The Bounded Stochasticity Theorem

A Comprehensive Technical Whitepaper

Hamza Abdullah, SDPF Framework 2026

April 2026

Contents

1 Executive Summary

The Bounded Stochasticity Theorem (BST) constitutes a foundational contribution to the formal specification of systems governed by stochastic executors, particularly those operating under the Software Development Prompting Framework (SDPF). This whitepaper provides a comprehensive analysis of the theorem, its mathematical foundations, its proofs, and its implications for AI governance, software certification, and regulated industries.

The BST resolves a fundamental contradiction that emerged when generative AI systems began displacing deterministic executors in software development workflows. Traditional formal methods, spanning Hoare logic, Dijkstra's predicate transformers, Z notation, VDM, the B-method, and Meyer's design by contract, were architected around a deterministic executor assumption. When the executor became stochastic—a language model that may produce different surface outputs for identical inputs—these methods appeared to become inapplicable. The industry's response was to retreat toward empirical evaluation, benchmark measurement, and statistical acceptance criteria rather than formal specification.

The BST reopens this door by demonstrating that stochasticity in the executor does not preclude deterministic specification, provided that specification constrains structural invariants rather than surface forms. The theorem establishes that for a complete SDPF specification defining structural invariants I, an AI-generated output G satisfies the specification if and only if G satisfies all invariants in I. Furthermore, any stochastic variation in G that satisfies I constitutes a conforming realization of S.

This finding carries profound implications. It provides certification bodies with a formal property against which to certify AI-governed systems. It offers regulated industries—healthcare, finance, aviation, pharmaceuticals, and defense—a principled basis for AI adoption in high-assurance contexts. It gives tool vendors an invariant set against which to build verification tooling. It supplies independent implementers with explicit failure conditions that delineate where the theorem applies and where it does not.

The theorem's scope is deliberately bounded. It does not hold when specifications are incomplete, when the Technical Verification Gate remains unverified, or when the AI model lacks sufficient capability to satisfy the defined invariants. These limitations are features rather than weaknesses: a theorem with precisely defined conditions tells implementers exactly when they may rely upon it and exactly when they cannot.

2 Introduction

2.1 The Problem: Formal Methods Under the Deterministic Executor Assumption

The formal methods tradition represents one of the most significant intellectual achievements in computer science. Beginning with Hoare's foundational work on program verification in 1969, continuing through Dijkstra's predicate transformer semantics, and extending through the development of specification languages such as Z, VDM, the B-method, and Meyer's design by contract, formal methods established a rigorous framework for specifying, verifying, and reasoning about software systems.

The entire tradition rests upon a foundational assumption that was never explicitly articulated because it did not need to be: the executor of a specification is deterministic. A human programmer follows instructions. A compiler translates code faithfully. A verified refinement step transforms a specification into implementation through mechanically sound steps. When the question arose whether an implementation satisfied a specification, the answer was a matter of proof, not probability.

This deterministic executor assumption pervaded every aspect of formal methods thinking. Hoare logic reasons about program states through preconditions and postconditions—mathematical statements about what must hold before and after execution. The executor's determinism guarantees that if the precondition holds and the program executes, the postcondition will hold afterward. Dijkstra's weakest precondition calculus similarly depends on deterministic program behavior. Design by contract, as articulated by Meyer, establishes invariant conditions that a correctly functioning implementation must maintain—conditions that a deterministic executor either satisfies or violates, with no probabilistic intermediate states.

The assumption held so universally that it became invisible. Researchers and practitioners in formal methods did not think of themselves as assuming determinism; they were simply reasoning about computation, which they understood to be deterministic. The assumption was load-bearing but unstated, invisible precisely because it was universal.

2.2 AI as Stochastic Executor: The Breaking of an Unstated Assumption

The introduction of large language models as software development executors broke this assumption in a way that the formal methods community did not immediately recognize. A language model is, by architectural necessity, a stochastic function. Given identical inputs at different times, it may produce outputs that differ in surface form—different variable names, different code structures, different documentation

phrasings—even when following the same underlying specification. This is not a defect; it is a consequence of the model's probabilistic nature, its training on diverse corpora, and its generation mechanism, which samples from probability distributions over token sequences.

The formal methods tradition's first response to this disruption was denial or dismissal. Early adopters of AI-assisted coding often treated the stochasticity as noise to be managed through prompt engineering or output filtering rather than as a fundamental challenge to specification practice. The more sophisticated response was growing acceptance that AI outputs could not be held to deterministic specifications—that specification-based verification was simply inapplicable to AI-generated code.

This response manifested industry-wide in the shift toward empirical evaluation. Rather than specifying what an AI system must produce, practitioners began measuring what it typically produced through benchmark suites, evaluation harnesses, and statistical acceptance criteria. Organizations developed red-teaming protocols to identify failure modes without specifying what correct behavior looked like. The field drifted from specification toward measurement, from proof toward evidence, from determinism toward statistics.

This drift carried significant costs. Empirical evaluation cannot exhaustively verify properties; it can only sample from the space of possible behaviors. Benchmarks measure behavior on previously observed inputs but provide no guarantees about novel inputs. Statistical acceptance criteria accept some probability of failure as normal rather than as a specification deficiency. These tools measure behavior; they do not govern it.

2.3 Industry Response: Benchmarks, Red-Teaming, and the Evaluation Trap

The industry's dominant response to AI's stochasticity took the form of evaluation frameworks rather than specification frameworks. Organizations invested heavily in developing benchmark suites—MMLU, HumanEval, MBPP, and numerous domain-specific alternatives—that measured AI performance on standardized tasks. These benchmarks provided comparative data about model capabilities but did not specify what a correctly functioning system must do.

Red-teaming emerged as a complementary approach: structured attempts to break AI systems by probing their failure modes. Red-teaming identifies vulnerabilities without specifying correct behavior. It tells practitioners what the system does wrong without telling them what it must do right.

The evaluation paradigm carried an implicit message: formal specification of AI systems is impossible or unnecessary. Specification was treated as incompatible with the stochastic nature of generative AI. The assumption underlying this conclusion

was precisely the old formal methods assumption—that specification requires deterministic executors—but now deployed in reverse, as an argument that specification itself was the casualty of AI's arrival rather than merely the executor's identity.

This conclusion was premature. It conflated two distinct questions: whether the executor must be deterministic, and whether the specification must constrain surface outputs. The formal methods tradition made no such conflation; it simply never had occasion to consider stochastic executors because none existed. The tradition's insight—that specifications should constrain invariants rather than instantiations—had always been present but had gone unstated because the distinction between invariants and outputs did not matter when all executors were deterministic.

