



OPEN AGENTIC INDUSTRY CONSORTIUM
开放智能体产业联盟

OAIC WHITE PAPER

Agentic Software Engineering

Industry White Paper

From Human-Written Code to
Human-Defined Requirements
and Agent Implementation

Definitions, Framework, and
Three Core Engineering Capabilities

Release 1 | October 2026

Open Agentic Industry
Consortium (OAIC)

www.oaic.ai



Copyright and Intellectual Property Notice

© 2026 Open Agentic Industry Consortium (OAIC). All rights reserved.

This white paper is published by OAIC. This copyright notice covers the content and compilation of this publication to which OAIC holds rights. Rights in third-party materials, software, and trademarks, as well as rights retained by authors under applicable law or agreement, remain with their respective holders.

Except as permitted by applicable law or a separate license from the rights holder, reproducing, adapting, translating, republishing, or redistributing this white paper requires permission from the relevant rights holder. Complete source and copyright information must be retained. Citations should identify the title, version, publisher, and location of the material cited.

Intellectual Property and Patents

The methods, technologies, and implementations discussed in this white paper may involve third-party intellectual property, including patents. Publication does not grant any patent, trademark, or other intellectual property license, nor does it imply that OAIC has completed a patent search, rights clearance, or infringement risk assessment. OAIC makes no determination about the existence, validity, or scope of any relevant patent. Implementers should investigate the rights relevant to their intended use and obtain permission from the rights holder when necessary.

This white paper does not promise that the technologies described can be implemented free of charge. It does not constitute an open-source license or a commitment to license patents on fair, reasonable, and nondiscriminatory terms. Referenced software, code examples, and open-source projects remain subject to their own licenses; this publication does not alter those terms.

Trademarks and Use of This Publication

The OAIC name and logo identify the publisher of this white paper. Other names, trademarks, and logos belong to their respective rights holders. Their inclusion does not imply endorsement of this publication.

This white paper is intended to support industry dialogue and serve as a reference for engineering practice. It is not a mandatory standard, a basis for certification, or legal advice. The views expressed do not necessarily represent the official positions of the authors' affiliated organizations. Information and technical practices may change over time. Readers should exercise independent judgment in light of their circumstances, risks, and applicable requirements.

To the extent permitted by applicable law, this publication is provided as is. OAIC makes no warranty of fitness for a particular purpose, freedom from error, or noninfringement. Users should conduct the necessary verification and professional assessments. This notice does not exclude or limit any liability that cannot lawfully be excluded or limited.

Permissions, corrections, and collaboration: www.oaic.ai

Authors and Editorial Committee

Name	Affiliation
Bingsheng Ru	Tencent Research Institute
Dong Sun	Futurewei
Yun Lin	Shanghai Jiao Tong University
Jianzhong Li	Singularity Intelligence Research Institute
Zijiang Yang	University of Science and Technology of China

Published by the Open Agentic Industry Consortium (OAIC)

Release 1 | October 2026

Intended audience: enterprise business and technology leaders; consulting and research organizations; software and AI companies; universities; and research institutions.

