

Rethinking Software Engineering Foundations in the Large Language Model Era

Huiwen Han

Enterprise Architecture, Lenovo Group, Beijing, China
hanhuiwen@gmail.com

Abstract

Software engineering theory has been shaped for five decades by three implicit assumptions: that human cognitive labor is the binding constraint on software production, that code is the primary engineering artifact, and that program execution can be treated as a deterministic mapping from inputs to outputs. These assumptions have rarely been made explicit because they have rarely been challenged.

The integration of large language models (LLMs) into software development introduces a different constraint structure. As code generation becomes inexpensive, the binding constraint shifts toward the precision of specification. As natural language specifications become the causal input to production, documentation assumes a central engineering role. And as the production process becomes probabilistic, classical deterministic execution models no longer fully characterize how software artifacts are produced.

We argue that these shifts are consistent with a broader reorientation of software engineering toward constraint-centric methods. To structure this transition, we introduce the Constraint Engineering Framework (CEF), a four-layer specification system grounded in the distinction between knowledge and control. We examine its implications for software lifecycle activities and design principles, and outline a research program that connects this perspective to ongoing empirical and formal work.

Keywords: *Software engineering methodology, large language models, paradigm shift, constraint engineering, probabilistic execution, Agile, software lifecycle*

1. Introduction

Software engineering was born in crisis. The 1968 NATO conference identified what became known as the "software crisis": the difficulty of reliably producing large software systems on time, within budget, and at acceptable quality. The theoretical response — developed over the following decades — produced a coherent architecture of lifecycle models, process frameworks, design principles, and formal methods. These advances were implicitly calibrated to a common constraint: the scarcity of human cognitive labor applied to code production.

Large language models alter this constraint structure. Systems such as GPT-4 and Claude can generate syntactically and semantically coherent code from natural language specifications across a wide range of tasks [1, 2]. Empirical studies and industrial deployments report substantial productivity gains for routine development work [3]. Multi-agent systems such as SWE-agent and Devin resolve software engineering tasks autonomously from natural language issue descriptions [4, 5]. As a result, the marginal cost of code production is reduced, in some cases dramatically.

When code generation becomes inexpensive, the central challenge of software engineering shifts — from producing code to specifying and constraining what should be produced. This shift propagates in three directions: upstream to the precision of specification, downstream to the verification of intent, and inward to the semantics of the generation process itself, which is probabilistic and context-dependent [6, 7].

When code generation becomes inexpensive, the central challenge of software engineering shifts from producing code to specifying and constraining what should be produced.

We argue that these simultaneous changes may signal a reorientation in the foundations of software engineering, rather than a simple extension of existing frameworks [27]. This raises a structural question: can existing software engineering frameworks be extended to accommodate these changes, or do they rely on assumptions that no longer hold simultaneously? The following sections argue that the challenge arises not from any single dimension, but from the interaction of all three. Section 2 identifies the three implicit assumptions. Section 3 introduces the theoretical foundations. Section 4 examines the Agile Manifesto under the new framework. Section 5 presents the Constraint Engineering Framework. Section 6 examines lifecycle and design principle implications. Section 7 outlines the research program and concludes.

Running Example

Throughout this paper, we use a single running example to ground the analysis: an enterprise software team building an authenticated API service using LLM-based code generation. The team is producing a Python web service in which an LLM generates endpoint implementations from natural language specifications, with human review before deployment.

The example is deliberately mundane — authenticated database access is a routine task — because the argument applies broadly, not only to frontier cases.

Contributions

This paper makes three contributions:

- **Explicit identification of three implicit assumptions** that underlie all classical SE theory, and analysis of how LLM integration simultaneously undermines their sufficiency.
- **The Constraint Engineering Framework** — a four-layer specification system grounded in the logical distinction between knowledge and control — as the organizing construct of a new theoretical architecture.
- **A systematic analysis of implications** for the software lifecycle, iteration methodology, and classical design principles, together with a research agenda linking paradigm claim to five companion empirical and formal studies.