2.4 The Resolution: Specify Invariants, Not Outputs

The Bounded Stochasticity Theorem articulates what the formal methods tradition's insight always implied but never required stating: specification should constrain what the output must preserve rather than what the output must be. This move from output specification to invariant specification is not new—it is latent in the formal methods tradition's own reasoning about program correctness. Invariant-based reasoning predates AI by decades.

What is new is the recognition that this move resolves the apparent contradiction between deterministic specification and stochastic execution. If a specification defines what properties the output must have—what mappings from inputs to outputs must hold, what error conditions must be handled, what requirements must be satisfied—rather than what the surface form of the output must look like, then a stochastic executor can satisfy the specification by producing any output that satisfies the invariants. Different surface forms that preserve the same invariants are not variations from the specification; they are alternative conforming realizations of it.

The BST provides the formal foundation for this resolution. It proves that for complete SDPF specifications, conformance reduces to invariant satisfaction, and that stochastic variation is bounded by the invariant set. The theorem does not claim the AI is deterministic; it claims the acceptance criterion is. It shifts determinism from the generator to the boundary around the generator.

3 Foundational Definitions

This section establishes the formal mathematical framework within which the Bounded Stochasticity Theorem is stated and proved. All definitions are presented in full mathematical notation following standard conventions in mathematical logic and formal specification.

Definition 1 (Output Space). Let $\mathcal{G}$ be the set of all possible AI-generated outputs. Each element $G \in \mathcal{G}$ represents a complete artifact produced by a generative AI system in response to a specification input.

The output space $\mathcal{G}$ encompasses the full range of artifacts that a generative AI system might produce, including but not limited to source code modules, API specifications, documentation, test suites, configuration files, and deployment scripts. The definition makes no assumptions about the structure or properties of individual outputs beyond their existence as finite artifacts produced by the AI system.

Definition 2 (Specification Space). Let $\mathcal{S}$ be the set of all SDPF specifications. Each element $S \in \mathcal{S}$ is a formally structured document satisfying the grammar and structural requirements defined in the SDPF Language Specification v1.3.1.

A specification $S \in \mathcal{S}$ is characterized by a set of required sections including the style declaration (S-01), style context (S-02), external contract (S-03), input contract (S-04), processing rules (S-05), output guarantees (S-06), exception handling (S-07), technical verification gate (S-08), verification requirements (S-09), and traceability matrix (S-10). Additionally, specifications may include style-specific sections as required by the declared SDPF style.

Definition 3 (Structural Invariant). A structural invariant $i \in I(S)$ is a formal predicate defined by a complete SDPF specification S that maps an output $G \in \mathcal{G}$ to a Boolean value:

$$i : \mathcal{G} \to \{\text{TRUE}, \text{FALSE}\}$$

The set $I(S)$ denotes the complete collection of structural invariants defined by specification S. Each invariant captures a property of the output that must hold across all conforming realizations, regardless of surface form variation. Structural invariants correspond to the semantic properties that the specification requires the implementation to preserve—not the syntactic characteristics that may vary freely.

Definition 4 (Invariant Satisfaction). An output $G \in \mathcal{G}$ satisfies the invariant set $I(S)$ of a specification S, denoted $\text{Sat}(G, I(S))$, if and only if:

$$\text{Sat}(G, I(S)) \equiv \forall i \in I(S) : i(G) = \text{TRUE}$$

This definition establishes that satisfaction requires all invariants in the specification's invariant set to hold for the given output. The conjunction over all invariants is strict: failure of any single invariant constitutes failure to satisfy the invariant set.

Definition 5 (Specification Completeness). A specification $S \in \mathcal{S}$ is complete, denoted $\text{Comp}(S)$, if and only if all of the following conditions hold:

1. **Section Completeness:** All required sections S-01 through S-10 are present and non-empty.
2. **Style Declaration:** A valid SDPF style is declared in section S-01.
3. **Requirement Coverage:** Every requirement in the specification is tagged with a priority level ([CRITICAL], [REQUIRED], or [OPTIONAL]).
4. **TVG Completeness:** All asserted technical facts in the specification have corresponding entries in the Technical Verification Gate (S-08), and all TVG entries have been verified against live output.
5. **Conflict Resolution:** No unresolved conflicts exist between [CRITICAL] requirements.
6. **Invariant Definability:** For every requirement, the specification permits derivation of at least one structural invariant that captures the requirement's semantic content.

$$\begin{aligned}\text{Comp}(S) \equiv\ & \text{SectionComplete}(S) \wedge \text{StyleDeclared}(S) \wedge \\ & \text{RequirementsTagged}(S) \wedge \text{TVGComplete}(S) \wedge \\ & \text{ConflictsResolved}(S) \wedge \text{InvariantsDerivable}(S)\end{aligned}$$

Definition 6 (SDPF Conformance). An output $G \in \mathcal{G}$ conforms to a specification $S \in \mathcal{S}$, denoted $\text{Conforms}(G, S)$, if and only if the specification is complete and the output satisfies all structural invariants:

$$\text{Conforms}(G, S) \equiv \text{Comp}(S) \wedge \text{Sat}(G, I(S))$$

Conformance thus has two necessary and sufficient conditions: the specification must be complete according to Definition 5, and the output must satisfy all structural invariants defined by that complete specification. Neither condition alone suffices; completeness without invariant satisfaction, or invariant satisfaction without a complete

specification, does not constitute conformance.

Definition 7 (Surface Form Equivalence). Two outputs $G_1 \in \mathcal{G}$ and $G_2 \in \mathcal{G}$ are surface-form equivalent, denoted $G_1 \cong G_2$, if and only if they satisfy the same structural invariants:

$$G_1 \cong G_2 \equiv I(G_1) = I(G_2) \text{ where } I(G) = \{i \in I(S) : i(G) = \text{TRUE}\}$$

Surface form equivalence captures the notion that two outputs differ only in their superficial characteristics—variable names, code formatting, documentation style—while preserving the same semantic properties. Under SDPF, surface-form equivalent outputs are treated as equally conforming realizations of the same specification.

Definition 8 (Generative AI as Stochastic Function). Let M be a generative AI model. M is modeled as a stochastic function that maps a specification $S \in \mathcal{S}$ to a probability distribution over outputs $\mathcal{G}$:

$$M(S) \sim P(G \mid S)$$

The probability distribution $P(G \mid S)$ captures the stochasticity inherent in the model's generation process, including sampling from token probability distributions, temperature-based variation, and other sources of nondeterministic output. The model may produce different outputs G on different invocations with the same specification S, each with some probability determined by the model's architecture and parameters.

