evidence shape.

Every specification declares exactly one style. The style is selected before any requirement is written, because the style governs how the specification is structured and what the AI model is told about its role. Style selection is the first architectural decision on every project.

The Verification Model

Verification in SDPF is not a post-generation inspection step. It is a structured gate with eleven named invariant checks that must all pass before a run can advance to VERIFIED status. The checks are individually identified and individually referenceable. A conforming tool evaluates all eleven on every gate execution. A single failure is blocking.

The verification gate enforces: that a specification existed before execution, that tests were generated from the specification before implementation, that every requirement has at least one test, that the style was applied correctly, that the policy engine is satisfied, that a CI workflow exists, and that a provenance signing key is configured. These checks collectively make it structurally impossible to skip any step in the SDPF discipline.

The Evidence Package

Every verified run produces a signed, tamper-evident evidence package automatically. It contains the validated problem statement, the locked specification with all requirement identifiers, all tests with traceability to requirements, the implementation hash, the complete verification closure record, and an HMAC-SHA256 provenance signature over the entire package.

The evidence package is not a report. It is a machine-readable, cryptographically signed artifact that proves: the problem was defined before the specification was written, the specification was complete before tests were generated, tests were locked before implementation began, every requirement traces to at least one test, and all eleven verification checks passed. A regulator, an auditor, or a new team member can verify any of these claims from the evidence package alone, without access to the original practitioners.

04

The Governed Lifecycle

From first problem observation to evidenced closure

SDPF defines a complete lifecycle from the first observation of a problem through the final archived evidence package. The lifecycle has two phases and seven pre-deployment stages. SDPF vNext — the runtime architecture described in Section 5 — extends this to ten stages including continuous post-deployment governance.

Stage	What Happens
Phase 0 — Observe, Define, Own	Problem observed. Quantified gap established. Problem statement validated against four tests. Problem owner named. Phase 1 gate opens.
S-1: Draft	Specification authored from validated problem statement. Style selected. All ten required sections written. Every requirement tagged.
S-2: Locked	TVG verified against live environment. No CRITICAL-CRITICAL conflicts. Specification locked. REQ-IDs assigned. Irreversible.
S-3: Tests Generated	Test suite generated from locked specification. Every test traces to a REQ-ID. TEST-IDs assigned.
S-4: Tests Locked	Test suite frozen. Implementation gate open. No implementation may be generated before this stage.
S-5: Code Generated	Implementation generated from locked specification. Every function annotated with its REQ-ID.
S-6: Verified	All eleven structural invariant checks pass. All tests pass. CLOSURE STATUS = COMPLETE.
S-7: Evidenced	Signed evidence package exported and archived. Lifecycle closed.

Every transition in this lifecycle is a hard gate. No out-of-sequence action is permitted. A conforming tool raises a blocking error on any gate violation. The discipline is enforced structurally, not through practitioner memory or organisational culture.

05

SDPF vNext

The runtime architecture for continuous governed execution

SDPF v1.3.1 ends at deployment. The specification is the authoritative contract up to the moment the system is shipped. After that, the contract sits in a folder and the running system is on its own.

For most systems, this is an acceptable operational risk that monitoring and alerting can manage. For regulated systems, it is not. A CQC inspector does not want to know that the system was correct at deployment. They want to know that it has remained correct throughout its operational life. A system whose database schema migrates three weeks after deployment, silently breaching a write-latency invariant, is non-compliant — whether or not anyone knows.

> *The specification does not stop being authoritative after deployment. SDPF vNext makes that authority enforceable.*

What vNext Is

SDPF vNext is an architectural framework — the specification of a governed execution runtime. It is not the runtime itself. The relationship between vNext and the runtime it specifies is the same as the relationship between an SDPF specification and the code generated from it. vNext is the specification. The runtime is what gets built from it.

vNext extends SDPF v1.3.1 in four directions: a formal invariant algebra with composition, inheritance, and conflict semantics; a closed-loop execution runtime that adapts and regenerates on verification failure within defined bounds; a runtime governance model that enforces invariants continuously after deployment; and a six-level verification assurance hierarchy from syntax validity through runtime invariant preservation.

