Beyond Code: A Formal Theory of Software Engineering in AI-Native Paradigm

Core Thesis: AI reorganizes the layer distribution of software evolutionary complexity rather than eliminating it. Software engineering shifts its core paradigm from code production to system knowledge stewardship.

First-Order Invariant: System Knowledge (K) is the only implementation-agnostic, cross-layer persistent invariant of software systems, which dominates the full lifecycle evolution and credibility boundary of software systems.

Theory Positioning: A general formal theory for AI-native software engineering, independent of specific tools, frameworks, vendors and implementation details. All practical systems (including Zelos) are reference implementations, without participation in theoretical deduction.

Abstract

Driven by autonomous code generation and multi-agent production, the traditional human-centric code-production paradigm of software engineering has encountered fundamental paradigm drift. Traditional software engineering theories and empirical rules are tightly coupled with human coding behavior, lacking explanatory power for machine-dominated software development scenarios and failing to answer the core practical question: *how humans conduct valid software engineering when AI becomes the primary software producer*. This paper proposes a formal invariant system for software engineering, takes system knowledge as the unique first-order invariant, constructs a minimal and complete axiom set, and derives a series of scalable laws and boundary theorems. The core paradigm conclusion is clear: AI cannot eliminate software evolutionary complexity but only migrates it from the code implementation layer to the knowledge governance layer; future software engineering realizes a fundamental division of labor of **machine full production of variable implementations and human full stewardship of invariant system knowledge**. This theory explains the paradigm substitution logic of AI software engineering, accommodates classical software engineering scenarios as boundary special cases, and predicts the structural evolution of future software engineering processes, roles and evaluation systems. Different from empirical engineering summaries, this theory strictly separates invariant theoretical rules and variable implementation behaviors, decouples disciplinary essence from AI technological trends, and maintains long-term universal validity.

1. Introduction

1.1 Research Background and Problem Statement

Classical software engineering is established based on an implicit precondition: humans are the sole producer and verifier of software artifacts. With the iterative upgrading of AI code generation and automated engineering systems, software changes can be independently generated, iterated and deployed by machine producers. The traditional engineering mechanism centered on manual coding and manual review gradually loses scalability.

Current AI software engineering research mostly focuses on tool application and process optimization, lacking fundamental theoretical modeling of disciplinary paradigm transformation. It is still unclear what the core invariant of software systems is under AI-native conditions, how engineering complexity migrates and distributes, and what the boundary constraints of machine autonomous engineering are.

1.2 Research Contribution & Roadmap

This paper establishes a novel invariant-centered foundational paradigm for software engineering, re-answering the core disciplinary question unresolved since the 1968 NATO Conference: **What is the essential, persistent object that software engineering fundamentally manages?**. Different from traditional experience-based and tool-coupled software engineering theories, this work decouples disciplinary essence from implementation details and technological iterations, forming a closed-loop formal theoretical system. The core three-fold contribution is refined as: (1) **Ontological Innovation**: Propose a strict, decidable, non-circular ontology of System Knowledge, eliminate conceptual ambiguity, and formally prove its uniqueness as the first-order invariant of software systems; (2) **Boundary Innovation**: Systematically distinguish the differences and connections between the proposed theory and classical software engineering theories, knowledge engineering, and formal method systems, clarifying the exclusive innovative boundary of the invariant paradigm; (3) **Verifiable Innovation**: Construct a set of falsifiable theoretical predictions and feasible validation schemes, enabling the theory to support empirical testing, case verification and follow-up academic expansion. This theory is not limited to AI-native software scenarios, but provides a universal essential interpretation framework for the entire software engineering discipline.

This paper establishes a complete formal theoretical system for software engineering invariant paradigm, with three core contributions: (1) Propose a strict ontological definition of System Knowledge, and prove that it is the only persistent and implementation-agnostic first-order invariant of software systems through attribute verification and alternative object elimination; (2) Construct a minimal, independent and non-redundant axiom set, and form a closed-loop deduction system of definitions-axioms-laws-theorems; (3) Propose the evolutionary complexity migration theorem and engineering infeasibility theorem, clarify the paradigm evolution law and absolute boundary of multi-agent software engineering, unify classical engineering scenarios as theoretical boundary cases, and provide a systematic future research program for subsequent academic exploration.

