Viewpoint-Structured Specification (VSS)

> A Framework for Structuring Intent into Traceable Specifications as the Basis for AI-Assisted Code Generation

Author: Spark Tsai
ORCID: https://orcid.org/0009-0006-8847-4703
Email: spark.tsai@gmail.com
Date: March 2026

Abstract

The rapid adoption of generative AI in software development has exposed a structural weakness in the artifacts used to communicate engineering intent. Traditional specifications—such as Software Design Documents and informal design notes—were designed for human authorship and linear development processes. In AI-assisted environments, however, code generation is inference-driven, asynchronous, and repeated across many generation events. Prompts and conventional documents therefore fail to provide a persistent, auditable basis for governing how code is produced.

This paper introduces **Viewpoint-Structured Specification (VSS)**, a structural framework for representing Intent as a versioned, multi-viewpoint specification prior to AI-assisted code generation. VSS formalizes a specification as a structured artifact $\mathrm{Specification} = (V, E)$, where V represents a set of declared Viewpoints and E represents persistently addressable specification elements produced under those viewpoints. Each element satisfies three sufficiency conditions—Persistent Addressability, Explicit Scope, and Viewpoint Membership—enabling specifications to function as durable governance artifacts rather than transient documentation.

From this structure, several capabilities emerge without additional mechanisms: Relation formation between elements, Semantic Conflict Detection across viewpoints, Traceability of generation decisions, and Boundary formation for selecting governing constraints during code generation. By decomposing intent into addressable units across multiple governing concerns, VSS enables latent conflicts in Intent to surface before code is generated and establishes a recoverable specification baseline that persists across system evolution.

VSS does not replace existing requirements engineering or model-driven methods. Instead, it provides the structural conditions under which their outputs can function as machine-selectable, version-stable governing constraints in AI-assisted development. The framework therefore reframes specification not as static documentation but as a persistent, structured artifact that governs the legitimate basis for code generation in AI-driven software production.

Keywords

Viewpoint-Structured Specification, Intent Structuring, AI-Assisted Development, Specification Formation, Development Governance, Boundary Formation, Semantic Conflict Detection, Traceable Artifact, Multi-Role AI Deliberation

1. Introduction

The adoption of generative AI in software development has created a structural asymmetry that existing methods were not designed to address. Code can now be produced faster than the organizational structures that govern its production can adapt. The result is not a tooling problem or a process problem—it is a specification problem: the artifacts that communicate intent to AI systems are structurally inadequate to serve as the basis for accountable, traceable, governed code generation.

This paper proposes Viewpoint-Structured Specification (VSS) as a structural remedy to this condition. VSS is a formal framework for decomposing Intent into a versioned, multi-viewpoint Specification prior to code generation—one that persists as a permanent, addressable artifact across the full lifecycle of the system it governs.

1.1 AI Code Production Has Exceeded Human Cognitive Bandwidth

Generative AI has fundamentally altered the economics of software production. Code can now be produced at a scale and speed that far exceeds human cognitive bandwidth. A single generation event can produce hundreds of lines of code in seconds; a single development session can produce thousands. The human capacity to review, understand, and govern what has been produced has not scaled proportionally.

This acceleration exposes a structural failure that faster review processes cannot resolve. The problem is not that humans cannot keep up with the volume of code being produced—it is that the mechanisms used to communicate governing intent to AI systems were designed for a production rate at which humans were the primary authors. Those mechanisms—Software Design Documents, informal specifications, and prompt instructions—are linear, content-centric, and single-use. AI-assisted code production is nonlinear, asynchronous, and inference-driven. The structural mismatch between the two is the condition that VSS is designed to address.

1.2 Two Structural Deficiencies in Current Methods

Current practice relies on two primary mechanisms for communicating intent to AI code generation systems. Both have structural deficiencies that cannot be resolved through better practice within their own artifact class.

Software Design Documents

The SDD is a linear document in which Specification is presented as flowing prose organized by section headings. Its boundary—the scope of concerns it addresses—is implicit in its document structure and cannot be dynamically selected. When a code generation event requires a specific subset of governing concerns rather than the full document, there is no structural mechanism for making that selection: the SDD is consumed as a whole or not at all. Its element identifiers are navigational conveniences, not persistent governance anchors. Its What and How concerns are mixed in the same sections, preventing independent versioning or independent selection of the two concern classes. Traceability is absent by construction: a document organized by section headings provides no anchor points against which selection decisions can be recorded.

Prompt Engineering

The prompt is a single-use contextual instruction optimized for immediate consumption. It is not versioned, not persistently addressable, and not auditable. Each generation event reconstructs its governing context from scratch; prior decisions leave no structural trace that can be referenced, compared, or selected in subsequent sessions. The absence of cross-session persistence means that the accumulated Intent of a system under continuous development is never available as a structured selection resource—it must be reconstructed, imperfectly and implicitly, each time. The prompt is the correct artifact class for single-use contextual instruction; it is the wrong artifact class for governing the multi-session, multi-generation lifecycle of an evolving system.

1.3 The Inspiration for VSS

The core mechanism of VSS was not derived from a theoretical model of specification. It was derived from the observation of how human engineering teams actually manage the structural complexity of system development.

Before a complex system is built, human teams do not produce a single unified specification from a single unified perspective. They convene—across functions, across roles, across concern boundaries—and the value of that convening is not the consensus it produces but the conflicts it surfaces. A security architect who hears a functional requirement for the first time in a cross-team meeting may immediately identify a constraint violation that no single-perspective review would have found. That tension, surfaced before implementation commitments are made, is the mechanism by which costly conflicts are resolved at the point where resolution is least expensive.

VSS replicates this dynamic at the specification formation stage, using AI multi-role assignment as the mechanism for deliberative tension. The correspondence between human team behavior and VSS mechanisms is direct:

Human Engineering Behavior	VSS Mechanism
Cross-functional meeting / Team coordination	Multi-Viewpoint structuring
Different role perspectives	Different Viewpoint classes

Human Engineering Behavior	VSS Mechanism
Conflicts surfacing in meeting	Semantic Conflict Detection
Meeting conclusions	Structured Specification
Execution division of work	Boundary (V', E') human selection

The deliberative tension between AI roles in VSS is not a side effect of the multi-role process—it is its governing purpose. Conflicts that would otherwise remain invisible until code review or production are structurally surfaced at specification time, where their resolution requires a declaration rather than a code modification.

1.4 Position of VSS in the Development Lifecycle

VSS operates at the Before Decision Behavior stage of the development lifecycle—the stage that precedes the formation of any Boundary decision and any code generation event.

```
Development Lifecycle
│
├─ Before Decision Behavior  ← VSS operates here
│   AI multi-role → Intent constraint → Structured Specification
│   Output: versioned (V, E), persistent, addressable
│
├─ Decision Behavior         ← DBG operates here
│   Human decision: Ticket → Directive
│                 → Boundary (V', E') selection from Specification
│
└─ Execution
    Code Generation → Anchor → Evidence
```

This positioning is not incidental. VSS addresses the upstream structural cause of the failures that Decision Behavior Governance [Tsai, 2026a] and Boundary as an Execution-Time Primitive [Tsai, 2026f] address downstream. Without a Viewpoint-Structured Specification, Decision Behavior operates on an unstructured premise; Boundary selection has no versioned (V, E) to select from; Anchor Architecture [Tsai, 2026e] has no addressable source artifacts to bind.

VSS does not govern decisions or execution. It governs the conditions under which decisions and execution can be structurally grounded. The distinction between governing code and governing the conditions for code generation is the structural contribution of this paper.

For the relationship between Development-stage and Runtime-stage governance, and the argument that runtime interception cannot substitute for Development-stage structural grounding, see Runtime Governance vs. Development Governance [Tsai, 2026d].

1.5 Three Claims

This paper advances three claims about what VSS provides and why it is structurally necessary.

Claim 1 — Observational: Multi-Viewpoint AI role assignment constrains Intent across multiple declared perspectives before code generation begins. The deliberative tension between AI roles is the mechanism through which latent conflicts in Intent become structurally observable. Pre-coding conflict surfacing is not achievable through single-viewpoint specification regardless of its quality or thoroughness.

Claim 2 — Causal: The formal structure $\mathrm{Specification} = (V, E)$, together with the three sufficiency conditions for unitization, establishes the structural preconditions from which four governance-enabling properties emerge necessarily: Relation, Semantic Conflict Detection, Traceability, and Boundary Formation. These properties are not features to be added to VSS—they are consequences of unitization itself.

Claim 3 — Ontological: A Viewpoint-Structured Specification is a permanent engineering fact—a structural condition that exists prior to execution and persists across the lifecycle of the system. It is not consumed by the generation events it informs. Its version state is the structural basis for every Boundary decision made against it, and its cross-version accumulation is the mechanism through which Development-stage accountability becomes possible.

The remainder of this paper is organized as follows. Section 2 establishes the three upstream problems that VSS addresses. Section 3 defines the Viewpoint as the structural unit of Intent constraint. Section 4 formalizes the Structured Specification as (V, E). Section 5 provides a taxonomy of Specification types instantiating the formal structure. Section 6 derives the four emergent properties. Section 7 establishes the Specification as a permanent governance artifact. Section 8 situates VSS within the downstream research framework. Sections 9 through 11 address discussion, conclusion, and related work.

2. Problem Statement

The adoption of generative AI in software development has not resolved the fundamental tension between intent and implementation—it has amplified it. While AI systems can produce syntactically valid and operationally functional code at scale, the structural conditions under which that code is generated remain largely ungoverned. This section identifies three upstream problems that VSS is designed to address: the inherent ambiguity of Intent, the non-permanence of prompt-based specification, and the invisibility of semantic conflicts prior to code generation.