The Runtime Architecture

Component	Function
IR Parser	Ingests a locked SDPF specification and produces the canonical Intermediate Representation — the computational object all downstream operations execute against. The specification ceases to be

	a document and becomes a structured governance contract.
Constraint Engine	Executes every TVG entry automatically against the live environment. Builds a constraint graph from all invariants. Verifies satisfiability — that all invariants can be simultaneously satisfied. Blocks on any failure.
Generation Runtime	Submits the structured IR to the AI generation model. Receives the generated artifact. Passes it directly to the Verification Engine.
Verification Engine	Runs all eleven structural checks and the full test suite. On failure, analyses the failure, refines the relevant invariant, regenerates, and reverifies — automatically, within defined bounds — before escalating.
Runtime Governance Monitor	After deployment, loads the invariant set from the IR and monitors the running system continuously. Detects drift. Attempts automated remediation. Records every event in the provenance chain for the lifetime of the system.

The Verification Assurance Hierarchy

vNext defines six levels of verification assurance. Regulated environments require L5 — runtime invariant preservation — which is only achievable through the Runtime-Observed stage of the lifecycle.

Level	What Is Verified
L0 — Syntax Validity	Specification grammar is valid. Required sections present.
L1 — Structural Completeness	All structural invariants satisfied. Dependency graph complete.
L2 — Behavioral Correctness	All behavioral invariants satisfied. Functional tests pass.
L3 — Security Validation	All security invariants satisfied. Access controls verified.
L4 — Semantic Confidence	Semantic and policy invariants satisfied. Regulatory constraints met.
L5 — Runtime Invariant Preservation	All invariants confirmed to hold under live operational conditions. Requires Runtime-Observed data. Mandatory for regulated deployments.

06

For Regulated Domains

What SDPF provides that no prior approach could deliver

Regulated industries — healthcare, finance, aerospace, government, pharmaceuticals — have been structurally unable to adopt generative AI in production. The reason is not model quality. The reason is audit trail. Regulatory frameworks require organisations to demonstrate that their systems behave as specified, that changes are controlled, and that compliance can be verified at any point by an independent auditor. None of the existing approaches to AI governance produce this.

What Existing Approaches Produce

Approach	What It Produces
Evaluations and benchmarks	A statistical measure of model behavior on a test set. Not traceable to specific requirements. Not signed. Not verifiable against a specific system.
Model cards	A description of model capabilities and limitations. Not specific to a deployment. Not traceable to implementation decisions.
Red-teaming	Evidence that known failure modes were tested. Not evidence that all invariants are satisfied. Not continuous.
Prompt templates	A starting point for instructions. Not verifiable. Not traceable. Not signed. Not continuous.
Monitoring and alerting	Detection of failures after they occur. Not prevention. Not traceable to specification. Not a governance contract.

What SDPF Produces

SDPF produces a complete, signed, tamper-evident chain from business problem through specification through implementation through verification through deployment through runtime operation. Every link in this chain is individually verifiable. The evidence package contains:

The validated problem statement — proving the system was built to solve a defined, measurable problem. The locked specification — proving the contract was complete and verified before any code was generated. The test suite with traceability — proving every requirement was covered by at least one test, and every test traces to a requirement. The implementation hash — proving the code that was verified is the code that was deployed.

The verification closure record — proving all eleven structural invariant checks passed. The provenance signature — proving none of it has been altered.

Under vNext, this chain extends through the operational lifetime. Every drift event, every remediation action, every re-verification cycle is added to the provenance chain with a timestamp and a signature. The evidence package is a living document, not a snapshot. A regulator can inspect it at any point and trace any observed behavior of the running system back to the invariant that governs it.

> *SDPF is the first audit trail of its kind for AI-generated software. It is the artifact regulated industries have been waiting for.*

07

For Standards Organisations

What SDPF provides for adoption as a governance standard

SDPF is designed from the ground up to be adoptable by independent parties without access to the original author. The Language Specification defines every normative rule with sufficient precision for independent implementation, certification, and auditing. The conformance classes define exactly what a conforming specification, tool, evidence package, and process must satisfy.