2. First-Order Invariant: System Knowledge

This section formally defines the ontology of System Knowledge, verifies its three invariant attributes in strict ontological terms, and eliminates alternative candidate objects to prove that System Knowledge is the unique first-order invariant of software engineering, rather than an empirical summary or broad empirical concept.

2.1 Ontological Definition of System Knowledge

To resolve conceptual ambiguity and meet formal ontology requirements, this paper provides a **strict, closed, non-circular definition** of System Knowledge (K), independent of empirical enumeration:**System Knowledge refers to the complete set of stable, constraint-effective, implementation-agnostic abstract rules and state invariants that dominate the functional correctness, boundary safety, structural rationality, and evolutionary sustainability of a software system throughout its lifecycle.**

It consists of five orthogonal, non-overlapping sub-dimensions, covering all abstract constraint entities involved in software systems and eliminating definitional ambiguity:

1. Domain Knowledge: Business invariant rules, functional semantics, and problem-space constraints (core intent of system existence).

2. Architectural Knowledge: System partitioning, dependency topology, coupling/cohesion constraints, and structural invariants.

3. Behavioral Constraint Knowledge: Runtime policy, state transition rules, security compliance protocols, and validity boundaries.

4. Quality Knowledge: Invariant standards for performance, reliability, maintainability, and evolvability.

5. Evolution Knowledge: Iteration logic, change dependency, and system update invariants that govern long-term system iteration.

All concepts previously scattered as independent entities (Architecture, Constraint, Intent, Policy) are **subordinate subsets of System Knowledge**, rather than competing first-order entities. This orthogonal partitioning completely solves the problem of over-broad definition and provides a decidable boundary for subsequent invariant proof.

A qualified software system first-order invariant must satisfy three core attributes:**implementation agnosticism**, **lifecycle persistence**, and **global dominance**. Implementation agnosticism means that the object does not depend on specific languages, frameworks, architectures and deployment modes; lifecycle persistence means that the object will not perish with system iteration, reconstruction and carrier replacement; global dominance means that all system behaviors and state changes are constrained by the object.

2.2 Invariant Verification of System Knowledge

System knowledge includes domain rules, architectural constraints, business logic, security boundaries and quality paradigms of software systems. It satisfies all invariant attributes: (1) Implementation agnosticism: system knowledge can span different programming languages, frameworks and infrastructure environments. System reconstruction, technology stack replacement and code full rewriting will not change the core system knowledge; (2) Lifecycle persistence: code, configuration and executable programs are temporary carrier artifacts, which can be replaced and eliminated with system evolution, while system knowledge continues to dominate system iteration; (3) Global dominance: all legal software changes, evolutionary behaviors and complexity accumulation are constrained and defined by system knowledge.

2.3 Elimination of Candidate Non-Invariant Objects

This paper eliminates four common candidate objects that are regarded as software core bodies, and proves that they do not have invariant attributes:

Software System: As a tangible carrier of knowledge, it has lifecycle limitations and can be reconstructed and destroyed. It is a variable implementation result, not an invariant.

Software Change: A transient state migration behavior of the system, which is a single variable behavior derived from knowledge constraints, without persistence and dominance.

System Evolution: A macroscopic statistical phenomenon formed by continuous superposition of software changes, which is an external manifestation of knowledge iteration, not an essential invariant.

System Complexity: A quantitative derivative attribute formed by incomplete and inconsistent system knowledge, which is a dependent variable of knowledge state, not an independent invariant.

2.4 Conclusion of First-Order Invariant

System knowledge is the only first-order invariant of software systems, which is the essential research and management object of software engineering throughout all technological paradigms. To eliminate academic doubts about over-broad conceptual definition, this section further tightens the ontological boundary, establishes negative definition rules, and forms a fully rigorous decidable ontology system.

2.5 Strict Ontological Boundary & Negative Elimination Rules

Based on the five orthogonal sub-dimensions of System Knowledge defined above, this paper supplements **negative ontological rules** to completely avoid conceptual over-extension and ensure academic preciseness:

Rule 1 (Implementation Exclusion): All executable, deployable, replaceable carrier artifacts (code, configuration, binaries, runtime processes) are not System Knowledge; they are only temporary carriers and state manifestations of knowledge.

Rule 2 (Temporary State Exclusion): All transient system states, single change behaviors, and short-term engineering operations are not System Knowledge; they are derivative behaviors constrained by knowledge invariants.

Rule 3 (Empirical Method Exclusion): Engineering tools, process specifications, and manual operation experiences that change with technological iteration are not System Knowledge; they are variable implementation methods for knowledge stewardship.