2. Three Foundational Assumptions of Classical SE

Classical software engineering rests on three assumptions that have operated as invisible axioms — never stated because never challenged. Making them explicit is the first step toward understanding what LLMs change.

A1: Human Cognitive Labor is the Binding Constraint

Every major SE framework is calibrated to a world in which software is produced by humans, line by line, at cognitive cost. This assumption motivates Agile's focus on reducing coordination overhead, modularity's focus on managing comprehension load, DevOps's automation of repetitive tasks, and COCOMO's cost models centered on developer effort [8, 9, 10].

The empirical grounding is well-established. Boehm's COCOMO model attributes the dominant fraction of development cost to human labor [8]. Dybå and Dingsøyr's systematic review of Agile literature identifies human coordination cost as the primary variable that Agile methods address [11].

A1 is the load-bearing assumption of classical SE theory. When it holds, the entire edifice makes sense. When LLMs reduce the marginal cost of code production toward zero for a broad class of tasks, the optimizations built on A1 become misaligned with the actual constraint structure.

A2: Code is the Primary Engineering Artifact

Classical SE treats code as the central artifact: requirements documents, design specifications, and architecture diagrams are inputs to or outputs from the coding process. The Agile Manifesto makes this explicit: "working software over comprehensive documentation" [12]. Documentation is overhead — necessary to a point, but never the point.

This prioritization is rational when A1 holds: if human cognitive labor is scarce, investing it in documentation rather than code is wasteful. While specifications have always existed, they were historically secondary — records of decisions made, not causal inputs to production.

A2 requires reexamination when the primary production input is natural language specification. When an LLM generates code from a specification, the specification is not documentation of code already written — it is the input that causes code to be generated. Its quality directly determines output quality. The artifact hierarchy inverts: specification becomes cause; code becomes effect.

A3: Execution is Deterministic

Classical formal methods — denotational semantics [13], operational semantics [14], axiomatic semantics [15] — model programs as deterministic functions. A program in a given environment produces a specific output. This enables formal verification, deterministic testing, and exact debugging.

A clarification is essential: code generated by LLMs, once generated, executes deterministically. A3 is undermined not at the runtime layer but at the generation layer — the layer at which the primary software artifact is produced. Classical SE implicitly assumes that artifact production is also deterministic: a developer given the same specification produces functionally equivalent code. This assumption underlies deterministic testing (reproducing failures), version-based configuration management, and code review (reconstructing the author's intent). When the artifact production engine is stochastic, these tools address only the artifact — they do not characterize the generation process that produced it [6].

Simultaneous Undermining: Why Extension Fails

In our running example, all three assumptions fail together. The team specifies "implement authenticated database access" to an LLM: A1 fails (binding constraint is specification clarity, not developer time); A2 fails (the specification, not generated code, is the primary artifact); A3 fails (the same specification produces different implementations, requiring distributional rather than single-shot testing).

The critical observation is that all three assumptions are simultaneously challenged. Existing frameworks optimize under A1, A2, and A3 jointly; fixing one while leaving the others intact produces incoherence. Extending Agile to accommodate probabilistic generation while retaining code as the primary artifact yields a framework whose measurement and quality tools are not designed for probabilistic artifacts.

The framework most needed — organized around specification precision, semantic constraint, and distributional quality assurance — does not exist in any current methodology. Recent empirical work documents LLM failure modes that existing SE frameworks lack vocabulary to classify [16, 17], suggesting the limitation is structural rather than localized to any specific method. A different organizing perspective may be required [18, 27].

	Assumption	What Classical SE Builds On It	How LLM Integration Undermines It
A1	Human cognitive labor is the binding constraint on software production	Agile (reduce coordination cost), modularity (reduce comprehension load), DevOps (automate repetition), COCOMO cost models, code review value	When LLMs reduce marginal code-generation cost toward zero, the constraint migrates upstream (to specification precision) and downstream (to intent verification). Optimizing human labor allocation becomes secondary to specifying and verifying intent.
A2	Code is the primary engineering artifact	Working software over documentation (Agile); progress measured in commits and coverage; expertise valued as programming skill	When an LLM generates code from a specification, the specification is not documentation — it is the causal input to generation. Specification quality directly determines output quality. The artifact hierarchy inverts.
A3	Execution is deterministic	Formal verification, deterministic testing, version-based configuration management, exact debugging	LLM artifact production is stochastic: the same specification produces a distribution over outputs, not a single output. Classical correctness (binary pass/fail) gives way to distributional correctness (probability mass on acceptable outputs above threshold θ).