Four Conformance Levels

Level	What It Certifies
Level 1 — Specification Conformance	The specification satisfies the SDPF grammar. All ten required sections present. TVG completed. All requirements tagged. Traceability matrix non-empty. No CRITICAL-CRITICAL conflicts.
Level 2 — Tool Conformance	The tool enforces Phase 0, enforces the three core principles as blocking gates, enforces stage sequence, assigns REQ-IDs, detects and resolves conflicts, produces conforming evidence packages, and supports all 17 normative styles.
Level 3 — Evidence Conformance	The evidence package contains all required fields, stage = VERIFIED, valid provenance signature, CLOSURE STATUS = COMPLETE, complete traceability, and the validated problem statement.
Level 4 — Process Conformance	The organisation completes Phase 0 before every project, uses a Level 2 tool, produces Level 1 specifications, generates Level 3 evidence for every release, and applies the change protocol to all post-lock modifications.

These conformance levels are independently verifiable. A certification body can evaluate any specification, tool, evidence package, or process against the normative rules in the Language Specification and issue a conformance determination without reference to Hamza Abdullah or any other specific authority. This is the property that makes SDPF suitable for adoption as a standard.

What Makes SDPF a Standard, Not a Methodology

Most governance frameworks for AI development are methodologies: sets of practices that, if followed, tend to produce better outcomes. SDPF is not a methodology. It is a formal

language with a foundational theorem, normative rules, and conformance requirements. The distinction matters for three reasons.

A methodology can be followed well or poorly, and there is no formal basis for distinguishing conforming from non-conforming practice. A standard with formal conformance requirements can be certified, audited, and enforced. A methodology produces better outcomes when its practitioners are skilled. A standard produces verifiable outcomes because its gates are structural, not cultural. A methodology is a professional community's shared practice. A standard is adoptable infrastructure.

SDPF is adoptable infrastructure. The Bounded Stochasticity Theorem is the formal property that makes it defensible. The Language Specification is the normative definition that makes it implementable. The conformance classes are the audit criteria that make it certifiable.

08

What Exists Today and What Comes Next

The current state of SDPF and the development roadmap

What Is Ready Today

SDPF v1.3.1 is a complete, stable specification. It is immediately implementable. A practitioner can adopt Phase 0, the specification language, the TVG, and the evidence standard today, without any additional tooling, and produce governed, evidenced AI-generated software.

Component	Status
Problem identification method — Phase 0	Ready. Implement today.
Specification language and grammar	Ready. Implement today.
TVG and pre-deployment verification	Ready. Implement today.
Evidence standard and provenance	Ready. Implement today.
17 normative styles	Ready. Implement today.
Four conformance levels	Ready. Certify today.
User Manual and Framework Tutorial	Ready. Practitioner guidance complete.
Language Specification v1.3.1 + Addendum A	Stable. All identified gaps closed.

What the vNext Roadmap Builds

SDPF vNext defines the architecture of the governed execution runtime. It is an architectural specification — complete and formally defined — awaiting implementation. The four-phase build roadmap:

Phase	Deliverable
Phase 1 — Formal Semantics	Mathematical foundations: constraint logic formalisation for the invariant algebra, transition semantics for the extended state machine, satisfiability theory and constraint solver integration.
Phase 2 — Execution Infrastructure	The computational substrate: semantic IR parser conforming to the normative schema, constraint graph engine with conflict detection, verification runtime implementing the six-level assurance hierarchy, provenance ledger.
Phase 3 — Autonomous Governance	The runtime governance layer: closed-loop execution runtime with bounded regeneration, runtime invariant monitoring and drift detection, automated remediation, rollback triggering, continuous verification.
Phase 4 — Semantic Intelligence	Semantic correctness extension: symbolic execution engine integration, semantic equivalence verification, proof-carrying generation.

09

A New Science

SDPF as the founding work of a new discipline

SDPF is not a methodology, a framework, or a set of best practices. It is the founding work of a new science: the science of specification for stochastic generators.

This science has a new object of study — specifications whose function is to bound the space of acceptable outputs from a probabilistic process. This object did not exist as a coherent object of study before generative AI made probabilistic production of software possible at scale. Prior disciplines touched it tangentially. None contained it.

Formal methods studied deterministic executors. AI safety studies the models themselves. Software methodology describes how humans organise themselves to produce software. Infrastructure tooling operates on environmental state after the artifact exists. None of these disciplines was built for the object SDPF studies — the relationship between a deterministic specification and a stochastic generator.