Definition 9 (Bounded Stochasticity). An output $G \in \mathcal{G}$ produced from specification $S \in \mathcal{S}$ exhibits bounded stochasticity, denoted $\text{Bounded}(G, S)$, if and only if for every output $G' \in \mathcal{G}$ with positive probability under $P(G' \mid S)$, G' is conforming if and only if it satisfies the invariants:

$$\text{Bounded}(G, S) \equiv \forall G' \in \mathcal{G} : P(G' \mid S) > 0 \Rightarrow (\text{Conforms}(G', S) \leftrightarrow \text{Sat}(G', I(S)))$$

Bounded stochasticity establishes that the space of possible outputs from the stochastic generator is partitioned by the invariant set: outputs either satisfy all invariants (and are conforming) or fail at least one invariant (and are non-conforming). There is no third category of outputs that are structurally conforming despite failing invariant checks, nor outputs that satisfy invariants without being conforming. The invariant set forms a complete boundary around acceptable stochastic variation.

4 The Bounded Stochasticity Theorem: Formal Statement

With the foundational definitions established, we now state the Bounded Stochasticity Theorem in its complete mathematical form.

Bounded Stochasticity Theorem

Let $S \in \mathcal{S}$ be a complete SDPF specification defining a set of structural invariants $I(S)$. Let $G \in \mathcal{G}$ be a generative AI output produced from $M(S)$.
Then the following statements hold:
Statement 1 (Conformance Characterization):

$$\text{Conforms}(G, S) \iff \text{Sat}(G, I(S))$$

That is, G conforms to S if and only if G satisfies all invariants in $I(S)$.
Statement 2 (Bounded Variation):

$$\forall G' \in \mathcal{G} : P(G' \mid S) > 0 \Rightarrow \text{Bounded}(G', S)$$

That is, every output with positive probability under the generative model exhibits bounded stochasticity with respect to S.
Corollary (Specification Independence): Given a complete specification S, the truth of $\text{Sat}(G, I(S))$ is independent of the particular surface form of G. Two surface-form equivalent outputs satisfy the same invariant set.
Corollary (Verification Sufficiency): For complete specifications, verification of invariant satisfaction is sufficient for establishing conformance. No additional checking of surface form characteristics is required.

5 Complete Mathematical Proof

This section presents a rigorous proof of the Bounded Stochasticity Theorem using the MSPV (Model-Solve-Prove-Verify) framework. The proof is presented in four phases, following standard mathematical practice for theorems of this complexity.

5.1 Phase 1: Model

The Model phase establishes the formal entities and their relationships within a precise mathematical framework.

5.1.1 Entity Definitions

We define the following entities:

- **Output Space** $\mathcal{G}$: The set of all finite artifacts producible by a generative AI system, as given in Definition 1.
- **Specification Space** $\mathcal{S}$: The set of all SDPF-conforming specifications, as given in Definition 2.
- **Invariant Set** $I(S)$: For each $S \in \mathcal{S}$, the set of structural invariants derivable from S, as given in Definition 3.
- **Stochastic Model** M: A generative AI system modeled as a stochastic function $M(S) \sim P(G \mid S)$, as given in Definition 8.

5.1.2 Key Properties

From the entity definitions, we establish the following properties:

Property M1 (Finite Invariant Sets): For any complete specification $S \in \mathcal{S}$, the invariant set $I(S)$ is finite. This follows from the finite nature of SDPF specifications and the requirement that each requirement be tagged and resolvable.

Property M2 (Invariant Determinism): Each invariant $i \in I(S)$ is a deterministic predicate: for any output $G \in \mathcal{G}$, $i(G)$ evaluates to exactly one of TRUE or FALSE. There is no probabilistic evaluation of invariants.

Property M3 (Conformance Decomposition): By Definition 6, $\text{Conforms}(G, S) \equiv \text{Comp}(S) \wedge \text{Sat}(G, I(S))$. This decomposition is foundational to the proof strategy.

5.1.3 Formalization of Stochastic Variation

The stochastic nature of the generative model M implies that for a fixed specification S, the model may produce any output $G' \in \mathcal{G}$ with probability $P(G' \mid S) \geq 0$. The

probabilities sum to 1 over the support of the distribution:

$$\sum_{G' \in \mathcal{G}} P(G' \mid S) = 1$$

We make no assumptions about the specific probability distribution, only that it is well-defined and that each invocation of $M(S)$ produces an output from $\mathcal{G}$ according to this distribution.

5.2 Phase 2: Solve

The Solve phase identifies the necessary conditions for the theorem to hold and derives the constraints under which the proof must operate.

5.2.1 Necessary Conditions

For Statement 1 ($\text{Conforms}(G, S) \iff \text{Sat}(G, I(S))$) to hold, we require:

Condition S1 (Completeness Precondition): $\text{Comp}(S)$ must be TRUE. If the specification is incomplete, the decomposition in Property M3 does not apply, and conformance cannot be characterized solely by invariant satisfaction. The theorem requires complete specifications; incomplete specifications fall outside its scope.

Condition S2 (Invariant Definability): Every element of $I(S)$ must be a well-defined predicate on $\mathcal{G}$. If invariants are undefined or ambiguous, invariant satisfaction cannot be meaningfully evaluated.

Condition S3 (Model Capability): The generative model M must be capable of producing outputs that satisfy $I(S)$. If the model's capacity is insufficient to satisfy the invariants, $\text{Bounded}(G, S)$ cannot hold for all G' with positive probability.

5.2.2 Derivation of the Forward Direction Constraint

Assume $\text{Conforms}(G, S)$. By Property M3:

$$\text{Conforms}(G, S) \equiv \text{Comp}(S) \wedge \text{Sat}(G, I(S))$$

Since $\text{Conforms}(G, S)$ is TRUE, both conjuncts must be TRUE. Therefore:

$$\text{Comp}(S) = \text{TRUE} \wedge \text{Sat}(G, I(S)) = \text{TRUE}$$

This establishes that conformance implies invariant satisfaction under the completeness precondition.

5.2.3 Derivation of the Reverse Direction Constraint

Assume $\text{Sat}(G, I(S)) = \text{TRUE}$. To conclude $\text{Conforms}(G, S) = \text{TRUE}$, we require:

$$\text{Comp}(S) \wedge \text{Sat}(G, I(S)) = \text{TRUE}$$

Given $\text{Sat}(G, I(S)) = \text{TRUE}$, we need only establish $\text{Comp}(S) = \text{TRUE}$. This requires that the specification has been verified as complete through the Technical Verification Gate and all other completeness conditions.