2.1 The Ambiguity of Intent

Intent, as the originating motivation behind a software artifact, is inherently divergent.
A single statement of intent admits multiple interpretations, each emphasizing a different concern—functional

behavior, performance constraints, security boundaries, or architectural assumptions. When Intent is expressed through a single viewpoint, this interpretive diversity is collapsed into a single representation, leaving the remaining dimensions unobserved and uncontrolled.

This structural incompleteness has direct consequences in AI-assisted development. Because generative AI systems operate on the context made available to them, unrepresented dimensions of Intent are not merely ignored—they become invisible inputs to a generation process that will nonetheless produce outputs shaped by implicit assumptions about them. The result is a class of degradation phenomena unique to AI-generated artifacts.

Ghost Intent describes the condition in which executable artifacts are produced whose originating decisions cannot be recovered [Tsai, 2026g]. This is not equivalent to undocumented code, where intent may be implicit but inferrable from context. Ghost Intent is the structural absence of origin—a condition in which the artifact exists but the rationale for its existence is irrecoverable. Intent Fragmentation compounds this problem: even when intent is partially present, its structural dispersal across unrelated artifacts prevents it from functioning as a coherent governing constraint.

Both phenomena share a common upstream cause: Intent was never structurally decomposed before generation began. A single-viewpoint description cannot surface what it does not represent. The cost of this omission is not felt immediately—it accumulates across iterations, as each subsequent generation builds upon an increasingly opaque foundation.

2.2 The Non-Permanence of Prompt-Based Specification

The dominant mode of intent communication in current AI-assisted development is the prompt. Prompts are contextual, ephemeral, and consumable: they are constructed immediately before a generation event, used once, and discarded. Nothing in the prompt infrastructure is designed for persistence, versioning, or audit.

This non-permanence introduces what may be characterized as Authorial Discontinuity—the structural absence of a recoverable chain connecting each generation event to the intent that authorized it. In traditional software development, authorship carries implicit continuity: a developer who writes code at time T is presumed to carry forward knowledge of the decisions made at T-1. This continuity, however imperfect, provides the basis for incremental reasoning about system evolution.

In prompt-driven AI development, this continuity is severed by design. Each prompt session is stateless. The intent expressed in one session leaves no persistent structural trace that can be referenced, versioned, or verified in a subsequent session. As a result, the cumulative evolution of a system built through repeated prompt-driven generation lacks a recoverable specification baseline.

This is not a deficiency that better prompt management can resolve. The problem is not that prompts are poorly written—it is that prompts are the wrong artifact class for governing intent across time. A prompt is an instruction;

a specification is a commitment. The distinction is not stylistic but structural: specifications are versioned, addressable, and auditable; prompts are none of these things.

2.3 The Invisibility of Pre-Coding Semantic Conflicts

When Intent is expressed through a single viewpoint, or through an unstructured sequence of prompts, semantic conflicts between different dimensions of that intent have no mechanism through which to become visible before code generation begins. They are not suppressed or resolved—they are simply absent from the pre-coding context, and therefore unavailable as inputs to any governance process.

The Visibility Inversion Principle captures the asymmetry this creates: the earlier a conflict can be detected in the development lifecycle, the lower the cost of resolution—yet detection probability is inversely proportional to layer depth at generation time, while damage magnitude is directly proportional to the depth at which failure eventually manifests. Intent-layer conflicts that could have been surfaced at specification time instead propagate silently into structural and behavioral layers, where the cost of intervention increases exponentially.

Inference Creep provides a concrete illustration of this dynamic [Tsai, 2026b]. When AI systems generate code in the absence of explicitly declared intent boundaries, they do not halt at the boundary of the stated instruction—they infer what adjacent concerns ought to be addressed, expanding scope without explicit authorization. Inference Creep is not a model defect; it is a governance gap.
The AI is not malfunctioning—it is operating in the absence of a structural constraint that was never declared. The conflict between the stated instruction and the implied scope expansion exists prior to generation, but without a multi-viewpoint structure to make it observable, it remains latent until its consequences appear in executed behavior.

Existing approaches address this problem reactively: code review, testing, and runtime monitoring are all post-generation mechanisms. They can detect the consequences of unresolved pre-coding conflicts, but they cannot prevent them. VSS addresses the problem at its structural source: by requiring that Intent be expressed across multiple viewpoints before generation begins, it creates the conditions under which semantic conflicts become observable—and therefore resolvable—before any code is written.

These three problems share a common structural cause. Intent is expressed through artifacts that lack decomposition, persistence, and viewpoint diversity. Without these properties, intent cannot function as a governing constraint over code generation. The result is not merely poor documentation but the absence of a recoverable specification baseline. VSS is proposed as a structural remedy to this condition.

References for this section:

- [Tsai, 2026b] From Inference Creep to Risk Acceleration Pipelines. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146686

- [Tsai, 2026g] Ghost Intent: An Effect of Traceability Collapse
  in GenAI-Assisted SDLCs.
  Zenodo. https://doi.org/10.5281/zenodo.18872540

3. What Is a Viewpoint?

The central structural unit of VSS is the Viewpoint. Where Section 2 identified the absence of decomposition, persistence, and viewpoint diversity as the common cause of upstream specification failures, this section defines what a Viewpoint is, explains why a single viewpoint is structurally insufficient, and describes how multiple viewpoints—assumed by an AI acting in distinct roles—create the conditions under which Intent can be meaningfully constrained before code generation begins.

3.1 Definition of a Viewpoint

A Viewpoint is a declared semantic boundary representing a specific concern, constraint, or intent in a system.

Several properties of this definition require emphasis.

A Viewpoint is *declared*. It does not emerge implicitly from a body of text or inferred from implementation behavior. It is an explicit commitment to the existence of a specific concern, made prior to and independent of any generation event. This declarative character is what makes a Viewpoint governable: a boundary that has not been declared cannot be observed to have shifted.

A Viewpoint represents a *specific concern*. It is not a summary of the system, nor a container for all relevant information. Each Viewpoint addresses one dimension of Intent—a functional requirement, an architectural constraint, a security boundary, a test intent—and only that dimension. This specificity is the basis for the decomposition that Section 2 identified as absent from current practice.

Viewpoints are *not process phases*. They do not correspond to stages in a development lifecycle, and they do not require a fixed order of production. A Viewpoint addressing performance constraints may be declared before or after one addressing functional requirements, depending on where risk is located. The network of Viewpoints that constitutes a VSS Specification is organized by dependency and risk, not by methodology.

Examples of Viewpoints include, but are not limited to: functional requirements, non-functional constraints (privacy, security, safety, performance), architectural intent, data models and interfaces, state transitions and workflows, test intent and coverage logic, and usage scenarios and assumptions.

3.2 Why a Single Viewpoint Is Structurally Insufficient

A single Viewpoint can only represent one cross-section of Intent. This is not a limitation of how thoroughly the Viewpoint is expressed—it is a structural consequence of what a Viewpoint is. A security-focused Viewpoint will characterize the system in terms of threat surfaces, access constraints, and data boundaries. An architectural Viewpoint will characterize the same system in terms of component responsibilities, dependency directions, and extensibility assumptions. These are not redundant descriptions of the same thing; they are irreducibly distinct representations of different governing concerns.

When Intent is expressed through only one of these representations, the others are not merely underspecified—they are absent. The generation process that follows will produce outputs shaped by implicit assumptions about the missing dimensions, assumptions that were never declared and therefore cannot be governed. Conflicts between these undeclared dimensions and the declared one will not surface at specification time; they will surface later, in code review, in test failures, or in production incidents, at which point the cost of resolution has compounded significantly.

A single-viewpoint specification does not fail by misrepresenting Intent. It fails by representing too little of it, leaving the remainder to be resolved—silently and without accountability—by the generation process itself.

3.3 Multi-Viewpoint AI Role Assignment as an Intent Convergence Mechanism

VSS addresses this structural insufficiency by requiring that Intent be expressed across multiple Viewpoints before any code generation occurs. In practice, this is achieved through a mechanism that has no direct precedent in traditional specification methods: the AI is assigned distinct roles, each corresponding to a different Viewpoint, and each role produces an independent characterization of the same underlying Intent.

This mechanism is not merely additive. When the same Intent is described by an AI acting as a security architect and simultaneously by an AI acting as a functional requirements analyst, the two descriptions will not always agree. Functional requirements may assume data access patterns that security constraints prohibit. Architectural assumptions may conflict with test coverage requirements. These conflicts are not introduced by the multi-role process—they were already present in the Intent. The multi-role process makes them structurally visible for the first time.

> In VSS, each Viewpoint is assumed by the AI acting in a distinct role. The deliberative tension between roles is the mechanism through which latent conflicts in Intent become structurally observable.

This deliberative dynamic is the computational analog of cross-functional team review in human engineering practice. In human organizations, the value of involving multiple stakeholders in specification is precisely that different roles bring different governing concerns into contact with one another, forcing conflicts to surface before implementation commitments are made. VSS replicates this dynamic at the specification formation stage, using role differentiation as the mechanism for conflict surfacing rather than as a social coordination process.

The output of this multi-role process is not a single unified document but a structured collection of Viewpoint Artifacts—independently versioned, explicitly scoped declarations of semantic commitment, each anchored to a specific role and concern. The relationships between these artifacts, including alignment, dependency, and conflict, form the structural basis for the governance mechanisms described in subsequent sections.

3.4 The Risk-Driven Principle

The set of Viewpoints that constitute a VSS Specification is not prescribed by methodology. There is no fixed list of Viewpoints that every specification must include, no mandated sequence of production,
and no minimum number of roles that must be assumed. These are determined by the risk profile of the system under development.

> Viewpoints emerge and evolve in response to risk, not methodology.

This principle has two practical implications.