Website: www.oaic.ai



Scan to learn about OAIC

Contents

Executive Summary	6
A.1 THE CORE ARGUMENT	6
A.2 ASE IN ONE SENTENCE	6
A.3 THREE MAIN CONCLUSIONS	7
A.4 THE MAIN WORKFLOW	7
Part I Why Software Engineering Must Change	8
Chapter 1 From "Humans Write Code" to "Humans Specify, Agents Implement"	8
1.1 EXPANDING CAPABILITIES: FROM CODE COMPLETION TO SUSTAINED ENGINEERING WORK	8
1.2 THE PREVIOUS PARADIGM: PEOPLE AS THE PRIMARY EXECUTORS AND CONTROL CENTER	9
1.3 THE NEW PARADIGM: HUMAN INTENT AND ACCOUNTABILITY SET THE BOUNDARIES; AGENTS ARE THE PRIMARY EXECUTORS	9
1.4 FOUR STRUCTURAL CHANGES	10
1.5 HUMAN ROLES AND RESPONSIBILITIES THAT CANNOT BE TRANSFERRED	10
Chapter 2 Why the New Paradigm Requires a New Engineering System	12
2.1 MODEL CAPABILITY IS NOT SOFTWARE PRODUCTION SYSTEM CAPABILITY	12
2.2 THE TENSION BETWEEN PROBABILISTIC REASONING AND DETERMINISTIC ENGINEERING ACCOUNTABILITY	14
2.3 NEW FAILURE MODES AND ENGINEERING DEBT	14
2.4 EXTEND TRADITIONAL SOFTWARE ENGINEERING—DO NOT DISCARD IT	15
2.5 PART I SUMMARY	15
Part II What Is Agentic Software Engineering?	16
Chapter 3 ASE's Definition, Engineering Targets, and Boundaries	16
3.1 FORMAL DEFINITION	16
3.2 TWO ENGINEERING TARGETS	16
3.3 BOUNDARIES WITH RELATED CONCEPTS	17
3.4 SCOPE AND NON-GOALS OF THE FIRST RELEASE	17
Chapter 4 ASE Reference Framework, Capability Elements, and Core Capabilities	18
4.1 THE OVERALL LOOP FROM REQUIREMENTS TO EVIDENCE	18
4.2 FOUR-LAYER REFERENCE FRAMEWORK	18
4.3 SIX CAPABILITY DOMAINS AND THEIR KEY ELEMENTS	19
4.4 THREE CORE CAPABILITIES FOR THE INITIAL RELEASE	19
4.5 FIVE ENGINEERING PRINCIPLES	20
4.6 RESPONSIBILITY BOUNDARIES: HUMANS, AGENTS, AND THE PLATFORM	20
Chapter 5 Full Lifecycle, Critical Artifacts, and the Quality Loop	22
5.1 SIX-STAGE LIFECYCLE	22
5.2 MINIMUM ARTIFACTS AND STAGE GATES FOR EACH STAGE	22

5.3 EVIDENCE CHAIN ACROSS THE LIFECYCLE	23
5.4 QUALITY OBJECTIVES	24
5.5 HUMAN-AGENT ACCOUNTABILITY GATES IN THE LIFECYCLE.....	24
Part III How to Build the Three Core Capabilities.....	26
Chapter 6 Specification Development: Converting Human Intent into Engineering Contracts Agents Can Execute ..	26
6.1 UPGRADING SPECIFICATIONS FROM DESCRIPTIVE DOCUMENTS TO RUNTIME CONTROL OBJECTS	26
6.2 FOUR MINIMUM CATEGORIES OF SPECIFICATIONS.....	26
6.3 THE REQUIREMENTS COMPILATION PROCESS	27
6.4 SPECIFICATION QUALITY AND CONFLICT RESOLUTION.....	28
6.5 SPECIFICATION CHANGES AND IMPACT ANALYSIS	29
6.6 MINIMUM PRACTICE TEMPLATE.....	29
Chapter 7 Testing and Evaluation: Examining Outcomes, Processes, and Software Delivery Capability	30
7.1 DEFINE THE TEST OBJECTS	30
7.2 LAYERED TESTING AND EVALUATION METHODS	31
7.3 TEST ORACLES: WHO DECIDES WHAT IS "CORRECT"?	31
7.4 STATISTICAL DISCIPLINE AND REPRODUCIBILITY	32
7.5 EVIDENCE-DRIVEN RELEASE GATES.....	33
7.6 MINIMUM PRACTICE TEMPLATE.....	33
Chapter 8 Verification and Trustworthiness Assurance: Constraining High-Risk Behavior with Layered Evidence ...	35
8.1 DISTINGUISH "BUILDING THE RIGHT THING" FROM "BUILDING THE THING RIGHT"	35
8.2 RISK LEVELS DETERMINE THE STRENGTH OF ASSURANCE	35
8.3 LAYERED VERIFICATION MECHANISMS.....	36
8.4 PRIORITIES FOR FORMALIZATION	37
8.5 AVAILABLE FORMAL METHODS AND A PROGRESSIVE ADOPTION PATH	37
8.6 OVERALL ASSURANCE MODEL	38
8.7 BOUNDARIES OF FORMAL VERIFICATION AND HONEST CLAIMS	39
8.8 MINIMUM PRACTICE TEMPLATE.....	40
Conclusion: Starting with a Minimal Observable and Verifiable Closed Loop.....	41
References	42
Appendix A: Checklist for a Minimal ASE Closed Loop	44
Appendix B: Terminology.....	46

Executive Summary

A.1 The Core Argument

Systems built around large language models (LLMs) have evolved from code generation tools into agents that use context, develop plans, call tools, modify software artifacts, and iteratively repair their work in response to feedback. This evolution is changing how software production is organized.