5.3 Phase 3: Prove

The Prove phase establishes the bidirectional logical equivalence and the bounded stochasticity claim through rigorous mathematical argument.

5.3.1 Proof of Statement 1: Conformance Characterization

Forward Direction ($\Rightarrow$):

Assume $\text{Conforms}(G, S)$. By Definition 6:

$$\text{Conforms}(G, S) \equiv \text{Comp}(S) \wedge \text{Sat}(G, I(S))$$

Therefore:

$$\text{Comp}(S) \wedge \text{Sat}(G, I(S)) = \text{TRUE}$$

From this conjunction being TRUE, we conclude:

$$\text{Sat}(G, I(S)) = \text{TRUE}$$

Reverse Direction ($\Leftarrow$):

Assume $\text{Sat}(G, I(S)) = \text{TRUE}$. Additionally, assume $\text{Comp}(S) = \text{TRUE}$ (scope condition from Phase 2).

Then:

$$\text{Comp}(S) \wedge \text{Sat}(G, I(S)) = \text{TRUE}$$

By Definition 6:

$$\text{Conforms}(G, S) = \text{TRUE}$$

Conclusion of Statement 1:

$$\text{Conforms}(G, S) \iff \text{Sat}(G, I(S)) \qquad \blacksquare$$

5.3.2 Proof of Statement 2: Bounded Variation

We must show that for any $G' \in \mathcal{G}$ with $P(G' \mid S) > 0$, Bounded(G', S) holds.

By Definition 9, Bounded(G', S) requires:

$$\forall G'' \in \mathcal{G} : P(G'' \mid S) > 0 \Rightarrow (\text{Conforms}(G'', S) \leftrightarrow \text{Sat}(G'', I(S)))$$

Let G'' be any output with $P(G'' \mid S) > 0$. We must show:

$$\text{Conforms}(G'', S) \leftrightarrow \text{Sat}(G'', I(S))$$

By Definition 6:

$$\text{Conforms}(G'', S) \equiv \text{Comp}(S) \wedge \text{Sat}(G'', I(S))$$

Since Comp$(S) = \text{TRUE}$ (assumption of the theorem), we have:

$$\text{Conforms}(G'', S) \equiv \text{Sat}(G'', I(S))$$

This establishes the bidirectional equivalence for any G'' with $P(G'' \mid S) > 0$. Since G'' was arbitrary, the equivalence holds for all such outputs.

Conclusion of Statement 2:

$$\forall G' \in \mathcal{G} : P(G' \mid S) > 0 \Rightarrow \text{Bounded}(G', S) \qquad \blacksquare$$

5.3.3 Corollary Proofs

Corollary 1 (Specification Independence):

Given $G_1 \cong G_2$ (surface-form equivalence), by Definition 7, $I(G_1) = I(G_2)$. Therefore, Sat$(G_1, I(S)) = \text{Sat}(G_2, I(S))$. The truth of invariant satisfaction—and hence conformance—is independent of surface form. ■

Corollary 2 (Verification Sufficiency):

For complete specifications, verifying Sat$(G, I(S))$ is sufficient to establish Conforms(G, S) by Statement 1. Surface form checking is neither necessary nor informative for conformance determination. ■

5.4 Phase 4: Verify

The Verify phase confirms that the proof correctly establishes the theorem and addresses potential objections or edge cases.

5.4.1 Scope Verification

The proof holds under the explicit scope conditions:

1. **Specification Completeness:** The theorem requires $\text{Comp}(S) = \text{TRUE}$. This is not an artifact of the proof strategy but a fundamental scope condition. The theorem does not apply to incomplete specifications.
2. **TVG Verification:** The Technical Verification Gate entries must be verified against live output. Unverified technical facts introduce factual errors that invalidate the completeness precondition.
3. **Model Capability:** The proof assumes the model can produce outputs satisfying $I(S)$. If the model's capability is insufficient, the space of achievable outputs may not intersect with the conforming region.

5.4.2 Edge Case Analysis

Case 1: Empty Invariant Set $I(S) = \emptyset$:

If $I(S) = \emptyset$, then $\text{Sat}(G, I(S)) \equiv \forall i \in \emptyset : i(G) = \text{TRUE}$ is vacuously TRUE for all $G \in \mathcal{G}$. Under this condition, Statement 1 reduces to $\text{Conforms}(G, S) \iff \text{TRUE}$, which implies $\text{Conforms}(G, S) = \text{Comp}(S)$. This is consistent: a specification with no invariants has conformance determined entirely by completeness, and any output satisfies the vacuous invariant condition.

Case 2: $P(G' \mid S) = 0$ for some G':

The bounded stochasticity statement only quantifies over G' with $P(G' \mid S) > 0$. Outputs with zero probability do not enter the bounded region because they are never produced. This is intentional: the theorem bounds the space of actually-possible outputs, not the space of conceivable outputs.

5.4.3 Proof Completeness

The proof is complete and self-contained. All steps follow from the definitions and established properties. No auxiliary assumptions beyond the stated scope conditions are required. The MSPV framework has been applied consistently throughout.

6 Scope Conditions and Failure Modes

A theorem with precisely defined scope conditions tells practitioners exactly when they may rely upon it and exactly when they cannot. This section articulates the conditions under which the Bounded Stochasticity Theorem does not hold, providing implementers with clear guidance about the theorem's applicability.

6.1 Condition 1: Specification Incompleteness

Failure Mode: $\neg\text{Comp}(S)$

When the specification fails to satisfy one or more completeness conditions, the theorem's proof does not apply. The decomposition $\text{Conforms}(G, S) \equiv \text{Comp}(S) \wedge \text{Sat}(G, I(S))$ requires $\text{Comp}(S) = \text{TRUE}$; without this, conformance cannot be characterized solely by invariant satisfaction.

Counterexample Construction:

Consider an SDPF specification missing the Exception Handling section (S-07). The invariant set $I(S)$ will not include invariants governing error condition handling. An output G that fails to handle critical error conditions may still satisfy the partial invariant set, yet clearly does not conform to the specification's intent. Formally: $\text{Sat}(G, I(S)) = \text{TRUE}$ but $\text{Conforms}(G, S) = \text{FALSE}$.