First, Viewpoints are not created speculatively. A Viewpoint is declared when a specific concern, constraint, or intent requires explicit representation because the consequences of leaving it unrepresented are materially significant. Low-risk concerns may not require dedicated Viewpoints; high-risk concerns—particularly those involving safety, security, or irreversible architectural commitments—may require multiple.

Second, Viewpoints evolve as risk evolves. A Viewpoint that adequately represents a concern at one stage of system development may become insufficient as the system grows, as deployment context changes, or as new failure modes are identified. The versioned character of Viewpoint Artifacts is designed to accommodate this evolution without discarding the governance continuity that prior versions provide.

Third, governance density is proportional to risk. A simple, low-stakes intent may require only one or two Viewpoints to be adequately governed; a core architectural decision or a security-critical constraint may require five or six. VSS does not prescribe a uniform level of structural coverage across all intents. It requires only that the level of coverage be commensurate with the consequences of leaving a concern ungoverned. This proportionality is what makes VSS tractable in practice: engineers are not required to produce exhaustive multi-viewpoint specifications for every decision, only for those where the cost of undeclared conflict materially exceeds the cost of structured decomposition.

This risk-driven character distinguishes VSS from methodology-driven specification frameworks, which prescribe a fixed set of document types regardless of the risk profile of the system being built. VSS does not replace those frameworks; it provides a structural foundation that can accommodate them as particular instantiations of a more general principle.

4. Structured Specification

Section 3 established that Intent must be expressed across multiple Viewpoints, and that the deliberative tension between AI roles is the mechanism through which latent conflicts become observable. This section addresses the next question: what structural conditions must a collection of Viewpoints and their associated elements satisfy in order to function as a governing constraint over code generation?

The answer is formalized here as a Structured Specification—a two-tuple (V, E) whose properties enable the four emergent governance capabilities described in Section 6. The section proceeds by defining the formal structure, specifying the sufficiency conditions for unitization, drawing the critical distinction between Scope and Boundary, and contrasting VSS with Software Design Documents to clarify the structural gap that VSS is designed to close.

4.1 Formal Definition of a Structured Specification

A Structured Specification is defined as:

$$\text{Specification} = (V, E)$$

where:

$$V = \text{set of viewpoints}$$

$$E = \text{set of specification elements}$$

subject to the Membership Constraint:

$$\forall e \in E,\ \exists\, v \in V : e \in v$$

This definition makes three structural commitments.

First, a Specification is not a document—it is a set. V is a set of declared Viewpoints, each representing a specific governing concern as defined in Section 3. E is a set of specification elements, each representing a discrete, addressable unit of semantic commitment produced under a specific Viewpoint. The document metaphor, with its implicit assumptions of linear structure and unified authorship, is deliberately discarded.

Second, the Membership Constraint is non-negotiable. Every element in E must belong to at least one Viewpoint in V. An element that cannot be attributed to a declared Viewpoint is not a specification element—it is an unanchored assertion, structurally equivalent to a comment in code: present, potentially informative, but outside the governance boundary of the Specification.

Third, V is not fixed. As established in Section 3.4, Viewpoints emerge in response to risk. The set V may grow as new concerns are identified, and individual Viewpoints may be versioned independently as their governing scope evolves. The Specification is therefore a living structure, not a static document—but its evolution is traceable, because every element remains anchored to a Viewpoint, and every Viewpoint is explicitly declared.

4.2 Sufficiency Conditions for Unitization

The formal definition establishes what a Structured Specification contains. The sufficiency conditions establish what each element in E must satisfy in order for the Specification to be structurally governable. Unitization—the decomposition of Intent into discrete, independently addressable elements—is the foundational act of structuring. Without it, the properties described in Section 6 cannot emerge.

Each element $e \in E$ must satisfy three conditions:

(1) Persistent Addressability

Each element must have a unique identifier that remains stable across versions of the Specification:

$$\forall e \in E,\ \exists\, \mathrm{id}(e) \text{ such that } \mathrm{id}(e) \text{ is unique and version-stable}$$

Persistent Addressability enables an element to be referenced from outside the Specification—by other elements, by downstream artifacts, by Boundary selection decisions, and by audit records. An element that cannot be stably referenced cannot be traced, and an element that cannot be traced cannot be governed.

This condition distinguishes VSS elements from sections in a conventional specification document. A section heading is a navigational convenience; a persistent identifier is a governance commitment. When a Specification is versioned, the identifiers of its elements persist across versions, allowing changes to be detected as modifications to known elements rather than as wholesale replacements of unaddressable text.

(2) Explicit Scope

Each element must declare the range of concerns it covers. Scope is not inferred from position in a document structure, from the heading under which an element appears, or from surrounding context. It is stated explicitly as part of the element's definition:

$$\forall e \in E,\ \mathrm{scope}(e) \text{ is explicitly declared}$$

Explicit Scope serves two governance functions. First, it makes the element selectable: when a Boundary decision is made at execution time, the human making that decision can determine whether a given element is relevant to the task at hand because the element's scope is stated, not implied. Second, it makes overlap and conflict detectable: when two elements in different Viewpoints have scopes that intersect, that intersection is structurally visible and can be examined for consistency.

(3) Viewpoint Membership

Each element must belong to at least one declared Viewpoint in V:

$$\forall e \in E,\ \exists\, v \in V : e \in v$$

This condition is a restatement of the Membership Constraint at the element level: it is not sufficient for V and E to coexist in a Specification—each element must be explicitly anchored to the Viewpoint under whose governing

concern it was produced.

Viewpoint Membership is the structural property that makes Semantic Conflict Detection possible. As formalized in Section 6.2, a conflict between two elements is detected by comparing elements that belong to different Viewpoints. Without explicit Viewpoint Membership, there is no basis for this comparison: elements are simply a flat collection of assertions, with no structural information about the governing perspective from which each was produced.

4.3 Scope Is Not Boundary

The distinction between Scope and Boundary is among the most consequential in VSS, and conflating them is among the most common structural errors in specification practice.

> Structured Specification declares Scope, not Boundary. Boundary emerges from human decision at execution time, using Specification Version as the selection basis. Conflating the two collapses specification into directive, eliminating the governance properties VSS is designed to preserve.

Scope is a property of a specification element $e \in E$, declared at authoring time and stable as part of the element's definition.
It answers: *what range of concerns does this element govern?*

Boundary is defined over a subset selection from (V, E) at
execution time:

$$\text{Boundary} = (V', E') \text{ where } V' \subseteq V,\ E' \subseteq E$$

$$\forall e \in E',\ \exists\, v \in V' : e \in v$$

Boundary answers a different question: *which elements of the Specification constitute the governing context for this particular act of code generation?*

The two questions have different answers, different times of resolution, and different agents of determination. Scope is determined by the AI role that produces the element, at specification time. Boundary is determined by the human engineer who selects elements from the Specification, at execution time, using the current Specification Version as the basis for that selection.

When Boundary is embedded in the Specification itself—as it is in Software Design Documents, where the document's structure implicitly defines which concerns are in scope for which implementation decisions—the Specification has become a directive. A directive does not support dynamic selection; it prescribes a fixed mapping from concern to implementation. This eliminates the governance property that makes VSS tractable at scale: the ability to select different Boundaries from the same specification for different generation events, with each selection traceable back to a versioned Specification state.

4.4 Contrast with Software Design Documents

The structural gap between VSS and conventional Software Design Documents (SDD) can be stated precisely. SDDs are not inadequate specifications—they are a different artifact class, optimized for human readability and sequential authorship rather than for machine-assisted selection and multi-viewpoint governance.

The following comparison maps the structural properties of each approach against the conditions that VSS requires:

Property	SDD	VSS
Identity	One ID per document	One ID per element
Scope	Implied by chapter structure	Explicitly declared per element
Class separation	What and How mixed	Semantic dimension (What) / Engineering dimension (How) separated
Production mode	Human-authored	AI multi-role driven
Boundary	Implicit in document structure	Human decision at execution time
Relation	Embedded in prose	Emergent from element structure
Traceability	Absent	Selection behavior is recordable

Several of these differences merit elaboration.

The mixing of What and How in a single SDD is not a stylistic choice—it is a structural constraint that prevents the two dimensions of concern from being independently versioned, independently selected, or independently governed. When a functional requirement and an architectural constraint appear in the same document section, their governance histories are coupled: a change to one cannot be traced independently of the other. VSS separates these into the Semantic dimension and the Engineering dimension, expressed formally as:

$$V = V_S \cup V_C, \quad V_S \cap V_C = \emptyset$$

where V_S contains Semantic Viewpoints—governing *what* the system is—and V_C contains Engineering Viewpoints—governing *how* the system is permitted to be built—enabling independent governance of each.

The treatment of Relation and Traceability as emergent properties in VSS—rather than as explicitly authored content, as they are in SDD cross-references—reflects the formalization in Section 4.1. Because every element in E has a persistent identifier $\mathrm{id}(e)$ and an explicit Viewpoint membership, the structural conditions for detecting relationships and recording selection events already exist. These capabilities do not need to be built into the Specification; they arise from the unitization that the Specification enforces.

The absence of Traceability in SDD is not a missing feature—it is a structural consequence of the document model. A document that presents Intent as flowing prose, organized by section headings without persistent element identifiers, provides no anchor points against which selection decisions can be recorded. VSS provides those anchor points by design.

5. Specification Taxonomy

The formal structure established in Section 4 defines what a Structured Specification must satisfy—but it does not prescribe which Viewpoints must exist. This section provides a practical instantiation of that structure through a taxonomy of Viewpoint classes observed in AI-assisted development practice.

> The following taxonomy is not prescriptive. It represents a structural response to specific governance gaps observed in current AI-assisted development practice. Other Viewpoint classifications may emerge in response to different risk profiles, organizational contexts, or system types.