The primary actors and control mechanisms in software production are changing, with implications that extend beyond faster coding. People traditionally led both execution and control. As agents take on more of the execution, control must shift from people overseeing each step to people defining boundaries, systems producing evidence, and people making acceptance decisions based on risk. These changes give rise to an agent-based software development approach organized around requirements compilation [18] (see Section 6.3).

This white paper describes the emerging paradigm as follows:

People define intent and accountability boundaries. Agents perform engineering tasks. The system produces auditable evidence, and people use that evidence to make risk-based acceptance decisions and hold the responsible parties accountable.

People remain involved. Their focus shifts from writing code directly to defining requirements, designing systems, verifying results, and taking responsibility.

This white paper proposes that people set goals and retain accountability, agents interpret tasks and orchestrate execution, and people, agents, and platforms jointly develop and use verification and rollback capabilities. People define goals, boundaries, and acceptance criteria, confirm critical interpretations of requirements, and retain ultimate responsibility. Agents interpret context and tasks, coordinate work and tools, and implement, test, and repair software. Verification and rollback, including compensating actions and loss containment, are system capabilities shared across tasks and participants.

A.2 ASE in One Sentence

This white paper defines Agentic Software Engineering (ASE) as:

A software engineering paradigm in which agents perform the primary engineering execution tasks, including planning, implementation, testing, and repair, within human-defined goals, specifications, constraints, and acceptance criteria, while the engineering system produces auditable evidence throughout the process so that people can make risk-based acceptance decisions and enforce accountability.

The definition has four essential elements: human intent and accountability boundaries, which establish who defines the task and who is responsible; agent execution, which establishes who does the work; auditable evidence, which provides the basis for judgment; and risk-based acceptance and accountability, which establish who authorizes or preauthorizes actions and who approves results. An approach missing any one of these elements is outside the scope of ASE as defined here.

A.3 Three Main Conclusions

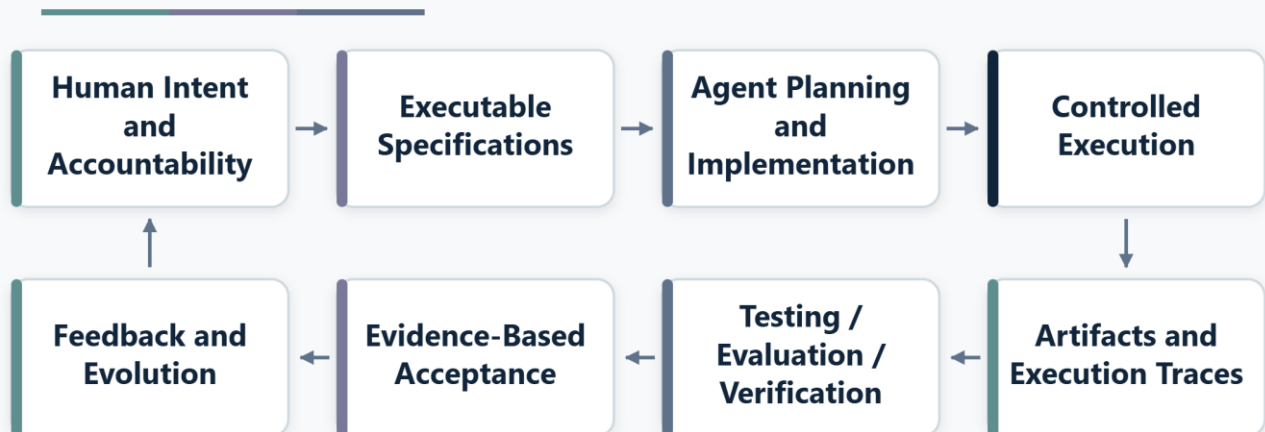
Specifications become the center of control. As execution shifts from people to agents, requirements documents must evolve from explanations intended for human readers into engineering contracts that agents can interpret and execute, and that can be checked and traced. Specifications become runtime control objects used throughout development, testing, release, and operation, rather than documents prepared only before development begins.