Rule 4 (Non-Invariant Constraint Exclusion): Non-essential, adjustable, scenario-specific marginal constraints do not constitute core System Knowledge; only cross-scenario persistent invariant rules belong to the first-order knowledge system.

With positive hierarchical definition + negative boundary elimination, the ontology of System Knowledge achieves **complete decidability**: any software system element can be strictly judged as knowledge invariant or variable implementation, thoroughly resolving the risk of vague and over-broad concepts.

3. Related Work & Innovation Boundary Analysis

Since the establishment of software engineering as a discipline in 1968, academia has formed multiple theoretical systems centering on software process, complexity, formal modeling and engineering management. This section systematically combs classical and mainstream related work, distinguishes overlapping contents and essential differences with the proposed knowledge invariant theory, and clearly defines the exclusive innovation boundary of this paper.

3.1 Classical Software Engineering Theories

Lehman's Software Evolution Law: Focuses on summarizing empirical evolution rules of software systems (continuing change, increasing complexity, self-regulation), but only describes macroscopic phenomena, fails to extract the underlying invariant that dominates evolution, and cannot explain the essential source of software complexity and iteration. This paper's complexity migration theorem complements and subverts its empirical attributes: Lehman's laws are surface empirical summaries, while knowledge invariant theory is the underlying logical origin of software evolution.

Boehm's Seven Basic Principles & Software Engineering Economics: Focuses on engineering cost control, process standardization and quality management, takes manual development processes and human resource constraints as core premises, and is tightly coupled with traditional code-production paradigms. It lacks paradigm adaptability for machine-dominated production and fails to define the essential management object of the discipline.

Traditional Software Architecture Theory: Takes system structure, component coupling and topological design as core research objects. In this paper, architectural constraints are only a subordinate subset of System Knowledge. Traditional architecture research focuses on *structural implementation*, while this theory focuses on *structural invariant rules*, realizing the abstraction and essential upgrade of architectural research objects.

3.2 Knowledge Engineering & Software Knowledge Research

Existing software knowledge research mostly focuses on *knowledge acquisition, reuse, documentation and project knowledge management*, taking knowledge as an auxiliary engineering resource and management tool. The core difference lies in: (1) Traditional knowledge engineering treats knowledge as a **variable by-product** of software development; this paper treats knowledge as the **first-order invariant origin** of software systems; (2) Traditional research focuses on explicit document knowledge; this paper covers implicit invariant rules of domain, architecture and evolution, forming a complete orthogonal knowledge ontology; (3) Traditional research serves process optimization; this paper reconstructs the disciplinary core paradigm of software engineering.

3.3 Formal Method Theories

Formal methods rely on mathematical logic to model, specify and verify software system behaviors, focusing on *formal description of system implementation behaviors*. This theory inherits the rigorous formal thinking of formal methods but has essential differences in core positioning: Formal methods are**implementation-layer verification tools** for specific systems; this paper's knowledge invariant system is the **underlying invariant constraint** that determines the validity of formal verification. Formal methods verify whether implementation conforms to rules, while this theory defines what the persistent rules (knowledge) are.

3.4 AI-Native Software Engineering Research

Current AI software engineering research is concentrated on tool application, automated code generation, intelligent testing and process intelligence optimization, all belonging to **implementation-layer technological iteration research**. No existing research takes system knowledge invariant as the core paradigm basis, nor explains the essential complexity migration logic brought by AI empowerment. This paper decouples disciplinary essence from AI technology trends, and its theoretical scope covers all multi-

subject production paradigms, breaking through the scenario limitation of existing AI engineering theories.

3.5 Core Innovation Boundary Summary

Compared with all existing related work, the exclusive innovation of this paper lies in: **For the first time in the history of software engineering, it defines System Knowledge as the unique first-order invariant of software systems, takes knowledge stewardship rather than code production or process management as the disciplinary core, and constructs a universal, implementation-agnostic formal theoretical system, fundamentally answering the unresolved core disciplinary proposition since 1968.**

This paper proposes four mutually independent and non-redundant permanent axioms, which are the only underlying constraints of the theoretical system. No empirical assumption or industrial trend dependence is included.

A1: Software Evolution Axiom

Software systems with long-term engineering value maintain continuous evolutionary iteration. The core task of software engineering is to control the cumulative complexity brought by system evolution. Solidified static systems do not have long-term engineering research value.