Table 1. Three foundational assumptions of classical SE: what each assumption grounds, and how LLM integration undermines its sufficiency. (Epistemic status: analytical argument.)

3. Theoretical Foundations

The framework proposed in this paper draws on four published theoretical results. We describe each briefly, emphasizing the specific insights applied in subsequent sections.

Intent-Conditioned Computation [19]

Intent-Conditioned Computation (IIC) replaces the classical program/data separation with a model in which behavior emerges from model capabilities, control intentions, externalized knowledge, and enforcement mechanisms. Three IIC results underpin the framework here: behavioral plasticity (directing the model differently produces qualitatively different behavior without code changes), context dependence (identical weights may produce different outputs depending on context), and probabilistic execution (behavior is a distribution, not a deterministic function). IIC also reframes governance as intent traceability: reconstructing the intent-context-outcome chain for any system execution.

Probabilistic Program Execution Semantics [6]

Probabilistic Execution Semantics (PPES) formalizes LLM generation as a stochastic process governed by prompt, hidden execution history, and observable outputs, introducing a third execution phase — infer-time — distinct from compile-time and runtime. Two PPES results are applied here: a behavioral equivalence criterion (two specifications are equivalent when their induced output distributions are sufficiently close) and bounded sensitivity (small model parameter changes produce bounded distribution shifts, bounding recovery time after foundation model updates).

The DIKCA Logical Architecture [20]

DIKCA extends the classical DIKW (Data–Information–Knowledge–Wisdom) hierarchy by identifying two additional layers — Control and Autonomy — that DIKW conflates with Knowledge. The central insight: knowledge and control belong to different logical types.

Knowledge can influence behavior, but only control can guarantee it. A representation may encode a norm without exercising normative force — enforcement requires implementation-level constraint, not semantic description alone. [20]

This distinction has immediate engineering consequences. Writing "please avoid SQL injection vulnerabilities" in a context prompt is a K-layer operation: it provides information that may shift the output distribution toward secure patterns. Enforcing a CI/CD pipeline check that rejects generated code containing string concatenation in database queries is a C-layer operation: it exercises normative force regardless of what the model generates. The K-layer operation is necessary but not sufficient for security. The C-layer operation is the enforcement mechanism.

4. The Agile Manifesto Reinterpreted

The Agile Manifesto [12], published in 2001, articulates four value statements that remain the dominant organizing framework for software development methodology. Each was a rational optimization under A1–A3. Under LLM-integrated conditions, each requires systematic reinterpretation — not abandonment, but reconceptualization of what the principle means when the primary production agent is a probabilistic foundation model.

We note at the outset that these reinterpretations apply specifically to LLM-integrated workflows. Classical Agile principles retain their validity in human-only or minimally LLM-assisted development contexts.

Agile Principle	Classical Rationale	LLM-Era Reinterpretation
Working software > Documentation	Docs are costly overhead when human labor is scarce; code is the artifact that matters	Documentation IS the primary production input. When LLMs generate code from specifications, specification quality directly determines output quality. Comprehensive specification is the precondition for working software, not its alternative.
Individuals > Processes and Tools	Processes impede human collaboration; face-to-face communication resolves ambiguity faster	An LLM has no persistent memory, no organizational alignment, no regulatory commitment. Process provides behavioral governance — C-layer constraints — not human coordination overhead. The Agile critique applies to human-coordination processes; it does not apply to AI behavioral governance.
Customer collaboration > Contract	Requirements are discovered progressively; fixed contracts cannot anticipate needed changes	Specification precision, not progressive delivery, is the binding constraint. Ambiguity in a specification is materialized immediately in generated code. Collaborative focus shifts from delivery iteration to precision of intent.
Responding to change > Following a plan	Plans cannot anticipate requirements volatility; rigidity is costly	Undisciplined changes to the constraint system destabilize the output distribution. Constraint system instability (not plan rigidity) is the new risk. Change management must govern CEF updates, not just feature scope.