Implication: Practitioners must verify specification completeness before relying on the theorem. The Technical Verification Gate and the ten required sections checklist provide the verification mechanism.

6.2 Condition 2: Technical Verification Gate Not Passed

Failure Mode: $\neg\text{TVG\_Passed}(S)$

The Technical Verification Gate verifies that every asserted technical fact in the specification is true in the target environment. If any TVG entry fails verification, the specification contains technical facts that are false. The invariant set derived from a specification with false technical facts may not accurately capture the intended system behavior.

Example:

A specification asserts a Python version of 3.11.15, but the target environment runs 3.12.1. The TVG catches this discrepancy before implementation begins. Without TVG passage, the AI receives a specification containing a false technical fact and produces an implementation that fails in the target environment—yet may satisfy the partial invariant set derived from the incorrect specification.

Implication: TVG passage is a necessary precondition for the theorem's applicability.

The SDPF framework enforces this as a blocking gate: no implementation begins until all TVG entries pass.

6.3 Condition 3: Model Lacks Sufficient Capability

Failure Mode: $\neg\text{Capable}(M, I(S))$

If the generative model M lacks sufficient capability to satisfy the invariants $I(S)$, then there may be no conforming outputs in the support of $P(G \mid S)$. The theorem's bounded stochasticity claim—that all outputs with positive probability are bounded by invariants—may still hold vacuously, but the theorem provides no guidance about achieving conformance when the model cannot produce it.

Example:

A specification defines invariants requiring formal correctness proofs for all algorithmic implementations. A model without theorem-proving capability cannot satisfy these invariants. The space of $P(G \mid S)$ may be entirely non-conforming, and no amount of specification refinement will produce conforming outputs.

Implication: Model selection must be informed by the invariants required by the specification. The theorem does not guarantee that conforming outputs exist; it only characterizes what conformance means when they do.

6.4 Condition 4: Invariant Extraction Ambiguity

Failure Mode: $\neg\text{UniquelyDerivable}(I(S))$

The theorem assumes that the invariant set $I(S)$ is well-defined and that its derivation from the specification S is unambiguous. If natural language requirements admit multiple interpretations, the derived invariant set may not capture the practitioner's intent.

Mitigation:

SDPF's requirement syntax rules (REQ-SYNTAX-1 through REQ-SYNTAX-6) constrain requirement text to reduce ambiguity. Explicit requirement tagging, the conflict resolution protocol, and the style-specific semantic rules further constrain interpretation. However, some ambiguity may persist in complex specifications.

Implication: Practitioners must exercise care in requirement specification to ensure derived invariants accurately capture intended behavior. Ambiguous requirements produce ambiguous invariants, and the theorem applies to the derived invariants, not the intended behavior.

7 Verification Framework: The Eleven Structural Invariant Checks

The SDPF verification framework operationalizes the Bounded Stochasticity Theorem through eleven structural invariant checks that test whether required structural properties hold at each stage of the SDPF lifecycle. These checks verify the process properties that establish the theorem's preconditions, not the surface characteristics of the output.

7.1 The Eleven Checks

Check	**Structural Invariant Tested**	**Formal Property Verified**
spec_exists	Intent was formally expressed from a validated problem statement before execution began.	Phase 0 completion, problem statement validation
style_defined	Intent was classified into a known domain before execution.	Valid style ID from the 17 normative styles or conforming custom style
tests_exist	Intent produced tests—specification was not bypassed.	TESTS_GENERATED stage reached
tests_locked	Tests were frozen before implementation—intent governed the test suite.	TESTS_LOCKED stage reached before CODE_GENERATED
implementation_exists	Implementation was produced after intent was locked.	CODE_GENERATED stage reached
traceability_complete	Every requirement has at least one test—no intent line was skipped.	$\forall R \in \text{Reqs}(S) : \exists T \in \text{Tests}(S) : \text{Traces}(R, T)$
style_constraints_present	The style's structural marker appears in the specification—style was applied.	Style-specific sections S-11, S-12, S-13 where required

Check	Structural Invariant Tested	Formal Property Verified
style_N_engine_verified	Style-specific engine tags appear in tests and code—style governed the artifacts.	Artifacts contain style-specific verification markers
policy_engine_passed	Specification content satisfies semantic rules for the declared style.	Policy evaluation returns PASS
ci_gate_ready	A CI workflow was generated—invariants can be enforced in automated pipelines.	CI configuration artifact present
provenance_signing_ready	A signing key is configured—evidence can be tamper-proofed.	Signing key available for HMAC-SHA256

7.2 Verification Norms

VER-1: All eleven checks shall be evaluated on every verification gate execution. No check may be skipped.

VER-2: A run shall not advance to VERIFIED unless all eleven checks return pass.

VER-3: The result of every check shall be recorded in the Verification Closure Record.

VER-4: A check result shall be boolean. Partial pass is not valid.

VER-5: A failed gate execution leaves the stage at CODE_GENERATED.

VER-6: The verification execution timestamp shall be recorded in the Verification Closure Record.

7.3 Relationship to the Theorem

The eleven structural invariant checks verify the preconditions for the Bounded Stochasticity Theorem rather than the invariant satisfaction itself. Specifically:

- **spec_exists** and **style_defined** verify that a complete specification exists.
- **tests_exist** and **tests_locked** verify that the specification-test-implementation ordering was followed.
- **implementation_exists** verifies that an implementation was produced.
- **traceability_complete** verifies that requirements coverage is complete.

- **style_constraints_present** and **style_N_engine_verified** verify that the specification's declared style was properly applied.
- **policy_engine_passed** verifies that the specification content satisfies the style's semantic rules.
- **ci_gate_ready** and **provenance_signing_ready** verify that the evidence infrastructure is in place.

These checks collectively establish $\text{Comp}(S)$, enabling the application of Statement 1: $\text{Conforms}(G, S) \iff \text{Sat}(G, I(S))$.

8 Five-Gate Verification Hierarchy

The five-gate verification hierarchy provides a layered verification structure that applies the Bounded Stochasticity Theorem's principles at multiple levels of system integration.

8.1 Gate Definitions

Gate	Name	What It Checks	Blocks
G1	Technical Verification Gate	All TVG entries verified against live output. No unverified technical facts in specification.	Build start
G2	Unit Verification	Each component satisfies its individual specification requirement.	Integration
G3	Integration Verification	All components work together as specified at every interface boundary.	System test
G4	Clean-Machine Verification	All outputs execute correctly in an environment with no development dependencies.	Release
G5	Closure Record	All verification results recorded and signed. CLOSURE STATUS = COMPLETE.	Shipment

8.2 Gate Progression

The gates must be passed in sequence. Each gate is a prerequisite for the subsequent gate:

- **G1** must pass before any build begins.
- **G2** must pass before components are integrated.
- **G3** must pass before system-level testing.
- **G4** must pass before release.
- **G5** must pass before shipment.