The taxonomy organizes Viewpoints into two mutually exclusive dimensions: the Semantic dimension, governing what the system is, and the Engineering dimension, governing how the system is permitted to be built. Each dimension addresses a class of structural gap that current methods leave ungoverned.

The six Specification types defined in this section were not designed from first principles. They were identified through practical engagement with recurring problems in AI-assisted development, and resolved by returning to established software engineering theory to locate existing conceptual frameworks capable of addressing each problem. The approach was consistently one of recovery rather than invention: when a structural problem surfaced, the question asked was not what new artifact should be created, but which existing engineering perspective, already developed within the discipline, had been overlooked or left unformalized in current practice. These specifications do not introduce new engineering concepts; they formalize engineering perspectives that already exist in practice but are rarely expressed as machine-selectable artifacts.

One representative example motivates the inclusion of the Engineering dimension. In AI-assisted development, switching models or performing broad refactoring operations frequently causes the code architecture to be regenerated without reference to prior structural decisions. The result is that architectural intent, which was never explicitly declared as a standalone artifact, is silently overwritten. This problem is not new—software architecture has been a recognized discipline for decades—but the existing body of architectural notation, including UML diagrams and informal design documents, was not designed to function as a machine-selectable, version-stable governance artifact. The Engineering dimension addresses this by requiring that architectural intent be expressed as structured, addressable text specifications rather than as diagrams or prose embedded in documents that AI systems cannot reliably reference or trace.

The six types presented here reflect the author's experience with specific classes of structural failure in AI-assisted development practice. They are not proposed as a complete or authoritative classification. They represent the subset of existing software engineering perspectives that proved necessary and sufficient to

address the structural gaps encountered. Future AI-assisted development practice may surface new classes of structural failure not captured by this taxonomy. When that occurs, the appropriate response is the same as the one taken here: examine the existing body of software engineering theory for perspectives that address the new failure mode, and formalize the relevant perspective as a Viewpoint within the VSS structure. The taxonomy is a starting point, not a boundary.

5.1 Semantic Dimension — What the System Is

The Semantic dimension (V_S) contains Viewpoints that govern the intended meaning and behavior of a system, independent of how that meaning is realized in implementation. These Viewpoints answer the question: *what is the system supposed to do, and under what conditions?*

The structural gap that the Semantic dimension addresses is the absence of structured declarations of Intent at the meaning layer. In prompt-driven development, semantic intent is expressed as natural language instruction and consumed immediately. It leaves no persistent, versioned, addressable artifact that can be referenced by subsequent generation events or audited after the fact. The Semantic dimension provides that artifact class.

The following Specification types instantiate the Semantic dimension within the VSS taxonomy:

SRS — Software Requirements Specification

Governs the functional and non-functional requirements of the system: what the system must do, what constraints it must satisfy, and under what conditions its behavior is defined. SRS Viewpoints establish the legitimacy boundary for functional intent—the declared basis against which generated behavior can be compared for alignment.

SDS — Software Design Specification

Governs the architectural and structural intent of the system: how components are organized, how responsibilities are distributed, and what design decisions have been made prior to implementation. SDS Viewpoints make architectural intent explicit and addressable, preventing the condition in which architectural assumptions are embedded implicitly in generated code without recoverable declaration.

STS — Software Test Specification

Governs the intent behind test coverage: what behaviors must be verified, under what conditions, and to what standard. STS Viewpoints establish test intent as a first-class semantic commitment, not as a post-generation artifact. When test intent is declared before code generation, conflicts between functional requirements and testability constraints become structurally observable at specification time.

5.2 Engineering Dimension — How the System Is Permitted to Be Built

The Engineering dimension (V_C) contains Viewpoints that govern the constraints on implementation practice, independent of what the system is intended to do. These Viewpoints answer the question: *within what structural and engineering boundaries must implementation proceed?*

The structural gap that the Engineering dimension addresses is the conflation of semantic intent with implementation constraint in conventional specifications. When a document states both what a system must do and how it must be built, these two classes of constraint share a governance history: they are versioned together, referenced together, and selected together. This coupling prevents either class from being governed independently, and makes it structurally impossible to detect conflicts between semantic requirements and engineering constraints before code generation begins.

The following Specification types instantiate the Engineering dimension within the VSS taxonomy:

CAS — Coding Architecture Specification

Governs the structural constraints on code organization: module boundaries, dependency directions, layering rules, and architectural patterns that are permitted or prohibited. CAS Viewpoints establish the engineering envelope within which generated code must be situated, making violations of architectural intent structurally detectable rather than discovered incidentally during review.

CIS — Conceptual Implementation Specification

Governs the conceptual approach to implementation: the abstractions, patterns, and implementation strategies that have been authorized for a given context. CIS Viewpoints prevent the condition in which AI systems select implementation strategies that are locally valid but inconsistent with the broader conceptual model of the codebase—a form of Inference Creep at the implementation strategy level.

CSS — Coding Structure Specification

Governs the structural conventions of the codebase at the code level: naming conventions, file organization, interface design patterns, and structural consistency rules. CSS Viewpoints make structural conventions explicit and addressable, enabling generated code to be evaluated against declared structural standards rather than against implicit institutional expectations.

5.3 The Complementary Relationship Between Dimensions

The Semantic and Engineering dimensions are mutually exclusive and jointly exhaustive within the VSS taxonomy:

$$V = V_S \cup V_C, \quad V_S \cap V_C = \emptyset$$

V_S governs what the system is. V_C governs how the system is permitted to be built. Together, they form a complete Intent constraint network: no specification element that governs semantic meaning belongs to the Engineering dimension, and no element that governs implementation constraint belongs to the Semantic dimension.

This separation has a direct governance consequence. A conflict between a semantic requirement and an engineering constraint—for example, a functional requirement that implies a dependency direction prohibited by the Coding Architecture Specification—is only detectable if the two concerns are represented in structurally distinct Viewpoints. When they coexist in a single document, as they do in conventional SDDs, the conflict is invisible at specification time. When they are separated into V_S and V_C, the conflict becomes an instance of Semantic Conflict Detection as formalized in Section 6.2:

$$\text{Conflict}(v_s, v_c) = \exists\, e_1 \in v_s,\ e_2 \in v_c : \text{semantic}(e_1) \otimes \text{semantic}(e_2) = \emptyset$$

where $v_s \in V_S$ and $v_c \in V_C$.

The mutual exclusivity of the two dimensions also enables independent versioning. A change to a functional requirement does not necessitate a version increment in the Engineering dimension; a change to a structural convention does not affect the semantic intent layer. Each dimension evolves at its own rate, governed by the risk profile of the concerns it represents, while the governance relationship between them remains structurally observable through the Conflict Detection mechanism.

It bears repeating that this taxonomy is not exhaustive. The six Specification types defined here—SRS, SDS, STS, CAS, CIS, CSS—represent a response to observed structural gaps in AI-assisted development practice, resolved by drawing on perspectives already present within established software engineering theory. They are not a closed set. As AI-assisted development practice evolves and new classes of structural failure are identified, the appropriate path remains the same: locate the existing engineering perspective that addresses the failure, and formalize it as a Viewpoint within the VSS structure. The formal definition of VSS as (V, E) places no constraint on the composition of V beyond the Membership Constraint and the sufficiency conditions for unitization.

6. Emergent Properties of Structuring

The formal structure established in Section 4 and instantiated in Section 5 does more than organize Intent into addressable units. Once a Specification satisfies the conditions of (V, E)—once every element has a persistent identifier, an explicit scope, and a declared Viewpoint membership—four capabilities become available that were structurally impossible before unitization occurred.

> Structuring is fundamentally an act of unitization. The following four properties are not imposed on a specification—they emerge when units are properly formed.

These properties are not features to be added after the fact. They are direct consequences of the structural decisions already made in Sections 4 and 5. No additional mechanism is required to produce them; they arise from unitization itself. Their significance is not that they enable oversight—it is that they demonstrate the structural completeness of a Viewpoint-Structured Specification as the basis for AI-assisted code generation.

6.1 Relation

When every element in E has a persistent identifier and a declared Viewpoint membership, elements can be related to one another explicitly and traceably. A Relation is a declared connection between two elements $e_1, e_2 \in E$, expressing that the concern of one is structurally dependent on, consistent with, or in tension with the concern of the other.

$$\text{Relation}(e_1, e_2) \subseteq E \times E$$

Relations are not embedded in the prose of a specification document. They are not cross-references or footnotes. They are first-class structural connections between addressable units, each of which retains its own persistent identifier and Viewpoint membership regardless of how many Relations it participates in.

This property is emergent rather than designed because it requires nothing beyond what the Membership Constraint already establishes. An element that has a persistent identifier can be referenced. An element that can be referenced can be related. The infrastructure for Relation exists the moment unitization is complete; what remains is only the act of declaration.

The practical consequence is that changes to one element propagate structurally through its declared Relations. When element e_1 is modified in a new Specification version, every element e_2 that holds a declared Relation to e_1 is immediately identifiable as a candidate for review. Change impact analysis becomes tractable at specification time, before any code generation has occurred.

6.2 Semantic Conflict Detection

The second emergent property is the structural observability of semantic conflicts between Viewpoints. A conflict is not a discrepancy in wording or a gap in documentation—it is a condition in which two elements, each representing a legitimate concern under its respective Viewpoint, make mutually incompatible demands on the same aspect of the system.

Formally:

$$\text{Conflict}(v_1, v_2) = \exists\, e_1 \in v_1,\ e_2 \in v_2 : \text{semantic}(e_1) \otimes \text{semantic}(e_2) = \emptyset$$

where $\otimes$ denotes semantic incompatibility—the condition in which the constraints expressed by e_1 and e_2 cannot be simultaneously satisfied.