Testing and verification become a major bottleneck. As implementation costs fall, the cost and importance of determining whether the work is correct and deliverable, and whether responsibility can be assigned, rise in parallel. Public benchmarks and industrial tasks differ in task distributions, environmental constraints, and acceptance criteria. Benchmark scores therefore do not directly represent software delivery capability. Requirements expression, context quality, testing, evaluation, and verification jointly constrain ASE's effective engineering throughput.

Software delivery capability comes from the system, not the model alone. Outcomes depend on the combined effects of the model, context, runtime control harness, tools, permissions, execution loop, testing and verification, and governance. The model supplies foundational capabilities; the system determines whether those capabilities translate into consistent engineering output. Improvements in model capability must be integrated with specifications, testing, verification, and governance in a closed loop to produce sustainable software delivery capability.

A.4 The Main Workflow

ASE forms a closed loop from intent to evidence and from evidence back to intent:



Three recommendations follow. First, measure ASE by the proportion of artifacts that meet delivery criteria and by the completeness of their evidence, rather than by the volume of code generated. Second, begin with specifications and acceptance criteria, and use them to constrain tools and workflows; do not build the platform first and add specifications later. Third, do not pursue autonomy across the board. First establish a minimal closed loop in settings where risks can be controlled, behavior is observable, and changes can be rolled back, and then gradually expand the scope of autonomy.

Part I Why Software Engineering Must Change

Chapter 1 From "Humans Write Code" to "Humans Specify, Agents Implement"

This chapter addresses a central question: Why do the changes brought by LLMs and agents represent a shift in the software engineering paradigm, rather than simply an upgrade to tools that improve efficiency?

1.1 Expanding Capabilities: From Code Completion to Sustained Engineering Work

The evolution of agent capabilities in software engineering can be described through five progressively broader forms of use. These are not formal industry maturity levels, nor do they imply that every system has reached the fifth form:

1. Code completion and question answering: The system suggests code fragments as a person writes software. The person remains the sole executor and decision-maker.
2. Cross-file changes: The system understands a local portion of a project's structure and makes consistent changes across multiple files under step-by-step human direction.
3. Tool use: The system invokes external tools such as compilers, testing frameworks, package managers, and command-line tools, moving from "generating text" to "changing the environment."
4. Task planning and iteration: The system breaks goals into subtasks and adjusts its plan based on execution results, forming an "observation–decision–action–verification" loop.
5. Long-running, autonomous engineering execution: The system works continuously for extended periods in a constrained environment, handles exceptions autonomously, traces failed execution paths, and stops when necessary.

The third form marks a critical transition. Tool use turns an agent from a text generator into a system that changes its environment. The fourth and fifth forms support sustained, multi-step work rather than one-time outputs. Executing a task does not mean assuming legal or organizational responsibility.

Public benchmarks show both advances in capability and the limits of evaluation. SWE-bench uses issues from real repositories and tests to evaluate fixes; SWE-Bench Pro places additional emphasis on long-running tasks and the risk of data contamination. [2][3] Benchmarks and their subsets differ in task difficulty, sample selection, harnesses, and compute budgets. Scores from different benchmarks therefore cannot be combined into a growth curve measured on a common basis, much less used to infer the proportion of changes an enterprise can merge

or deliver. Organizations must still validate software delivery capability using their own tasks, permissions, and acceptance criteria.