This sequencing ensures that the theorem's preconditions are established before verification results are trusted, and that verification occurs at appropriate granularity as the system is assembled.

8.3 G1: Technical Verification Gate Details

The Technical Verification Gate (G1) deserves special attention as the first gate and the mechanism by which the theorem's scope conditions are partially verified.

TVG Structure:

Each TVG entry specifies:

- **Tool/Asset:** The name of the tool, library, API, or configuration element.
- **Asserted Value:** The value stated in the specification.
- **Verification Command:** An executable command that can be run in the target environment.
- **Pass Condition:** The expected output or pattern that constitutes verification.
- **HALT Rule:** The action to take if verification fails.

TVG Normative Rules:

- **TVG-1:** Every tool version, library version, API endpoint, flag, path, and environment variable asserted in the specification shall have a TVG entry.
- **TVG-2:** Each TVG entry shall contain the asset name, asserted value, verification command, pass condition, and HALT rule.
- **TVG-3:** If any TVG check fails, the specification shall be updated before implementation proceeds. Work shall not continue around a TVG failure.
- **TVG-4:** A conforming tool shall surface TVG failures as blocking errors.
- **TVG-5:** A specification shall not be locked until all TVG entries have been verified against live output and have passed.

9 Formal Verification Protocol

This section presents the verification algorithm in pseudocode, providing a precise specification for tool implementers.

9.1 Verification Algorithm

```
function Verify_Bounded_Stochasticity(S, G, M):
    # PHASE 1: Check Completeness Precondition
    if not Comp(S):
        return CONFORMANCE_UNKNOWN("Specification incomplete")

    # PHASE 2: Verify Technical Facts
    if not TVG_Passed(S):
        return CONFORMANCE_UNKNOWN("TVG not passed")

    # PHASE 3: Compute Invariant Set
    I = DeriveInvariants(S)

    # PHASE 4: Verify Model Capability
    if not Capable(M, I):
        return CONFORMANCE_UNKNOWN("Model cannot satisfy invariants")

    # PHASE 5: Check Invariant Satisfaction
    all_satisfied = TRUE
    for each i in I:
        if not i(G):
            all_satisfied = FALSE
            failed_invariant = i
            break

    # PHASE 6: Apply Theorem (Statement 1)
    if all_satisfied:
        conformance = TRUE
        basis = "Sat(G, I(S)) = TRUE; Comp(S) = TRUE"
    else:
        conformance = FALSE
        basis = f"Sat(G, I(S)) = FALSE; Failed: {failed_invariant}"

    # PHASE 7: Verify Boundedness (Statement 2)
    # This is verified by checking that TVG and completeness conditions
```

```
    # hold, which ensures the support of P(G|S) is bounded
    boundedness_verified = Comp(S) and TVG_Passed(S)

    # PHASE 8: Generate Verification Closure Record
    VCR = VerificationClosureRecord(
        specification = S,
        output = G,
        conformance = conformance,
        sat_result = all_satisfied,
        failed_invariant = failed_invariant if not all_satisfied else None,
        bounded = boundedness_verified,
        timestamp = current_timestamp_utc()
    )

    return VCR
```

9.2 Subroutine Definitions

```
function Comp(S):
    # Check all completeness conditions
    return (
        SectionComplete(S, [S-01, S-02, ..., S-10]) and
        StyleDeclared(S) and
        RequirementsTagged(S) and
        TVGComplete(S) and
        ConflictsResolved(S) and
        InvariantsDerivable(S)
    )

function TVG_Passed(S):
    for each entry in S.TVG:
        result = Execute(entry.verification_command)
        if not Match(result, entry.pass_condition):
            return FALSE
    return TRUE

function DeriveInvariants(S):
    I = empty set
    for each requirement in S.requirements:
        invariant = DeriveStructuralInvariant(requirement)
        I.add(invariant)
```

```
    return I

function Capable(M, I):
    # In practice, this is established by successful verification
    # on test outputs, not by formal characterization
    # The theorem assumes this condition for meaningful application
    return TRUE
```

9.3 Verification Closure Record Schema

The Verification Closure Record (VCR) captures the complete verification state:

- **specification:** Version identifier of the verified specification.
- **verification_timestamp:** ISO 8601 UTC timestamp of verification execution.
- **verifying_entity:** Identity of the entity performing verification.
- **check_results:** Array of eleven boolean results, one per structural invariant check.
- **conformance:** Boolean indicating overall conformance determination.
- **sat_result:** Boolean indicating invariant satisfaction.
- **failed_invariant:** Identifier of first failed invariant, if any.
- **closure_status:** One of COMPLETE, INCOMPLETE, or BLOCKED.
- **unresolved_items:** List of unresolved items if status is not COMPLETE.

10 Historical Context and Related Work

The Bounded Stochasticity Theorem does not emerge from a vacuum. It sits within a rich tradition of formal methods research while addressing a novel context that the tradition never contemplated.

10.1 The Formal Methods Tradition

Formal methods in software engineering trace their lineage to Hoare's 1969 paper "An Axiomatic Basis for Computer Programming," which introduced the Hoare triple $\{P\}\ C\ \{Q\}$ for specifying program behavior. A Hoare triple states that if precondition P holds before execution of command C, then postcondition Q holds after execution. The elegance of this notation is that it specifies what must be preserved across execution rather than how execution proceeds.

Dijkstra's 1975 "Guarded Commands, Non-determinacy and Formal Derivation of Programs" extended this thinking through the weakest precondition calculus. Dijkstra showed how to derive programs from specifications by computing the weakest precondition that guarantees a desired postcondition. The executor was assumed deterministic and sequential; nondeterminacy appeared only in specification, not execution.

The Z notation (IBM, 1979), VDM (IBM and Brailsford, 1979), and the B-method (Abrial, 1994) provided comprehensive specification languages supporting the formal development of software systems. Each assumed a deterministic refinement step from specification to implementation.

Meyer's design by contract (1986) introduced contract semantics into object-oriented programming through preconditions, postconditions, and invariants enforced at runtime. While Meyer acknowledged that contracts could be dynamic, the practical application assumed deterministic execution within defined contracts.