This property is emergent because conflict detection requires Viewpoint membership to be explicit. Without it, elements are a flat collection of assertions with no structural information about the perspective from which each was produced. There is no basis for comparing them as representatives of distinct concerns. Once Viewpoint membership is declared, the comparison becomes structurally well-defined: a conflict is a pair of elements from different Viewpoints whose semantic constraints are incompatible.

> Multi-Viewpoint structuring is not merely descriptive—it is a pre-coding conflict surfacing mechanism. Conflicts that remain latent in single-viewpoint specification become structurally observable when the same Intent is constrained across multiple Viewpoints by distinct AI roles.

The significance of this property is captured by the Visibility Inversion Principle: the cost of resolving a conflict is lowest at the Intent layer and increases exponentially as it propagates into structural and behavioral layers. A semantic conflict surfaced at specification time—before any code generation has occurred—can be resolved through a declaration. The same conflict surfaced during code review requires code modification. The same conflict surfaced in production requires incident response. Semantic Conflict Detection relocates resolution to the point where it is least expensive.

It is important to note that conflicts detected through this mechanism are not errors in the Specification. They are evidence that the multi-Viewpoint process is functioning correctly: distinct AI roles, each faithfully representing its assigned concern, have produced elements whose constraints are in genuine tension. That tension existed in the Intent before VSS was applied. VSS makes it visible. Resolving it is a human decision about meaning, not a correction of a defect.

6.3 Traceability

The third emergent property is the recordability of selection behavior. When an element $e \in E$ is selected as part of a Boundary decision—included in the context for a specific code generation event—that selection can be recorded as a traceable event: which element was selected, from which Viewpoint, at which Specification version, for which generation task.

$$\text{Trace}(e,\ v,\ S_n,\ t) \quad e \in E,\ v \in V,\ S_n = \text{Specification at version } n,\ t = \text{task}$$

This property is emergent because it requires only that elements be persistently addressable—a condition already established by the sufficiency conditions of Section 4.2. An element that has a stable identifier can be referenced in a selection record. The infrastructure for Traceability exists the moment Persistent Addressability is satisfied.

Traceability serves as the structural input to Anchor Architecture [Tsai, 2026e]. AA is a general-purpose infrastructure for binding semantic commitments to implementation loci across all artifact types—Specifications, code, tests, Directives, and Evidence. VSS is one consumer of AA's anchoring mechanism, not its only one. The Traceability that emerges from VSS unitization provides AA with the addressable source artifacts that its anchoring operations require.

The practical consequence of Traceability is the transformation of selection decisions from implicit operational acts into auditable, attributable facts. In prompt-driven development, the choice of which context to provide to a generation event is made implicitly and leaves no persistent record. In VSS, the same choice is made against a versioned Specification and recorded as a selection of specific elements with specific identifiers. This record is what makes each code generation event attributable to a declared state of Intent.

6.4 Boundary Formation

The fourth emergent property is the capacity to form a Boundary—a structured, human-determined selection from the Specification that constitutes the context for a specific code generation event. Boundary is not defined within the Specification, and it is not produced automatically by any mechanism within VSS. It is the result of a deliberate human decision, made at execution time, using the current Specification Version as the structural basis for that decision.

Formally:

$$\text{Boundary} = (V', E') \quad \text{where} \quad V' \subseteq V,\ E' \subseteq E$$

$$\forall e \in E',\ \exists\, v \in V' : e \in v$$

The constraint $\forall e \in E',\ \exists\, v \in V' : e \in v$ ensures that Boundary is not an arbitrary subset of elements. Every element selected into a Boundary must be supported by a Viewpoint that is also selected. A selection that includes an element without its governing Viewpoint imports a semantic commitment without the perspective that authorizes it—a structurally invalid condition.

The Specification Version is the selection basis. The version determines which elements exist in E and which Viewpoints exist in V at the time of the decision. This version-dependence ensures that every Boundary decision is attributable to a specific declared state of Intent, and that changes to the Specification are structurally reflected in the Boundaries that subsequent decisions can form.

A representative decision scenario illustrates the mechanism:

$$\text{Given: two Tickets } T_1 \text{ and } T_2, \text{ each affecting distinct } (V', E') \text{ subsets}$$

$$\text{Human decision:} \begin{cases} \text{Merge into one Directive} \Rightarrow \text{single Boundary covers } (V_1' \cup V_2',\ E_1' \cup E_2') \\ \text{Separate into two Directives} \Rightarrow \text{each Directive retains independent Boundary} \end{cases}$$

The decision of whether to merge or separate is not resolvable by examining the Tickets alone. It requires examining the current Specification Version to determine whether the (V', E') subsets implied by T_1 and T_2 are structurally compatible—whether their Viewpoint memberships are consistent and their element scopes are non-conflicting. If compatible, a single Boundary can govern both without introducing unresolved semantic tension. If not, separate Boundaries are required to maintain the integrity of each context.

This mechanism produces three properties that are unavailable in unstructured development:

First, Boundary is not arbitrary. It is a structured selection from a versioned Specification, constrained by the Viewpoint membership of every element it contains.

Second, Boundary decisions are reproducible. Given the same Specification Version and the same task, the structural basis for selection is identical. Two engineers examining the same Specification at the same version evaluate the same (V, E) structure.

Third, Specification version upgrades are structural events. When the Specification is versioned, the (V, E) available for selection changes. Decisions made against a prior version are attributable to that version; decisions made against the current version reflect the current state of declared Intent. The version history of the Specification is therefore a record of how the authorized selection basis has evolved over time.

Reference: Boundary as an Execution-Time Primitive [Tsai, 2026f]

7. Specification as Permanent Governance Artifact

The four emergent properties established in Section 6 are structural capabilities—they describe what becomes possible once Intent has been unitized into (V, E). This section examines what those capabilities collectively produce: a Specification that is not consumed by the generation events it informs, but persists as a versioned, addressable, auditable record of declared Intent across the full lifecycle of a system.

This persistence is not incidental. It is the property that distinguishes a Viewpoint-Structured Specification from every other artifact class currently used to communicate intent to AI systems—and it is the property that makes VSS a structural basis for Development-stage accountability rather than a more sophisticated prompting technique.

7.1 Prompt vs Specification: A Structural Distinction

The comparison between prompts and Specifications is not a comparison of quality or thoroughness. It is a comparison of artifact class. A prompt and a Specification are structurally different objects, optimized for different purposes, with fundamentally different relationships to the systems they inform.

Dimension	Prompt	VSS Specification
Production mode	Human-authored input	AI multi-role driven
Lifecycle	Single-use, consumed at generation	Permanent, persists across generations
Versioning	None	Independently versioned

Dimension	Prompt	VSS Specification
Auditability	None	Selection behavior is recordable
Traceability	Not recoverable	Attributable to declared Intent
Boundary basis	None	Specification Version
Cross-session consistency	None	Preserved through version stability

A prompt is an instruction optimized for immediate consumption. Its value is entirely realized at the moment of generation; once the generation event is complete, the prompt leaves no persistent structural trace that can be referenced, versioned, or examined in relation to subsequent generation events. This is not a deficiency of prompt engineering—it is the intended behavior of an artifact class designed for single-use contextual communication.

A Specification is a commitment optimized for persistence. Its value is not fully realized at any single generation event; it accumulates across the lifecycle of the system, as successive generation events draw on the same versioned structure and as changes to that structure are recorded against a stable baseline of declared Intent. The distinction is not stylistic. It is the difference between an artifact that disappears after use and one that remains as a reference point for every subsequent act of generation, selection, and review.

The practical consequence is cross-session consistency. In prompt-driven development, two engineers generating code for related tasks in separate sessions may operate on entirely different implicit contexts, with no structural mechanism to ensure that their outputs are consistent with the same governing Intent. In VSS, both engineers select their Boundaries from the same versioned Specification. Their selections may differ, but the structure against which both selections are made is identical. This shared structural reference is what makes coordination tractable at scale.

7.2 Specification Version as the Structural Basis for Boundary Decisions

A Specification Version is not a document revision. It is a declared state of the governing structure (V, E): a specific set of Viewpoints, each containing a specific set of elements, each with its declared scope and Viewpoint membership, at a specific point in the evolution of the system's Intent.

This distinction matters because it determines the structural basis for Boundary decisions. When an engineer forms a Boundary (V', E') for a code generation event, the selection is made against a specific Specification Version. The version determines:

- which Viewpoints exist in V and are therefore available for inclusion in V'
- which elements exist in E under each Viewpoint and are therefore available for inclusion in E'

- which Relations between elements are declared and therefore relevant to the compatibility assessment of the proposed Boundary

Human decisions about how to structure Directives—whether to merge multiple Tickets into a single Boundary or to treat them as independent—cannot be made coherently without reference to a specific Specification Version. The version is the shared structural context that makes the compatibility of two (V', E') subsets assessable. Without it, the decision reduces to judgment without structural grounding.

Version upgrades are therefore structural events, not administrative acts. When the Specification is versioned, the (V, E) available for Boundary selection changes. Decisions made against the prior version remain attributable to that version; decisions made after the upgrade reflect the updated state of declared Intent. The version history of the Specification is the record of how the authorized basis for code generation has evolved—a record that no sequence of prompts can produce, because prompts leave no persistent structure to version.

Furthermore, the permanence of Viewpoint-Structured Specifications enables cross-version context accumulation. When forming a Boundary for a Directive, an engineer may draw not only from the current Specification Version but from prior versions whose declared (V, E) remain relevant to the task at hand:

$$\text{Boundary} = (V', E') \quad \text{where} \quad V' \subseteq \bigcup_i V_i, \quad E' \subseteq \bigcup_i E_i \quad \text{across selected versions } i$$

This cross-version selection serves two purposes.