Table 2. Agile Manifesto principles: classical rationale and LLM-era reinterpretation. Each transformation is conditional on LLM integration depth; Agile is not superseded but reconceptualized. (Epistemic status: analytical judgment.)

The pattern across all four principles is consistent: what changes is not the goal (quality, responsiveness, collaboration, working software) but the mechanism by which each goal is pursued when the primary production agent is a stochastic foundation model rather than a human developer. Agile's insight that methodology should be calibrated to the binding constraint remains correct — the constraint has shifted.

4.5 Why Extension Is Insufficient

Three extension strategies might be proposed, each failing for a distinct structural reason. Extending Agile requires adding AI behavioral governance and distributional testing that are foreign to its human-coordination model. Extending formal methods cannot address the generation level: verification assesses a specific artifact but cannot characterize whether a stochastic generation process produces acceptable outputs. Extending DevOps treats generation as a black box, with no mechanisms for governing specification quality or detecting distributional drift. Each extension addresses the symptom — how to handle generated code — but not the source: the absence of a framework for governing the generation process itself.

5. The Constraint Engineering Framework

If specification becomes the primary input to software production and behavior becomes probabilistic, then the central engineering problem shifts: from constructing artifacts to constraining generative processes. The question is no longer only how to build software, but how to shape the range of behaviors that a generative system can produce. This shift motivates the need for an organizing construct that can express intent, bound behavior, and support iterative refinement under uncertainty. We propose the Constraint Engineering Framework (CEF): a four-layer specification system grounded in the distinction between knowledge and control, operating over a Model Capabilities substrate, with three adaptive feedback loops and Human-in-the-Loop governance.

5.1 The K/C Separation Principle

The central insight of the CEF is that knowledge and control are distinct and should be treated differently in system design. Knowledge can influence behavior by shifting the likelihood of certain outputs. Control constrains behavior by enforcing what outputs are allowed. Conflating the two leads to a common failure mode: systems that appear guided but are not guaranteed to comply. The formal proof that K-layer modifications cannot substitute for C-layer enforcement appears in the companion paper PRISM-P3 [21].

- **K-layer artifacts (Context, Skills)** inform but do not enforce. A Context document describing security best practices increases the probability that generated code follows those practices. It does not guarantee compliance. The output distribution shifts toward secure patterns but retains non-zero mass on insecure outcomes.
- **C-layer artifacts (Rules)** enforce but do not explain. A Rule implemented as a CI gate that rejects generated code containing direct SQL string concatenation exercises normative force regardless of what the model generates. It does not "explain" why SQL injection is bad — it enforces the constraint through deterministic code execution.

- **Conflating K and C produces false security:** An organization that relies on context descriptions without C-layer enforcement has a system that may reduce violation frequency but cannot bound it. This is the most common failure mode in current LLM-integrated development practice [17].

5.2 The Four Layers

The CEF comprises four layers above the MC substrate:

- **Context (K-layer):** the interpretive reference frame. Domain conventions, regulatory environment, team standards. Lowest rate of change. Shifts the output distribution toward domain-appropriate generation; does not enforce it.
- **Skills (K-layer):** organizational knowledge crystallization. Solution path prescription. Team accumulated conventions for how specific task types should be approached. Activates preferred solution paths within MC; does not guarantee compliance.
- **Rules (C-layer):** hard behavioral invariants. Must be enforced through deterministic code (automated checks, CI gates), not expressed as suggestions in Context. A Rule expressed as "please avoid X" is not a Rule — it is a Context entry.
- **Specification (A-layer):** task-level intent + negative scope + acceptance criteria. Defines what is to be generated AND what must not be generated or modified. Negative scope constraints are primary mechanisms for bounding generative scope.