None of these traditions contemplated stochastic executors. The assumption was universal and unstated, not because the researchers lacked imagination, but because no stochastic executors existed in the contexts they studied.

10.2 Why the Theorem Was Not Needed Before

The formal methods tradition had the insight underlying the BST—specify invariants, not outputs—without requiring its formalization as a theorem. When every executor is deterministic, the distinction between specifying outputs and specifying invariants is merely pedagogical. A deterministic executor that fails to produce the specified output either violates the specification or produces an alternative that satisfies the same invariants. The distinction doesn't matter in practice because all executors are held to the same standard.

The theorem became necessary only when executors became stochastic. Now the distinction is consequential: an output that satisfies invariants but differs from an expected surface form is a conforming realization rather than a failure. The theorem provides the formal vocabulary for making this distinction precise.

10.3 Comparison to Codd's Theorems

Codd's 1970 paper "A Relational Model of Data for Large Shared Data Banks" introduced twelve rules for relational database systems, of which Rule 3 (Systematic Treatment of Null Values) and Rule 10 (Integrity Independence) are particularly notable. More importantly, Codd's work established the mathematical foundation—relational algebra and set theory—showing that the relational model's query language could be characterized by formal properties that guaranteed correct behavior.

Like Codd's theorems, the BST provides a formal foundation that justifies an architectural choice. Codd showed that relational databases could be characterized by mathematical properties independent of implementation. The BST shows that stochastic AI outputs can be characterized by invariant satisfaction independent of surface form.

The analogy extends to practical consequences. Before Codd, database systems were navigational and application-dependent. After Codd, the relational model provided a standard that enabled portability, interoperability, and independent verification. The BST provides similar potential for AI-governed systems: a standard that enables certification, regulation, and independent verification.

10.4 Comparison to the End-to-End Argument

Saltzer, Reed, and Clark's 1984 end-to-end argument articulated a fundamental principle of network architecture: reliability and correctness properties must be implemented at the endpoints of a communication system rather than in the network itself. The argument is not a theorem in the strict mathematical sense, but it functions as one—a formal principle that justifies an architectural choice.

The end-to-end argument resolved debates about where functionality should reside in network systems. It provided a principled basis for preferring endpoint implementation over intermediate implementation, even when intermediate implementation seemed more efficient.

The BST plays an analogous role for AI-governed systems. It provides a principled basis for preferring invariant specification over output specification, resolving the debate about whether AI systems can satisfy formal specifications. It shifts determinism from the executor to the boundary around the executor, just as the end-to-end argument shifts reliability from the network to the endpoints.

10.5 Constrained Generation Tools

A separate line of work has pursued the underlying intuition operationally without formalizing it. JSON Schema validators apply structural constraints to model outputs. Grammar-constrained decoders restrict token sampling to syntactically valid continuations. Guidance libraries (Microsoft) and LMQL provide token-level control over generation.

These tools operationalize the invariant intuition at the syntactic level. They enforce structural constraints on output form—valid JSON, correct syntax, matching schemas—without formalizing the move. They ship code; they do not state theorems.

The contribution of the BST is not the operational insight but its formal articulation. By stating the theorem explicitly, SDPF provides a foundation that constrained generation tools lack: a precisely scoped claim about when invariant specification governs stochastic output.

11 Implications and Applications

The Bounded Stochasticity Theorem carries significant implications across multiple domains of AI development, governance, and deployment.

11.1 Implications for AI Safety

The theorem provides AI safety practitioners with a principled specification framework where none previously existed. Traditional AI safety approaches focus on the model itself—alignment, interpretability, mechanistic understanding. These approaches treat the model as the primary locus of safety concern.

The BST redirects attention from model internals to specification completeness. The safety-critical question becomes whether the specification completely captures the required invariants, not whether the model can be aligned to produce specific outputs. This shift enables:

- **Complete Coverage Analysis:** Given a complete specification, practitioners can verify whether the invariant set covers all safety-critical properties. Incompleteness is detectable and correctable before deployment.
- **Bounded Failure Modes:** When the theorem holds, failure modes are bounded by invariant violations. An output that satisfies all invariants is safe by construction, regardless of its surface form.
- **Reproducible Conformance:** Conformance is a binary property, not a statistical one. Either the output satisfies all invariants or it does not. This determinism enables rigorous safety arguments.

11.2 Implications for AI Governance

Governance frameworks require formal properties to certify against. Empirical evaluation provides evidence; formal properties provide guarantees. The BST provides the formal property—invariant satisfaction under complete specification—that governance frameworks require.

Regulatory bodies can cite the theorem as the basis for audit requirements. Rather than requiring specific implementation approaches or model selections, regulators can require invariant-based specification, complete TVG verification, and conformance verification through the eleven structural invariant checks. This approach is technology-neutral and outcome-focused.

Specific governance applications include:

- **Healthcare:** FDA regulatory submissions can reference invariant-based specification as the standard for AI/ML-enabled devices. Completeness and conformance replace statistical validation.

- **Finance:** OCC, SEC, and international banking regulators can require SDPF conformance for AI systems approving loans, making trading decisions, or flagging suspicious transactions.
- **Aviation:** FAA and EASA certification can incorporate invariant-based specification for flight management systems, air traffic control, and maintenance prediction.
- **Defense:** DoD acquisition standards can require SDPF conformance for autonomous systems, weapons targeting, and intelligence analysis.

11.3 Implications for Regulated Industries

Regulated industries face a common challenge: how to adopt AI systems that satisfy regulatory requirements for predictability, auditability, and safety. The BST provides the theoretical foundation for AI adoption by establishing that:

1. **Complete specifications can govern stochastic outputs.** The historical objection—that AI cannot satisfy deterministic specifications because it is stochastic—is resolved by the theorem's equivalence.
2. **Conformance is verifiable.** The eleven structural invariant checks provide a concrete procedure for verifying conformance, replacing statistical sampling with deterministic checking.
3. **Evidence is producible.** The evidence package schema provides a standardized format for documenting conformance, enabling regulatory review and independent audit.

11.4 Implications for Certification Bodies

Certification bodies—ISO, IEC, IEEE, NIST, and national standards organizations—require formal standards to certify against. The BST enables certification bodies to:

- **Define conformance levels** based on the theorem's conditions: specification completeness, TVG passage, invariant satisfaction, and bounded stochasticity verification.
- **Accredit testing laboratories** to verify conformance using the eleven structural invariant checks and the verification algorithm.
- **Issue certificates** attesting to conformance, enabling market participants to demonstrate regulatory compliance.