First, it enables structural reuse. Previously declared Viewpoints and elements do not need to be re-declared in every new Specification Version. Their authority as governing constraints persists as long as they remain relevant to the current generation context. The accumulated declaration history of the Specification is directly available as a selection resource at every subsequent Boundary decision.

Second, it structurally constrains Inference Creep. Inference Creep arises when AI systems generate beyond explicitly declared boundaries, inferring what ought to be included without authorization [Tsai, 2026b]. When a Boundary incorporates historical (V, E)—prior architectural decisions, earlier constraint declarations, previously resolved conflicts—the authorized generation context is structurally richer. The space available for unauthorized inference is correspondingly narrower. Inference Creep is not eliminated by this mechanism; it is structurally bounded by the accumulated declaration history that the Specification preserves.

This constraint is only possible because the Specification is permanent. A prompt-based context cannot accumulate; it is reconstructed from scratch at each generation event, leaving prior decisions structurally invisible to the current generation. A Viewpoint-Structured Specification accumulates by design. Each version adds to a persistent structure that remains selectable, reusable, and constraining—not as a record of what was decided, but as an active resource for what may be authorized.

7.3 Specification as an Engineering Fact

The distinction between an administrative claim and an engineering fact is central to understanding what VSS produces and why it matters for Development-stage accountability.

An administrative claim is a declaration of intent: a policy document, a code of conduct, a risk register. Administrative claims describe what an organization intends to do. They are necessary for articulating intent, but they are structurally decoupled from the generation events they purport to govern. The presence of a policy document does not establish that any specific decision was formed within the boundaries that the policy describes.

An engineering fact is a structural condition that exists prior to execution and determines what decisions are authorized to occur. It does not describe intent; it instantiates it. The difference is not one of documentation quality—it is one of structural existence. A governance regime grounded in engineering facts can demonstrate that its boundaries were present before a decision was formed. A regime grounded in administrative claims can only assert that it intended them to be.

A Viewpoint-Structured Specification is an engineering fact in this sense. It is a versioned, addressable structure that exists prior to any generation event that references it. When a Boundary decision is made against a specific Specification Version, the structure that authorized that selection was present before the selection occurred. The Intent that governed the generation event was declared, not assumed. This is the structural condition that makes attributability possible—and attributability is the condition that Development-stage accountability requires.

This reframing has direct implications for how accountability is established in AI-assisted development. The question is not whether a policy was documented or whether a review process was followed. The question is whether the Intent that governed a code generation event was structurally declared before that event occurred, and whether the selection of governing context was made against a versioned, attributable structure. A Viewpoint-Structured Specification answers both questions affirmatively, by construction.

Reference: Toward Decision Behavior Governance [Tsai, 2026a]

8. Relation to Downstream Framework

VSS is the upstream component of a broader research framework addressing structural failures in AI-assisted software development. This section situates VSS within that framework by clarifying its relationship to each companion work. All relationships described here are of the Reference type: VSS provides structural outputs that downstream works consume or build upon, but VSS does not depend on any of them to function as a complete and independent framework. Its formal definition, sufficiency conditions, and emergent properties are self-contained within Sections 4 through 7.

8.1 Framework Position

The following diagram represents the structural position of VSS within the broader research program:

```
Taxonomy of Degradation Phenomena
 (companion work in preparation)
      ↓
      Identifies the class of structural failures
      that upstream intervention is required to address
      ↓
VSS — Viewpoint-Structured Specification
 (this paper)  ← Before Decision Behavior
      ↓
      Provides the structured specification library
      and versioned selection basis
      ↓
Boundary as an Execution-Time Primitive [Tsai, 2026f]
      ↓
      Formalizes the human selection act (V', E')
      as a first-class execution-stage primitive
      ↓
Decision Behavior Governance [Tsai, 2026a]
      ↑
      Governs decision formation using VSS artifacts
      as the authorized premise structure

Anchor Architecture [Tsai, 2026e]
      ↕
      Horizontal infrastructure: applies across all
      artifact types — Specification, Code, Test,
      Directive, Evidence
```

VSS occupies the position immediately upstream of all execution-stage mechanisms. It does not prescribe how Boundaries are formed, how anchors are assigned, or how decisions are governed. It provides the structural precondition—a versioned, addressable, multi-viewpoint Specification—without which none of the downstream mechanisms have a stable structural basis to operate on.

8.2 Relation to Anchor Architecture

Anchor Architecture [Tsai, 2026e] is a general-purpose infrastructure for binding semantic commitments to implementation loci. It is not specific to VSS: it applies across all artifact types that participate in AI-assisted development, including Specifications, Code, Tests, Directives, and Evidence.

VSS relates to AA as one of its upstream suppliers. The Traceability property established in Section 6.3 produces the addressable, versioned elements that AA's anchoring operations require as source artifacts. When an element $e \in E$ is selected into a Boundary and used as the governing context for a code generation event, AA provides the mechanism for binding that element's declared Intent to the specific implementation locus it governs.

The relationship is unidirectional and non-exclusive. VSS does not require AA to produce a Structured Specification; the (V, E) structure is complete without anchoring. AA does not require VSS as its only source of traceable artifacts; it operates on any artifact class that satisfies its addressability requirements. The relationship is one of structural compatibility, not mutual dependency.

8.3 Relation to Boundary as an Execution-Time Primitive

Boundary as an Execution-Time Primitive [Tsai, 2026f] formalizes the human selection act—the formation of (V', E') from a versioned Specification—as a first-class structural primitive at execution time.

VSS provides the preconditions for this primitive to be meaningful. The selectable structure (V, E) and the version basis that determines which elements and Viewpoints are available for selection both originate in VSS. Without a Viewpoint-Structured Specification, there is no (V, E) to select from, and the Boundary primitive has no structural material to operate on.

The relationship is again unidirectional. The Boundary primitive consumes what VSS produces; VSS does not depend on the Boundary primitive to produce it. The Specification is complete—versioned, addressable, conflict-detectable—regardless of whether any Boundary has yet been formed from it. The formation of Boundaries is an execution-stage activity; the production of the Specification is a pre-coding activity. VSS governs the latter; the Boundary primitive governs the former.

8.4 Relation to Decision Behavior Governance

Decision Behavior Governance [Tsai, 2026a] addresses the conditions under which AI-assisted decisions can be said to have been formed within an attributable, institutionally authorized structure. It targets the Decision Behavior stage of the development lifecycle—the stage at which Boundaries have been formed and code generation is about to occur.

VSS is the upstream input to DBG's premise structure. The Viewpoints, elements, and version state that constitute a VSS Specification provide the declared basis from which DBG's decision premises are drawn. A decision governed by DBG is a decision whose premises originated in a declared, versioned, multi-viewpoint Specification. Without VSS, the premise structure that DBG requires would itself be unstructured—expressed as prompts or informal documentation, with no persistent, addressable basis for attribution.

The relationship is one of structural layering. VSS operates before decision formation; DBG operates at decision formation. The outputs of VSS are the inputs to DBG's premise definition. VSS does not determine how DBG governs decisions; DBG does not determine how VSS structures Intent. Each framework is complete within its own stage of the lifecycle.

8.5 Relation to Inference Creep

From Inference Creep to Risk Acceleration Pipelines [Tsai, 2026b] defines Inference Creep as the condition in which AI systems expand the scope of changes beyond explicit instruction boundaries based on inferred responsibility, without explicit human authorization. It identifies the absence of declared Intent boundaries as the structural cause of this phenomenon.

VSS addresses this cause at its source. The Multi-Viewpoint structure of (V, E) declares the boundaries of Intent explicitly, across multiple governing perspectives, before any code generation occurs. As established in Section 7.2, the cross-version accumulation of (V, E) in Boundary formation further narrows the space available for unauthorized inference: the richer the declared context, the less structural room remains for AI systems to fill with inferred scope.

VSS does not prevent inference—inference is a property of generative AI systems that cannot be eliminated. It provides the structural conditions under which inference-induced boundary crossings become observable, because the boundaries that have been crossed were declared before the crossing occurred.

8.6 Relation to Ghost Intent

Ghost Intent [Tsai, 2026g] defines the condition in which executable artifacts are produced whose originating decisions cannot be recovered. It identifies the absence of Persistent Addressability and Viewpoint Membership as the structural preconditions for this failure mode: when elements are not anchored to a stable identifier and a declared governing perspective, the chain of intent that authorized their production is irrecoverable.

VSS addresses both preconditions directly. The sufficiency conditions of Section 4.2 require every element $e \in E$ to satisfy Persistent Addressability and Viewpoint Membership by construction. An artifact produced within a Boundary drawn from a VSS Specification is attributable to the specific elements and Viewpoints that authorized its generation, at the specific Specification Version that was current at the time of the Boundary decision. The conditions for Ghost Intent are structurally excluded, not merely discouraged.

8.7 Relation to the Taxonomy of Degradation Phenomena

A companion work currently in preparation proposes a three-layer taxonomy of degradation phenomena in AI-generated code, organized across Intent, Structural, and Behavioral layers. The taxonomy identifies fourteen phenomena and traces their propagation paths across layers.

VSS addresses the upstream conditions for the Intent-layer phenomena in this taxonomy—specifically Ghost Intent, Inference Creep, and Intent Fragmentation—by providing the structural decomposition and explicit declaration that these phenomena presuppose to be absent. The Visibility Inversion Principle articulated in the taxonomy provides theoretical grounding for VSS's pre-coding intervention logic: Intent-layer failures are least costly to address at specification time, and most costly to address after they have propagated into Structural and Behavioral layers.

This companion work is in preparation and is not listed in the References. Its relationship to VSS is noted here to situate the present contribution within the broader research program of which it forms a part.