5.3 Running Example: CEF in Practice

Consider a task: implement authenticated database access (GET /users/{id}) in a Python web service.

- **Context:** OWASP Top 10 guidelines; organizational database conventions; existing authentication architecture. Effect: shifts output distribution toward secure, convention-compliant code. Does not guarantee compliance.
- **Skills:** parameterized query template (organization-standard pattern); JWT validation pattern. Effect: activates preferred solution paths; reduces output variance for known task types.
- **Rules:** CI gate rejecting SQL string concatenation; automated test asserting authentication precedes data access; scope assertion verifying no modification outside specified file. Effect: enforces compliance regardless of generation output.
- **Specification:** "Return user profile for {id}. Must not modify authentication middleware. Must not expose internal IDs. Must not create tables." Negative scope constraints bound generative scope, preventing the model from expanding into adjacent functionality.

5.4 Adaptive Feedback and Governance

The CEF is not static. It evolves through three feedback loops operating at different timescales. At the fastest cadence (per CI run), lint violations and test failures trigger Rule updates. At an intermediate cadence (per iteration), consistent semantic failure patterns trigger Skill refinement. At the slowest cadence (monthly or per model update), production monitoring and model version changes trigger Context updates and full Rule re-audit.

Human-in-the-Loop governance is the meta-control mechanism: humans review accumulated feedback signals, judge whether they warrant constraint updates, determine which layer (K or C) the update targets, and authorize changes. These governance responsibilities have no direct classical SE analog — they concern the constraint system itself, not the artifacts it produces.

Each generation episode should produce an intent provenance record (context version, skills, rules

version, specification, model version, output sample), enabling post-hoc audit by reconstructing the specification chain without model interpretability. The formal treatment, including convergence proofs and the K/C Non-Substitutability Theorem, appears in PRISM-P3 [21].

6. Implications: Lifecycle and Design Principles

6.1 Thought-Driven Iteration

Classical software development iterates by expanding functionality in response to external feedback. In LLM-integrated settings, the primary challenge is different: the space of possible solutions is already large, and often under-constrained.

Iteration therefore shifts from adding features to refining intent. Each step narrows the range of acceptable behaviors rather than expanding the set of required ones. The process becomes one of progressively constraining generation, reducing ambiguity, and aligning outputs with intended outcomes. In this view, iteration operates not on code directly, but on the specification that governs its generation. The formal convergence theory for this process, including the Principle of Minimal Generative Scope, appears in the companion paper PRISM-P4 [22].

6.2 The Software Lifecycle

Every phase of the software development lifecycle transforms under the new framework. The transformation follows a consistent pattern: activities centered on human code production shift toward activities centered on intent definition and verification governance.

Lifecycle Phase	Classical Focus	LLM-Era Focus	Expertise Shift
Requirements	Eliciting enough requirements from stakeholders	Expressing intent with sufficient precision to guide generation; defining negative scope (what must not be generated)	Business Analyst → Semantic Arbiter: from document production to constraint design
Design	Structural decomposition: class hierarchies, component boundaries	Constraint architecture: CEF layer design; MC capability assessment; constraint consistency	Architect → Constraint Architect: from structural decomposition to constraint system design
Development	Code composition: translating designs into implementations	Directed generation: specification writing, context window management, intent-implementation alignment judgment	Developer → Director/Evaluator: from code author to generation director
Code Review	Style and defect detection: naming, structure, logic errors	Intent validation: does generated code implement intended behavior? Scope adherence: did generation stay within specified scope?	Reviewer → Intent Validator: from style checking to intent verification

Testing	Functional verification: single-shot assertions	Distributional testing: N repeated generation runs; boundary validation; scope boundary testing	Tester → Boundary Analyst: from deterministic assertion to distributional characterization
Security/ Compliance	Post-hoc audit: scan completed code for violations	Embedded constraint design: encode compliance requirements as C-layer Rules enforced before generation	Auditor → Constraint Designer: from post-hoc detection to pre-generation prevention
Operations	System maintenance: patching, scaling, incident response	AI system governance: MC monitoring (model version changes); output distribution drift detection; intent provenance audit	Operations → AI System Governor: from infrastructure maintenance to behavioral governance

Table 3. Software lifecycle transformation: classical primary activities, LLM-era primary activities, and expertise shifts. (Epistemic status: analytical prediction; empirical validation is the subject of PRISM-P6 [23].)