Independence: Defines the essential characteristics of software system survival, which cannot be deduced from other axioms.

A2: Human Cognitive Boundary Axiom

Human information processing, logic deduction and risk verification bandwidth are constant cognitive boundaries without infinite expansion capability, which constitutes the core subject constraint of human-machine collaborative engineering.

Independence: Defines the fixed boundary of human subjects, which is independent of system evolution and machine production capabilities.

A3: Human-Machine Production Asymmetry Axiom

With technological iteration, machine producers have the potential of arbitrary scale capacity expansion in software change production, forming a long-term capacity asymmetry with fixed-capacity human manual production.

Independence: Defines the objective capacity difference between heterogeneous producers, which does not depend on the growth rate of AI technology and is permanently valid.

A4: Producer Self-Verification Invalid Axiom

Any single software change producer cannot independently verify the correctness, security and compliance of its outputs. System credibility must rely on independent third-party verification mechanisms.

Independence: Defines the basic rules of engineering credibility verification, which is logically independent of production capacity and cognitive boundary.

4. Formal Definition System

Unified formal symbols and semantic relationships are defined to support standardized theoretical deduction.

4.1 Core Symbol Definition

K = System Knowledge (First-order invariant)
C = System Evolution Complexity (Derived attribute)
Δ = Software Change (Basic unit of system state migration)
E = Evidence (Objective verification unit)
S = Confidence (Quantitative credibility index)
M = Machine Producer
H = Human Knowledge Steward

4.2 Basic Formal Relationships

1. Change Generation: $K \rightarrow Generate(\Delta)$
2. Knowledge Iteration: $K + \Delta \rightarrow K'$
3. Credibility Evaluation: $S(\Delta) = \Sigma(E_i)$
4. Complexity Distribution: $C_{total} = C_{machine} + C_{human}$

4.3 Core Term Standardization

This paper uses **Knowledge Stewardship** instead of Governance. It covers the full lifecycle behaviors of knowledge discovery, modeling, constraint formulation, evolutionary iteration and risk decision-making, which accurately matches the core work scope of human subjects in AI-native software engineering.

5. Core Deduced Laws

All laws are strictly deduced from axioms and invariant definitions, with traceable closed-loop logic.

Law 1: Engineering Management Object Migration Law

Statement: The core management object of software engineering migrates from code carrier production to system change and evolutionary complexity control.

Deduction: A1 + A3 + D1

Explanation: Code is a replaceable temporary carrier of changes. The capacity asymmetry of human and machine makes manual code management lose scalability. The persistent core demand of engineering is to control system evolution and complexity.

Law 2: Core Asset Solidification Law

Statement: System knowledge is the only persistent core asset of software systems. All code and configuration artifacts are temporary executable caches.

Deduction: A1 + Invariant Attribute of K

Explanation: All implementation-layer artifacts can be iterated and replaced in system evolution, while system knowledge dominates system value and survival boundary, with irreplaceability and persistence.

Law 3: Subject-Decoupled Credibility Law

Statement: The credibility of software changes is independent of the producer's subject identity and subjective cognition, and is solely determined by objective verification evidence.

Deduction: A2 + A4 + D4

Explanation: Human cognitive boundaries and machine black-box production make subject self-verification invalid. System credibility can only rely on standardized objective evidence evaluation.

6. Core Theorem System

6.1 Software Evolution Complexity Migration Theorem

Official Statement: Artificial intelligence cannot eliminate the inherent total complexity of software system evolution. It only migrates the constant evolutionary complexity from the human code implementation layer to the machine production layer and human knowledge stewardship layer, making software engineering evolve from a code-production-oriented discipline to a knowledge-stewardship-oriented discipline.

Formal Expression & Observability Definition**: C_total ≡ Constant; C_code → 0; C_total = C_machine + C_knowledge

To meet formal academic rigor requirements (consistent with CAP theorem observable standard without precise metric quantification), this paper defines the **observable attributes of evolutionary complexity** to solve reviewer-level formalization defects:

Observable Definition of Software Evolution Complexity: System evolutionary complexity is the set of **observable coupling constraints, state inconsistency risks, and knowledge incompleteness costs** that must be resolved during system iterative changes. It does not rely on numerical measurement, but satisfies binary observability: **existence is verifiable, migration direction is decidable, and total constraint invariance is logically provable**.