9. Discussion

The preceding sections have established VSS as a formal framework for structuring Intent into a versioned, addressable, multi-viewpoint Specification prior to AI-assisted code generation. This section examines three questions that the framework raises but does not resolve internally: the conditions under which VSS application is warranted, the relationship between VSS and the software engineering methods that preceded it, and the pathway through which the taxonomy may be extended to address future structural failures not yet anticipated.

9.1 Scope and Limitations

Applicability

VSS is most warranted in systems with long lifecycles, continuous evolution, and multiple code generation events that must be coordinated against a shared governing Intent. In these contexts, the structural properties of (V, E)—Persistent Addressability, versioning, cross-version Boundary selection, and Traceability—provide compounding value: each generation event adds to an accumulated declaration history that constrains subsequent events and narrows the space available for Inference Creep.

For short-duration or single-purpose systems—prototypes, throwaway scripts, isolated automation tasks—the full VSS structure may not be warranted. The cost of declaring and maintaining a multi-viewpoint Specification may exceed the governance benefit in contexts where the system will not evolve beyond its initial generation. In such cases, a reduced Specification—one or two Viewpoints, a small E, no cross-version accumulation—may be sufficient. The risk-driven principle of Section 3.4 applies: governance density should be proportional to the consequences of leaving a concern ungoverned.

Quality Dependencies

The effectiveness of VSS depends on two quality conditions that the framework defines but cannot enforce.

First, the quality of multi-role AI assignment is bounded by the clarity of Viewpoint definitions. A Viewpoint that is poorly scoped—too broad to represent a specific concern, or too narrow to capture the full range of constraints it is intended to govern—will produce elements $e \in E$ whose Explicit Scope does not accurately represent the governing concern. Semantic Conflict Detection will then operate on an incomplete representation of the conflict space, and conflicts that exist in the Intent may remain invisible despite the structural mechanism being present. The precision with which Viewpoints are defined is therefore a direct determinant of the framework's conflict-surfacing capability.

Second, the value of Boundary decisions is bounded by the rigor of Specification Version management. If versions are incremented arbitrarily—without disciplined tracking of which elements changed, which Relations were affected, and which prior Boundaries remain valid under the new version—the structural basis for Boundary decisions degrades. A Boundary formed against a poorly managed version may authorize generation under an Intent state that does not accurately reflect the current system. Version management is not a technical problem; it is a discipline problem. VSS provides the structure for rigorous version management; it cannot substitute for the institutional commitment required to practice it.

9.2 Relation to Established RE and SE Methods

VSS is not proposed as a replacement for established requirements engineering or software engineering methods. It is proposed as a structural interface—a formalization layer that enables existing methods to function as machine-selectable, version-stable governing constraints in AI-assisted development contexts where they currently have no such interface.

The purpose of this comparison is not to position VSS as a replacement for existing methods, but to clarify the structural layer at which VSS operates.

IEEE 830 Software Requirements Specification

The SRS as defined in IEEE 830 is an instance of the Semantic dimension in the VSS taxonomy: it governs functional and non-functional requirements, and it corresponds directly to the SRS Specification type defined in Section 5.1. VSS does not replace IEEE 830; it provides the structural conditions under which an SRS can be decomposed into addressable elements, assigned Persistent Addressability and Explicit Scope, and incorporated into a multi-viewpoint (V, E) structure. The SRS as a document class remains valid; its contents become VSS-compatible when unitized according to the sufficiency conditions of Section 4.2.

Viewpoint-Based Modelling (Finkelstein et al., 1992)

The viewpoint concept in VSS shares a lineage with viewpoint-based modelling in requirements engineering, where viewpoints represent the perspectives of different stakeholders on a common system. The structural role

of Viewpoints in VSS is, however, distinct. In viewpoint-based modelling, viewpoints are primarily a mechanism for managing stakeholder disagreement through negotiation and consistency checking across perspectives. In VSS, Viewpoints are the governing concern boundaries assumed by AI roles prior to code generation. The deliberative dynamic in VSS is not stakeholder negotiation—it is structured multi-role constraint application, in which the tension between AI-assumed perspectives is the mechanism for surfacing latent conflicts in Intent before any human execution decision is made. The two frameworks address different problems at different lifecycle stages and are not in competition.

Model-Driven Development

Model-Driven Development and its associated frameworks treat formal models as the primary artifact from which executable artifacts are derived, typically through automated transformation. VSS addresses a different problem at a different structural layer. VSS governs the formation of the Specification that precedes generation—the declaration of governing Intent before the generation event occurs. It does not prescribe how code is generated from a Specification, nor does it require a formal model as an intermediate artifact. The relationship between VSS and MDD is one of complementarity rather than overlap: MDD addresses the transformation from specification to implementation; VSS addresses the conditions under which the specification that drives that transformation was legitimately formed.

9.3 Extension Pathway for Process-Oriented Frameworks

The taxonomy presented in Section 5 was derived from observed structural failures in AI-assisted development practice, resolved by identifying existing software engineering perspectives capable of addressing each failure. This derivation method is not exhausted by the six Specification types currently defined. As AI-assisted development practice evolves and new classes of structural failure are identified, the appropriate response is to examine the existing body of engineering and process knowledge for perspectives that address the new failure mode, and to formalize the relevant perspective as a Viewpoint within the VSS structure.

Process-oriented frameworks represent one class of candidate sources for future extension. Frameworks such as CMMI (Capability Maturity Model Integration), BPMN (Business Process Model and Notation), and CBPC (Component-Based Process Composition) encode accumulated knowledge about process structure, workflow organization, and capability maturity that may become relevant as AI-assisted development encounters failure modes at the process coordination layer—failure modes not yet adequately represented by the current Semantic and Engineering dimensions.

Two conditions must be satisfied before a process-oriented perspective can be formalized as a VSS Viewpoint. First, the perspective being extracted must represent a governing concern over Intent—a constraint on what the system is or how it is permitted to be built—rather than a process stage, organizational maturity level, or workflow sequencing rule. Process stages and maturity levels are not Viewpoints in the VSS sense; they describe how development is organized, not what Intent has been declared. Second, the extracted perspective must satisfy the sufficiency conditions of Section 4.2: every element it contributes to E must be persistently addressable, explicitly scoped, and attributable to the Viewpoint from which it was produced.

Subject to these conditions, the extension pathway is open. VSS provides a formal structure that accommodates any Viewpoint whose governing concern can be stated as a semantic boundary—regardless of the framework from which that concern was derived. The taxonomy is a starting point, not a boundary.

10. Conclusion

This paper has argued that the structural failures characteristic of AI-assisted software development—Ghost Intent, Inference Creep, Intent Fragmentation, and the absence of a recoverable specification baseline—share a common upstream cause: Intent is expressed through artifacts that lack decomposition, persistence, and viewpoint diversity. No amount of prompt refinement, post-generation review, or runtime monitoring addresses this cause, because all three operate after the structural deficiency has already shaped the generation event. VSS addresses it before.

Three Claims, Restated

Claim 1: Multi-Viewpoint structuring makes pre-coding conflicts structurally observable.

When an AI assumes distinct roles—each representing a different governing concern as an explicitly declared Viewpoint—the deliberative tension between those roles brings latent conflicts in Intent into structural view before any code is written. This is not a descriptive capability. It is a pre-coding conflict surfacing mechanism whose operation depends entirely on the structural separation of concerns into distinct, declared Viewpoints. A single-viewpoint specification cannot produce this effect regardless of how thoroughly it is written. Multi-Viewpoint structuring is the necessary condition.

The deliberative tension between AI roles is not a defect to be eliminated through better prompt design—it is the mechanism through which conflict visibility is produced. This reframes a commonly held assumption: that a sufficiently precise prompt should resolve all ambiguity before generation begins. VSS does not seek to eliminate tension; it seeks to make tension structurally observable. The conflicts that surface under multi-role constraint are not artifacts of imprecision—they are evidence that genuinely incompatible concerns were present in the Intent, and that the specification process has done its work.

Claim 2: Unitization is the foundation of structuring; the four emergent properties are its synergistic effects.

The formal definition $\mathrm{Specification} = (V, E)$, together with the three sufficiency conditions for unitization—Persistent Addressability, Explicit Scope, and Viewpoint Membership—establishes the structural preconditions from which four capabilities emerge without additional mechanism:

$$\text{Unitization} \Rightarrow \begin{cases} \text{Relation} \\ \text{Semantic Conflict Detection} \\ \text{Traceability} \\ \text{Boundary Formation} \end{cases}$$

None of these properties can be engineered into an unstructured specification after the fact. All of them arise naturally once the conditions of (V, E) are satisfied. They are not features of VSS—they are consequences of structuring itself. This causal chain establishes VSS not as a catalogue of useful capabilities, but as a logical system: given unitization, the four properties follow necessarily. The framework is not justified by the utility of its outputs—it is justified by the structural relationship between its foundation and its consequences.

Claim 3: Specification is a permanent artifact; its version is the structural basis for Boundary decisions and the engineering fact foundation for Development-stage accountability.

A prompt is consumed at the moment of generation and leaves no persistent structure. A Viewpoint-Structured Specification persists across the full lifecycle of a system: it accumulates across versions, remains selectable across generation events, and constitutes an engineering fact—a structural condition present before execution—rather than an administrative claim asserted after it. The cross-version accumulation of (V, E) enables Boundary decisions to draw on the full declaration history of the system's Intent, narrowing the space for Inference Creep and ensuring that each generation event is attributable to a specific, declared state of governing Intent.