The transformation predicts systematic effort redistribution: away from the development phase (where LLMs reduce marginal cost) and toward the poles (requirements/specification precision at one end; review, testing, security, and governance at the other). This prediction is empirically testable and constitutes the primary research question of PRISM-P6.

Two lifecycle roles emerge with no classical predecessors: one concerned with managing behavioral changes when the underlying foundation model is updated, and one concerned with maintaining the specification chain for each generation episode to support governance and compliance audit.

6.3 Design Principles

Classical software design principles — encapsulation, SOLID, high cohesion and low coupling — are preserved in intent but shift in mechanism. Encapsulation moves from type-system enforcement to semantic capability hiding via the Skill construct; failure modes shift from compile-time type errors to soft, plausible-but-wrong implementations detectable only through distributional testing. The Dependency Inversion Principle maps directly to the K/C architectural boundary: K-layer specifications express intent; C-layer enforcement is model-independent. The complete design theory, including formal IIC-role mappings for all SOLID principles, appears in PRISM-P5 [24].

7. Research Program and Conclusion

7.1 The Validation Research Program

The paradigm-level claims in this paper generate testable predictions. Five companion studies provide the validation program:

- **PRISM-P2 (IEEE TSE):** Systematic six-dimension comparative analysis of every classical SE construct under the new framework. Falsifiability: fewer than half requiring substantive reinterpretation would revise the claim to partial reorientation.
- **PRISM-P3 (TOSEM):** Controlled experiment comparing six CEF configurations, measuring K/C separation effect on output quality, scope adherence, and consistency. Falsifiability: no measurable improvement over conflated configurations would revise the K/C distinction's architectural primacy.

- **PRISM-P4 (IEEE TSE):** Formal convergence theory for thought-driven iteration with think-aloud empirical validation. Falsifiability: failure to operationalize ϵ -observational equivalence as a convergence criterion would require reformulation.
- **PRISM-P5 (TOSEM):** Design study and practitioner survey validating semantic OOP/SOLID mappings. Falsifiability: no measurable design quality improvement from IIC/DIKCA awareness would revise the mappings' practical utility.
- **PRISM-P6 (ICSE):** 12–18 month longitudinal case study across eight organizations testing lifecycle effort redistribution. Falsifiability: no redistribution toward specification and governance would revise the constraint-migration argument.

7.2 Conclusion

Software engineering has developed a coherent theoretical foundation over five decades, organized around assumptions that have historically held: that human cognitive effort is the primary constraint, that code is the central artifact, and that execution can be treated as deterministic. Large language models challenge the sufficiency of these assumptions when they are taken together.

The argument of this paper is not that existing methods are invalid, but that they are calibrated to a different constraint structure. As that structure changes, the focus of engineering shifts — from constructing code to expressing and constraining intent.

7.3 Scope and Limitations

This reorientation does not apply uniformly. Classical SE methods remain fully effective where implementations are deterministic (embedded systems, safety-critical software), where LLM assistance is limited to code completion with human developers as primary authors, or where regulatory requirements mandate deterministic, traceable, human-authored implementation. The framework applies where LLM-based generation is a central production mechanism — and the distinction scales with integration depth: teams using LLMs for 20% of their code have different constraint needs than teams using them for 80%. Acknowledging this scope boundary strengthens rather than weakens the central argument: the reorientation is not a universal replacement of classical SE, but a framework for a growing class of development contexts in which classical assumptions are no longer jointly sufficient.

The emerging challenge is therefore not only how to build software efficiently, but how to precisely express and reliably enforce what we intend software to do.

The Constraint Engineering Framework provides one way to reason about this shift: a layered system in which knowledge shapes behavior and control enforces it, supported by iterative refinement and human oversight. The PRISM series provides the systematic research program — theoretical, empirical, and formal — to examine, validate, and refine this perspective.