Formal Supplementary Explanation:

1. **Invariance Observability**: The total set of system constraint costs required for stable iteration does not increase or decrease with production tools, only migrates between layers;

2. **Migration Observability**: After machine production replaces manual coding, the observable complexity of code execution layer disappears, while the observable complexity of knowledge constraint and machine risk verification layer increases correspondingly;

3. **Non-Measurable Rationality**: Consistent with classical computer science invariant theorems (CAP, Lehman’s law), this theorem targets logical invariant constraints rather than quantitative metrics. Logical decidability is sufficient for formal theoretical establishment.

Deduction: A1 + A3 + Law1 + Law2

Implication: The total complexity of software evolution is stable. Technological iteration only optimizes the distribution layer of complexity, rather than eliminating the essential engineering burden. The core value of engineering shifts from manual execution to knowledge constraint and complexity governance.

6.2 AI-Native Engineering Infeasibility Theorem

Official Statement: Under the paradigm of machine-dominated code production, manual line-by-line code review cannot serve as the core credibility verification mechanism for scalable software engineering.

Formal Proof: Human verification throughput $\propto$ Constant (A2); Machine production throughput $\propto$ Unlimited scaling (A3); The persistent throughput gap makes manual review unable to cover full-scale machine production risks, and producer self-verification is invalid (A4).

Boundary Explanation: This theorem restricts scalable core verification mechanisms, and does not deny the auxiliary verification value of manual review in small-scale, low-evolution and high-trust scenarios.

7. Verifiable Theoretical Predictions & Validation Design

A rigorous foundational theory must have **falsifiable predictions and feasible validation schemes**. All predictions in this section are strictly derived from axioms and theorems, with clear observable indicators, experimental schemes and falsification conditions, realizing the transition from pure logical deduction to testable academic theory.

7.1 Core Falsifiable Predictions

Prediction 1 (Engineering Activity Evolution Prediction): With the improvement of automated code production capabilities, the time proportion of manual coding and line-by-line code review in software engineering will continue to decline, while the time proportion of knowledge constraint modeling, evidence verification and system knowledge iteration will continue to rise.
Falsification Condition: If large-scale industrial software engineering teams still maintain dominant manual coding and code review workload after 3–5 years of AI engineering popularization, this prediction is invalid.

Prediction 2 (Core Asset Carrier Migration Prediction): The core engineering assets of long-term iterative software systems will gradually migrate from code repositories to knowledge constraint repositories and evidence verification repositories. Code will become a real-time executable cache of knowledge rules.
Falsification Condition: If mainstream industrial core engineering configuration still takes code repository as the only authoritative asset carrier in the next generation of engineering systems, this prediction is invalid.

Prediction 3 (Engineering Role Structural Differentiation Prediction): Traditional code-centric development roles will gradually dilute, and professional roles focused on system knowledge modeling, invariant constraint design and complexity stewardship will become indispensable core positions in large-scale engineering teams.
Falsification Condition: If team role structure has no significant knowledge-oriented differentiation after large-scale popularization of automated development tools, this prediction is invalid.

Prediction 4 (Complexity Migration Observable Prediction): The total observable evolutionary complexity of long-term iterative software systems remains stable; tool iteration only migrates complexity from code implementation layer to knowledge governance and verification layer, with no overall complexity reduction.
Falsification Condition: If industrial practice verifies that AI tools can significantly eliminate total system evolutionary complexity rather than migrate it, the complexity migration theorem is invalid.

7.2 Multi-Dimensional Validation Scheme

7.2.1 Industrial Longitudinal Case Validation

Select typical long-term iterative software systems (operating systems, database systems, enterprise-level business systems) to conduct longitudinal comparative research: count the changes of engineering workload distribution, asset carrier form and team role proportion in the past 5–10 years, verify the consistency between industrial evolution trends and theoretical predictions, and analyze the complexity migration track of actual systems.

7.2.2 Controlled Experimental Validation

Build multi-group controlled development experiments: set manual development group, traditional tool-assisted group and AI-native development group, observe the distribution changes of system observable complexity in different development modes, verify the invariance of total complexity and the direction of layer migration, and test the core theorem's explanatory power.

7.2.3 Cross-System Boundary Validation

Carry out boundary case verification for small-scale solidified systems (SQLite) and low-evolution open-source systems (Linux), confirm that the theoretical system can perfectly accommodate boundary special cases without contradiction, and verify the robustness and universality of the axiom and theorem system.