What This Science Makes Possible

Capability	Why It Matters
Deterministic governance of stochastic production	For the first time, outputs of AI generation are either conforming or not — and the distinction is verifiable, not statistical.
Auditable AI in regulated contexts	The evidence package is the first audit trail of its kind. It traces every decision from problem through requirement through implementation through verification through runtime.
Specification as the primary unit of engineering labour	When the generator is stochastic, the specification is the deterministic element. The specification becomes the work. Code becomes its mechanical realisation.
A discipline of AI use distinct from AI safety	Safety asks whether a model is safe to deploy. This science asks how a deployed model's output can be bounded to a specification. Both questions must be answered. Only one was named before SDPF.

The science is at its founding. The foundational result has been stated. First-generation apparatus exists. The object is named. What remains is what always remains at the founding of a discipline: adoption, replication, implementation by independent parties, development of a community of practitioners, accumulation of case studies, and refinement of apparatus through contact with diverse real-world problems.

> *The test of a founding work is not whether it is immediately adopted. The test is whether, when adoption happens, the adopted discipline continues to rest on the original foundation.*

Conclusion

The industry has been trying to solve the wrong problem. It has been trying to make AI models more reliable. SDPF solves the right problem: it makes the specifications given to AI models more complete.

A complete specification — one where every fact is verified, every invariant is defined, every requirement is traceable, and every error condition is named — gives an AI model nothing to guess about. The output is not a better guess. It is a mechanical execution of a verified contract. The Bounded Stochasticity Theorem is the formal result that makes this defensible: conformance is a property of invariant satisfaction, not surface-form reproduction, and any output that satisfies all invariants is a conforming realisation of the specification regardless of how it is written.

SDPF provides the language for writing complete specifications, the mechanism for verifying every technical fact before execution, the lifecycle protocol for governing every stage from problem definition through evidenced closure, and — through vNext — the runtime architecture for extending that governance throughout the operational lifetime of the system.

For developers and architects, it is a practitioner discipline available today. For regulated organisations, it is the audit trail their regulatory frameworks require. For standards bodies, it is a formally defined, independently certifiable governance standard for AI-generated software. For researchers and institutions, it is the founding work of a new science.

The framework is complete. The document set is stable. The conformance requirements are defined. Adoption does not require access to the author. It requires reading the specification.

The SDPF Document Set

Document	Contents
SDPF Language Specification v1.3.1	The normative authority. Grammar, styles, Phase 0, TVG, Bounded Stochasticity Theorem, verification model, evidence standard, conformance requirements.
SDPF Language Specification v1.3.1 — Addendum A	Stabilisation patches. TEST-ID format. Normative Annex B with all eleven VCR-INV identifiers. Foreword correction. Updated normative requirement index.

SDPF User Manual and Framework Tutorial	Practitioner guide from zero knowledge to first verified, evidenced run. Phase 0 step by step. Specification authoring. TVG. Generation. Verification. Full project checklist.
SDPF Styles Playbook v2.1	Detailed guidance on all 17 normative styles. Selection matrix. Application guidance.
SDPF Architect Thinking v2.2	How an experienced practitioner thinks, decides, and works at mastery.
SDPF Engineering Intent	Why SDPF makes engineering intent structurally impossible to ignore. REQ-ID chains. The eleven checks.
SDPF Bounded Stochasticity Theorem	The formal result. Why it is relevant, unique, and important. The argument for its necessity.
SDPF — A New Science	SDPF as the founding work of a new scientific discipline.
SDPF Problem Definition	The five failure modes. The specific gap. The consequence of not solving it.
SDPF vNext Comprehensive Reference v2	Full governed execution architecture. Phase 0. Normative IR schema. Closed-loop runtime. Six-level verification hierarchy. Runtime governance. Provenance key management.
SDPF vNext Integrated Case Study	The medication dispensing API governed twice — manually under v1.3.1 and by the vNext runtime — side by side at every stage.

Citation

Abdullah, H. (2026). SDPF: Governed AI Execution — The Specification as the Governing Contract. Software Development Prompting Framework White Paper.

For the normative specification: Abdullah, H. (2026). SDPF Language Specification, Version 1.3.1. Software Development Prompting Framework.

Problem first. Specification second. Facts before execution. Verification always.

Hamza Abdullah · Software Development Prompting Framework · 2026