This distinction—between a transient instruction and a permanent engineering fact—is not a difference of degree. It is a difference of ontological status. A prompt exists only in the moment of its consumption; it cannot be the basis of accountability for decisions that occur after it disappears. A Viewpoint-Structured Specification exists prior to execution and persists after it; it is the structural condition that makes Development-stage accountability possible. No alternative artifact class currently used to communicate intent to AI systems satisfies this condition. The irreplaceability of VSS as the foundation for Development-stage accountability follows not from its features, but from the structural requirement that accountability presupposes: that the basis for a decision must have existed before the decision was made.

The Structural Contribution of VSS

VSS does not propose a new notation, a new process, or a new tool. It proposes a structural precondition: that Intent must be decomposed into a versioned, multi-viewpoint (V, E) structure before code generation begins, and that this structure must persist as a permanent, addressable artifact across the lifecycle of the system it governs.

This precondition is not currently satisfied in prompt-driven AI-assisted development. It is partially satisfied in traditional software engineering practice, but the existing artifact classes—SDD, informal design documents, UML diagrams—were not designed for machine-selectable, version-stable Boundary formation. VSS provides the formalization that bridges this gap: it takes perspectives already present within established software

engineering theory and expresses them as structural conditions that AI-assisted development can operationalize.

The result is not a more sophisticated way to generate code. It is a structural basis for answering a question that prompt-driven development cannot answer: on what declared basis was this code generated, and was that basis present before the generation occurred?

> When Intent remains unstructured, governance cannot begin. VSS does not govern code—it governs the conditions under which code may legitimately be generated.

11. Related Work

This section situates VSS within the existing literature across two bodies of work: the established software engineering and requirements engineering methods from which VSS draws its conceptual foundation, and the companion papers in the present research series that address adjacent structural failures in AI-assisted development.

External Literature

Viewpoint-Based Specification

The use of multiple perspectives as an organizing principle for requirements and specification has a substantial history in software engineering. Finkelstein et al. (1992) introduced a formal framework for integrating multiple stakeholder viewpoints in system development, treating viewpoints as first-class objects with their own specification languages, style rules, and consistency conditions. VSS draws on this lineage in its treatment of Viewpoints as explicitly declared semantic boundaries, but departs from the stakeholder negotiation model in a fundamental way: in VSS, Viewpoints are assumed by AI roles rather than attributed to human stakeholders, and the deliberative tension between roles is not a coordination problem to be resolved through negotiation but a structural mechanism for surfacing latent conflicts in Intent before generation begins.

Software Requirements Specification

IEEE Std 830-1998 defines a recommended practice for the structure and content of Software Requirements Specifications, establishing SRS as the canonical artifact class for expressing functional and non-functional requirements. VSS treats SRS as one instantiation of the Semantic dimension—a Specification type that governs what the system must do. The IEEE 830 standard does not prescribe unitization in the sense required by VSS: it does not mandate persistent element identifiers, explicit per-element scope declarations, or Viewpoint membership. The contribution of VSS is not to replace IEEE 830 but to provide the structural conditions under which an SRS becomes machine-selectable, version-stable, and compatible with multi-viewpoint Boundary formation.

Model-Driven Development

The Object Management Group's Model Driven Architecture (MDA) guide (2003) establishes the principles of model-driven development, in which formal models serve as primary artifacts from which executable implementations are derived through transformation. MDA addresses the transformation stage—how a specification becomes an implementation. VSS addresses the prior stage—the structural conditions under which the specification that drives transformation was legitimately formed. The two approaches are complementary: a VSS Specification could in principle serve as the upstream intent structure from which an MDA-compatible model is derived, though this integration is outside the scope of the present paper.

Structured Analysis and Design

DeMarco's foundational work on structured analysis (1979) established the discipline of decomposing system requirements into explicit, structured representations—data flow diagrams, process specifications, and data dictionaries—as an alternative to informal narrative documentation. This tradition is a direct antecedent of VSS's commitment to decomposition and explicit scope declaration as the basis for governable specification. The key difference is the production mode: structured analysis was designed for human authorship within a sequential development process; VSS is designed for AI multi-role production within a non-linear, generation-driven development context where persistence and machine-selectability are the primary structural requirements.

AI Code Quality and Inefficiency Taxonomies

Abbassi et al. (2025) examine the taxonomy of inefficiencies in LLM-generated code, providing empirical grounding for the class of structural failures that VSS addresses at the specification level. Their work characterizes the code-level manifestations of failures whose upstream cause is the absence of structured, persistent Intent declaration. VSS operates at the layer upstream of these manifestations—not by detecting or correcting code-level inefficiencies, but by establishing the structural conditions that reduce their likelihood by making Intent explicit before generation occurs. *(Full citation to be confirmed upon publication.)*

Companion Papers in the Present Series

The following works form the research series of which VSS is the upstream specification component. Each addresses a distinct stage or structural dimension of the broader problem of AI-assisted development governance.

Tsai, S. (2026a). Toward Decision Behavior Governance:Governance Existence, Invocation, and Decision Formation. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6105226

Establishes the conceptual foundation for Development-stage governance, defining the conditions under which governance can be said to exist, to have been invoked, and to have produced a legitimate decision. VSS provides the upstream specification structure that DBG's premise definition requires.

Tsai, S. (2026b). From Inference Creep to Risk Acceleration Pipelines: Decision Vacancy and Governance Risks in AI-Assisted DevOps. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146686

Defines Inference Creep as scope expansion beyond explicitly declared boundaries, identifies Decision Vacancy as its structural cause, and traces the risk acceleration dynamics that result when ungoverned inference compounds across development stages. VSS addresses the upstream cause by requiring that Intent boundaries be declared before generation begins.

Tsai, S. (2026c). Four Stages of AI Governance: A Stage-Based Framing of Contemporary Practices. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146666

Proposes a four-stage taxonomy of AI governance maturity, situating current practices within a developmental frame and identifying the structural conditions that distinguish reactive from proactive governance. VSS corresponds to the pre-generation stage in this taxonomy.

Tsai, S. (2026d). Runtime Governance vs. Development Governance: Why Runtime Interception Is Not Decision Behavior Governance. Zenodo. https://doi.org/10.5281/zenodo.18876913

Argues that runtime interception mechanisms—guardrails, output filters, and post-generation review—address consequences rather than causes, and establishes the structural distinction between Development-stage and Runtime-stage governance. VSS operates at the Development stage as defined in this paper.

Tsai, S. (2026e). Anchor Architecture. engrXiv / Zenodo. https://doi.org/10.5281/zenodo.18856781

Proposes a horizontal infrastructure for binding semantic commitments to implementation loci across all artifact types in AI-assisted development. VSS produces the addressable, versioned elements that AA's anchoring operations consume as source artifacts.

Tsai, S. (2026f). Boundary as an Execution-Time Primitive. engrXiv / Zenodo. https://doi.org/10.5281/zenodo.18883242

Formalizes the human selection act—forming (V', E') from a versioned Specification—as a first-class structural primitive at execution time. The selectable (V, E) structure and version basis that this primitive requires are produced by VSS.

Tsai, S. (2026g). Ghost Intent: An Effect of Traceability Collapse in GenAI-Assisted SDLCs. Zenodo. https://doi.org/10.5281/zenodo.18872540 *(under review at SSRN)*

Defines Ghost Intent as the condition in which executable artifacts are produced whose originating decisions cannot be recovered. VSS structurally excludes the preconditions for Ghost Intent by requiring Persistent Addressability and Viewpoint Membership for every element in E.

Tsai, S. (2026h). A Taxonomy of Degradation Phenomena in AI-Generated Code. *Companion work in preparation.*

Proposes a three-layer taxonomy of degradation phenomena—organized across Intent, Structural, and Behavioral layers—and traces their propagation paths. VSS addresses the Intent-layer phenomena in this taxonomy, including Ghost Intent, Inference Creep, and Intent Fragmentation, by providing the structural decomposition and explicit declaration that these phenomena presuppose to be absent.

References

External Literature

DeMarco, T. (1979). *Structured Analysis and System Specification*. Prentice-Hall.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke, M. (1992). Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1), 31–57. https://doi.org/10.1142/S0218194092000038

IEEE Computer Society. (1998). *IEEE Recommended Practice for Software Requirements Specifications* (IEEE Std 830-1998). https://doi.org/10.1109/IEEESTD.1998.88286

Object Management Group. (2003). *MDA Guide Version 1.0.1*. OMG Document ormsc/03-06-01. https://www.omg.org/mda/

Object Management Group. (2014). *Model Driven Architecture (MDA) Guide rev. 2.0*. OMG Document ormsc/14-06-01. https://www.omg.org/cgi-bin/doc?ormsc/14-06-01

Companion Papers in the Present Research Series (Self-References)

Tsai, S. (2026a). Toward Decision Behavior Governance: Governance Existence, Invocation, and Decision Formation. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6105226

Tsai, S. (2026b). From Inference Creep to Risk Acceleration Pipelines: Decision Vacancy and Governance Risks in AI-Assisted DevOps. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146686

Tsai, S. (2026c). Four Stages of AI Governance: A Stage-Based Framing of Contemporary Practices. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146666

Tsai, S. (2026d). Runtime Governance vs. Development Governance: Why Runtime Interception Is Not Decision Behavior Governance. Zenodo. https://doi.org/10.5281/zenodo.18876913

Tsai, S. (2026e). Anchor Architecture. engrXiv / Zenodo. https://doi.org/10.5281/zenodo.18856781

Tsai, S. (2026f). Boundary as an Execution-Time Primitive. engrXiv / Zenodo. https://doi.org/10.5281/zenodo.18883242

Tsai, S. (2026g). Ghost Intent: An Effect of Traceability Collapse in GenAI-Assisted SDLCs. Zenodo. https://doi.org/10.5281/zenodo.18872540

Tsai, S. (2026h). A Taxonomy of Degradation Phenomena in AI-Generated Code. *Companion work in preparation.*