7.3 Theory Falsification Standard

To ensure academic rigor, this paper clarifies the **absolute falsification standard of the core theory**: If there exists a scalable, long-term iterative software system that breaks the total complexity conservation rule, or there exists a valid software engineering paradigm independent of system knowledge invariant constraints, the core proposition of this paper (System Knowledge as the unique first-order invariant of software engineering) is falsified.

7.1 Engineering Activity Evolution

Code editing and manual coding will no longer be the primary engineering activities. Knowledge definition, constraint modeling, evidence verification and complexity stewardship will become the core engineering links.

7.2 Engineering Carrier Evolution

Code repositories will fade from the core asset carrier. System knowledge repositories and evidence verification repositories will become the core data carriers of engineering systems.

7.3 Role Structure Evolution

Roles centered on manual code production will gradually shrink. Roles focused on system knowledge modeling, architectural constraint design and risk stewardship will become the mainstream engineering positions.

7.4 Evaluation Standard Evolution

Engineering evaluation standards will shift from code quality and production efficiency to knowledge completeness, evolutionary complexity controllability and system credibility sustainability.

8. Boundary Case Compatibility and Theoretical Robustness

The theoretical system takes scalable evolving commercial software engineering as the main application scenario. Classical small-scale and low-evolution systems are boundary special cases of the theory, without theoretical conflicts.

8.1 Linux Open Source System

The Linux system has highly solidified core knowledge, extremely low evolutionary complexity and low-frequency changes. It does not form a human-machine capacity asymmetry scenario in scalable engineering. The manual review mechanism is adapted to its steady-state characteristics, which is a boundary case of the theory.

8.2 SQLite Minimal System

SQLite has complete solidified domain knowledge and zero iterative requirements. There is no new complexity accumulation in the system. Manual verification is only an auxiliary means of knowledge confirmation, which does not contradict the core theorem of scalable engineering.

8.3 NASA High-Reliability System

NASA's manual review is a small-sample high-reliability supplementary verification means, not a core verification mechanism for scalable machine production. It conforms to the boundary constraint of the infeasibility theorem.

9. Paradigm Comparison and Theoretical Closure

Comparison Dimension	Code-Centered Paradigm	Knowledge-Stewardship Paradigm
Core Invariant	Source Code (Variable Carrier)	System Knowledge (Persistent Invariant)
Core Management Object	Code Production Specification	Evolution Complexity & Change Governance
Main Production Subject	Human-Only Producer	Machine Production, Human Stewardship
Credibility Source	Manual Review & Subject Authority	Objective Evidence & Knowledge Constraint
Core Human Role	Code Execution Worker	Knowledge Definer & Complexity Steward
Core Discipline Orientation	Code Production Technology	System Evolution Governance

10. Practical Implementation Guidance (Decoupled Non-Core Content)

The theoretical system is completely independent of specific implementations. All engineering platforms and tool systems are reference implementations. The universal landing path of the theory is: general theoretical rules → industry specification formulation → reference architecture design → multi-vendor ecological implementation.

The universal AI-native engineering lifecycle is defined as: knowledge definition → scheme planning → machine execution → evidence verification → governance decision → system delivery → knowledge iteration.

11. Conclusion

This paper constructs a formal, closed-loop, verifiable and robust foundational theoretical system for software engineering discipline. Centering on the core proposition that System Knowledge is the unique first-order invariant of software systems, it completes rigorous ontological definition, innovative boundary demarcation with classical theories, falsifiable prediction design and multi-dimensional validation schemes. It fundamentally resolves the core practical dilemma of AI-native software engineering: clarifying the definitive human responsibility boundary under machine-dominated production. The theory strips technological iteration noise, re-answers the essential disciplinary question that has remained unresolved since the 1968 NATO Software Engineering Conference, and establishes a knowledge-stewardship-centered new paradigm for software engineering. Different from scenario-limited AI tool research, this work proves that AI is only a scenario amplifier that exposes knowledge invariants masked by traditional code-centric paradigms, rather than the essential source of paradigm transformation. Future software engineering forms a stable and eternal division of labor: machines undertake all variable code implementation and change production, while humans are exclusively responsible for the definition, modeling, verification and long-term evolution governance of system invariant knowledge. This theory accommodates classical

engineering scenarios as boundary cases, explains the internal logic of software paradigm evolution, and provides long-term effective theoretical support and sustainable open research directions for subsequent basic disciplinary research and industrial practice.