The software crisis of 1968 was a crisis of complexity: how to manage human-authored systems at scale. The challenge of the LLM era is a crisis of intent: how to express, constrain, and verify what we want from systems that generate artifacts faster than we can fully specify them.

Acknowledgments

The PRISM series builds on four published theoretical foundations: IIC [19], PPES [6], the Coalgebraic Framework [7], and DIKCA [20]. The authors thank the reviewers of each foundation paper whose scrutiny strengthened the theoretical basis of this applied series.

References

- [1] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.
- [2] OpenAI, "GPT-4 Technical Report," arXiv:2303.08774, 2023.
- [3] S. Peng et al., "The impact of AI on developer productivity: Evidence from GitHub Copilot," arXiv:2302.06590, 2023.
- [4] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," arXiv:2405.15793, 2024.
- [5] S. Cognition, "Introducing Devin," cognition.ai, 2024.
- [6] H. Han, "Probabilistic Program Execution Semantics for Foundation Models: A Formal Operational Framework," Authorea preprint, 2026. Submitted to Formal Aspects of Computing (FAC). <https://www.authorea.com/users/903147/articles/1384389>
- [7] H. Han, "Probabilistic Execution Beyond Classical Systems: A Context-Dependent Coalgebraic Framework with Canonical Minimal Realisation," Authorea preprint, 2026. Planned for Logical Methods in Computer Science (LMCS). <https://www.authorea.com/users/903147/articles/1391658>
- [8] B. W. Boehm, Software Engineering Economics. Prentice Hall, 1981.
- [9] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," Commun. ACM, vol. 15, no. 12, 1972.
- [10] IEEE CS, Guide to the SWEBOOK, v3.0, 2014.
- [11] T. Dybå and T. Dingsøy, "Empirical studies of agile software development," Inf. Softw. Technol., vol. 50, 2008.
- [12] K. Beck et al., "Manifesto for Agile Software Development," agilemanifesto.org, 2001.
- [13] D. Scott and C. Strachey, "Toward a Mathematical Semantics for Computer Languages," Oxford PRG Monograph, 1971.
- [14] G. D. Plotkin, "A Structural Approach to Operational Semantics," Aarhus University, 1981.
- [15] C. A. R. Hoare, "An axiomatic basis for computer programming," Commun. ACM, vol. 12, no. 10, 1969.
- [16] Y. Liu et al., "Is Your Code Generated by ChatGPT Really Correct?" NeurIPS, 2023.
- [17] H. Khlaaf et al., "A Hazard Analysis Framework for Code Synthesis LLMs," arXiv:2207.14157, 2022.
- [18] T. Kuhn, The Structure of Scientific Revolutions. University of Chicago Press, 1962.
- [19] H. Han, "Intent-Conditioned Computation: A Complementary Framework for Understanding Modern AI," Authorea preprint, 2026. Submitted to AI Magazine. <https://www.authorea.com/users/903147/articles/1399148>
- [20] H. Han, "Intelligence Beyond Knowledge: Control, Autonomy, and the Logical Architecture of Artificial Agency," PhilPapers, 2024.
- [21] H. Han, "The Constraint Engineering Framework as a DIKCA-Grounded Control System," PRISM-P3 (in preparation).
- [22] H. Han, "Convergent Refinement: Formal Theory of Thought-Driven Iteration," PRISM-P4 (in preparation).
- [23] H. Han, "Effort Redistribution Across the LLM-Era Software Lifecycle," PRISM-P6 (in preparation).
- [24] H. Han, "Semantic Object-Orientation: SOLID and OOP for LLM-Integrated Design," PRISM-P5 (in preparation).
- [25] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems," NeurIPS, 2015.
- [26] S. Amershi et al., "Software engineering for machine learning: A case study," ICSE-SEIP, 2019.
- [27] H. Han, "From Machine-Oriented to Agent-Oriented Programming: Foundations of a Semantic Paradigm for the LLM Era," Zenodo preprint, 2026. Submitted to Minds and Machines. <https://zenodo.org/records/19640321>