11.5 Implications for Tool Vendors

Tool vendors building AI-powered development environments can leverage the BST to provide formal guarantees about their outputs. By implementing:

- The SDPF lifecycle protocol with stage gates,

- The Technical Verification Gate with live verification,
- The eleven structural invariant checks,
- The evidence package schema with provenance signing,

tool vendors can offer products that produce verifiably conforming outputs. Customers receive not merely useful tools but formally governed systems whose outputs satisfy invariant-based specifications.

12 Limitations and Future Work

The Bounded Stochasticity Theorem represents progress toward formal governance of AI systems, but it has limitations that scope its applicability and identify directions for future research.

12.1 Specification Completeness Is Not Automatically Verified

The theorem requires $\text{Comp}(S) = \text{TRUE}$, but the SDPF framework relies on human review and tool enforcement to establish completeness. The eleven structural invariant checks verify that the required sections exist and that processes were followed, but they do not automatically verify that the specification content is semantically complete.

Future Work: Research into automated completeness verification could reduce reliance on human judgment. Techniques from requirements engineering—goal modeling, problem frames, completeness checking—could be adapted to verify specification completeness algorithmically.

12.2 Model Capability Is Not Formally Characterized

The theorem assumes $\text{Capable}(M, I(S))$ but does not provide a method for formally characterizing model capability. In practice, capability is established through empirical testing rather than formal analysis. The theorem applies when the model can satisfy invariants; it does not tell practitioners how to determine whether a given model can do so.

Future Work: Formal characterization of model capability relative to invariant classes could enable principled model selection. If invariants can be classified by difficulty or type, empirical or analytical methods could map model capabilities to invariant classes.

12.3 Invariant Extraction from Natural Language

The SDPF requirement syntax constrains natural language to reduce ambiguity, but significant interpretation remains. Different practitioners may derive different invariants from the same requirement text. The theorem applies to the derived invariant set, not to the practitioner's intent.

Future Work: Research into invariant extraction algorithms—formal methods for deriving formal invariants from informal specifications—could reduce interpretation variance. Controlled natural language subsets, semantics-based extraction, and AI-assisted invariant derivation are promising directions.

12.4 Scaling to Complex Systems

The theorem has been demonstrated on individual specification units, but complex systems comprise multiple specifications, multiple models, and emergent properties. Whether the theorem scales to system-level properties—performance, scalability, composability—remains an open question.

Future Work: Extension of the theorem to composition of specifications, interface contracts between specifications, and system-level invariants would broaden its applicability to complex AI systems.

13 Conclusion

13.1 Summary of Contributions

This whitepaper has presented a comprehensive analysis of the Bounded Stochasticity Theorem, establishing its mathematical foundations, presenting its complete proof using the MSPV framework, articulating its scope conditions and failure modes, and exploring its implications for AI governance, safety, and regulated industries.

The BST resolves a fundamental apparent contradiction in formal specification: the methodology is deterministic, but the executor is stochastic. The theorem demonstrates that this contradiction was never real—it arose from specifying the wrong thing. Specify what the output must preserve, not what the output must be, and the stochasticity of the generator becomes acceptable variation within bounded invariants.

13.2 The Theorem's Significance

A framework without a theorem is a methodology. A framework with a theorem is an infrastructure. The formal methods tradition established this principle through Hoare logic, Dijkstra's weakest preconditions, and design by contract. The BST extends the principle to stochastic executors.

The relational database field had Codd's theorems. Network architecture had the end-to-end argument. Formal methods had Hoare logic. AI-governed software development now has the Bounded Stochasticity Theorem. This is not a coincidence of framing—it is a structural parallel that identifies the formal property necessary for a technology to become adoptable infrastructure.

13.3 The Theorem's Significance Beyond SDPF

While the theorem is stated within the SDPF framework, its insight is framework-independent. Any specification methodology that defines structural invariants over AI-generated outputs can apply the theorem's equivalence. The formal move—specifying invariants rather than outputs—is applicable wherever stochastic executors must satisfy deterministic specifications.

This generality suggests the theorem will age well. Frameworks evolve; formal cores do not. The Bounded Stochasticity Theorem, properly understood, is the formal core of any approach to governing AI systems through specification. It is the statement that makes SDPF not merely a methodology but an infrastructure—the kind of infrastructure that certification bodies can certify against, that regulators can cite, that tool vendors can build to, and that independent implementers can verify conformance against.

13.4 Final Statement

The Bounded Stochasticity Theorem matters more than the framework it sits inside. The framework is a system of discipline; many frameworks aspire to discipline. What distinguishes SDPF is not the discipline itself but the formal property that makes the discipline defensible—the property that says, with precise conditions, that following this discipline produces outputs which conform to deterministic specifications even when produced by stochastic generators.

Remove the theorem and SDPF is one methodology among several. Keep it, and SDPF is the only specification framework with a principled account of why it works on the executors that now dominate software production.

The theorem has been stated. It has been proved. Its conditions are explicit. Its limitations are acknowledged. What remains is for the field—researchers, practitioners, regulators, and tool vendors—to recognize its implications and build upon its foundation.

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

14 References

[1] Abdullah, H. (2026). *The Bounded Stochasticity Theorem: Why It Is Relevant, Unique, and Important*. Software Development Prompting Framework.

[2] Abdullah, H. (2026). *SDPF Language Specification, Version 1.3.1*. Software Development Prompting Framework.

[3] Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 576–580.

[4] Dijkstra, E. W. (1975). *A Discipline of Programming*. Prentice Hall.

[5] Meyer, B. (1986). Design by Contract. In *Advances in Object-Oriented Programming* (pp. 1–50). Temple University.

[6] Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377–387.

[7] Saltzer, J. H., Reed, D. P., & Clark, D. D. (1984). End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4), 277–288.

[8] IEEE (2018). *ISO/IEC/IEEE 29148:2018 — Systems and software engineering — Life cycle processes — Requirements engineering*.

[9] ISO (2023). *ISO/IEC 25010:2023 — Systems and software engineering — SQuaRE — Product quality model*.

[10] NIST (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*.

[11] IETF (1997). *RFC 2119 — Key words for use in RFCs to indicate requirement levels*.

[12] NIST (2008). *FIPS 198-1 — The Keyed-Hash Message Authentication Code (HMAC)*.

Document prepared by Hamza Abdullah, SDPF Framework 2026. All mathematical definitions, proofs, and formal specifications derive from the SDPF Language Specification v1.3.1 and the original Bounded Stochasticity Theorem paper.

— End of Document —