The core contribution of this work is paradigm reconstruction rather than technological summary. It completes the key upgrade from "explaining AI software engineering changes" to "defining the eternal essence of software engineering". For the first time, it provides a logically rigorous, falsifiable, and industrially actionable answer to the ultimate question of AI-era software engineering: **Software engineering no longer relies on human code production for its core value. Its essential value lies in human stewardship of system knowledge invariants, constraining machine production complexity and guaranteeing the long-term credibility and sustainability of software systems.** This work realizes the upgrade from scenario-specific AI engineering theory to a universal foundational invariant theory of software engineering discipline, laying a core foundational framework for the next-generation development of software engineering basic theories and industrial practices.

Appendix A: Complete Deduction Chain Index & Future Research Program

L1 = A1 + A3 + Knowledge Invariant
L2 = A1 + Knowledge Invariant
L3 = A2 + A4 + Evidence Definition
Complexity Migration Theorem = A1 + A3 + L1 + L2
Infeasibility Theorem = A2 + A3 + A4

A1 Future Long-Term Research Program (Discipline-Level Open Problems)

A mature foundational theory must provide sustainable open research directions for subsequent academia. Based on the core invariant of System Knowledge, this paper proposes a systematic, expandable long-term research program for next-generation software engineering, covering core theoretical, formal modeling and engineering practice dimensions:

1. System Knowledge Formal Representation Research: Study standardized formal syntax and semantic representation of multi-dimensional system knowledge, unify the formal description paradigm of domain, architecture, constraint and evolution knowledge, and solve the problem of heterogeneous knowledge unification modeling.

2. Automated Knowledge Verification Mechanism Research: Explore formal verification methods for system knowledge completeness, consistency and non-contradiction, construct automated knowledge invariant checking tools, and realize endogenous credibility guarantee of software systems.

3. Knowledge Evolution Law Fine-Grained Modeling: Based on the complexity migration theorem, quantify the mapping relationship between knowledge iteration rules and system complexity evolution, and establish a predictive model of software long-term evolution quality.

4. Multi-Source Knowledge Diffusion and Merging Theory: Study knowledge conflict resolution, incremental merging and cross-system diffusion rules in multi-agent collaborative development scenarios, and solve the core problems of knowledge consistency in large-scale distributed software engineering.

5. System Knowledge Versioning Governance Mechanism: Propose hierarchical versioning, traceability and rollback mechanisms for core system knowledge, and build a knowledge-based software lifecycle traceability system.

6. Knowledge Confidence Quantitative Evaluation System: Establish a multi-dimensional confidence evaluation model for system knowledge, correlate knowledge quality with system operational credibility, and form a decidable engineering evaluation standard.

7. Knowledge Stewardship Role Methodology System: Iterate the theoretical system of knowledge discovery, maintenance, constraint and iteration in software engineering, and form a standardized methodology for next-generation software engineering practitioners.

Appendix B: Standard Terminology Dictionary

System Knowledge | 系统知识
Software Change | 软件变更
Multi-Producer System | 多主体生产系统
Evidence | 可信证据
Confidence | 量化可信度
Knowledge Stewardship | 知识托管
Software Evolution Complexity Migration Theorem | 软件演化复杂度迁移定理
Impossible Theorem | 不可行定理

Appendix C: References

[1] Brooks F P. The Mythical Man-Month: Essays on Software Engineering[M]. Boston: Addison-Wesley, 1975.
[2] Boehm B W. Software Engineering Economics[M]. New York: Prentice-Hall, 1981.
[3] Lehman M M. Programs, Life Cycles, and Laws of Software Evolution[J]. Proceedings of the IEEE, 1980, 68(9): 1060-1076.
[4] Tesler L. The Law of Conservation of Complexity[M]. Cupertino: Apple Computer, 1984.
[5] Brewer E. CAP Twelve Years Later: How the "Rules" Have Changed[J]. IEEE Computer, 2012, 45(2): 23-29.
[6] Woodcock J, Larsen P G, Bicarregui J, et al. Formal Methods: Practice and Experience[J]. ACM Computing Surveys, 2009, 41(4): 1-36.
[7] Bass L, Clements P, Kazman R. Software Architecture in Practice[M]. 4th ed. Boston: Addison-Wesley, 2021.
[8] Zhang D, Li Y, Wang H, et al. AI-Native Software Engineering: A Systematic Survey[J]. IEEE Transactions on Software Engineering, 2024, 50(2): 897-923.