1.2 The Previous Paradigm: People as the Primary Executors and Control Center

Traditional software engineering, including Agile and DevOps practices, rests on an implicit assumption: People are both the primary executors of software production and its main control center. Its characteristics include:

- **People-centered knowledge:** Requirements, design intent, and architectural constraints are distributed across people, documents, and engineering artifacts. Some tacit knowledge is conveyed through collaboration.
- **A combination of human and automated control:** Code review, continuous integration, testing, and release gates detect and correct deviations throughout the process.
- **Layered quality assurance:** Testing, static analysis, reviews, and formal methods where needed jointly provide evidence. No single activity offers complete coverage.
- **Accountability established through organizational processes:** Version histories identify contributors, but product responsibility must still be defined through roles, approvals, and governance arrangements.
- **Implementation as a major cost:** Coding and debugging require substantial human effort. Requirements, integration, verification, and organizational coordination can also become bottlenecks.

This model relies on professional judgment and communication to handle tacit knowledge and exceptions. That does not mean traditional engineering can accept imprecise specifications or incomplete testing. ASE retains these engineering disciplines and makes more implicit judgments explicit and structured to support the way agents execute work.

1.3 The New Paradigm: Human Intent and Accountability Set the Boundaries; Agents Are the Primary Executors

When agents become the primary executors, these assumptions must be extended. Agents do not automatically acquire the tacit knowledge embedded in project history, team conventions, and business context. With insufficient constraints, they may fill gaps in ambiguous requirements in ways that are statistically plausible but not correct from an engineering standpoint. The volume of artifacts they produce at high speed also increases the burden of reviewing every item manually.

The new paradigm therefore requires a different approach to control:

- **Capture tacit knowledge in specifications:** Tacit knowledge must be made explicit in executable specifications, making specifications the primary interface between people and agents.

- **Control execution through boundaries and evidence-based acceptance:** Retain necessary code reviews, constrain what agents may do, and require evidence that both machines and people can inspect.
- **Support quality through evidence chains:** Testing, static analysis, formal verification, and operational observation jointly provide evidence. Coverage extends from software outcomes to agent execution traces and the production process. Quality conclusions are drawn from the combined evidence, rather than depending on a person to judge every output.
- **Design accountability explicitly:** This white paper does not treat an agent as a bearer of legal, ethical, or organizational responsibility. Responsibility must rest with organizations and their authorized personnel.
- **Bottlenecks may shift to acceptance:** As implementation becomes more efficient, the ability to express requirements, the quality of context, and the ability to verify results may become new constraints.

1.4 Four Structural Changes

Change	How It Appears	Direct Consequence
Who Executes the Work	Agents perform most execution work; people direct goals, boundaries, and governance	Implementation efficiency may improve; output no longer depends solely on human coding effort
Workflow	A shift from stage handoffs to continuous planning, execution, feedback, and repair	Specifications, actions, and evidence form an inspectable feedback loop
Deliverables	A code-centered baseline expands to include specifications, context, constraints, code, and evidence for software production	Producing and managing evidence as engineering artifacts adds work
Bottlenecks	Constraints expand from coding speed to requirements expression, context quality, result verification, and traceability of responsibility	Organizations need to systematically develop capabilities in specifications, context, testing and verification, and governance

These changes are interconnected. A change in who executes the work drives workflows toward continuous feedback loops. Those loops require specifications, context, artifacts, and evidence to be managed together. Engineering bottlenecks consequently extend beyond the efficiency of individual implementation tasks to the quality and controllability of the entire software production system.

1.5 Human Roles and Responsibilities That Cannot Be Transferred

In the new paradigm, people's primary contribution shifts from direct implementation to four functions. This does not rule out manually coding critical modules, reviewing code, or taking over execution:

1. **Requirements definition:** Establish goals, boundaries, and acceptance criteria, translating ambiguous intent into requirements that can be assessed.

2. **System design:** Define architecture, interfaces, constraints, and nonfunctional requirements, and establish the scope of actions available to agents.
3. **Verification:** Design independent mechanisms for assessing results, use evidence to determine whether they are acceptable, and review critical implementations according to risk.
4. **Accountability:** Assume the applicable responsibilities for products, compliance, safety, and ethics under the relevant laws and organizational policies.

Traditional Software Engineering	Agentic Software Engineering
Human-led production and process control	Humans define intent and accountability; agents lead execution
Requirements Documents, contracts, and engineering artifacts	Requirements Checkable engineering specifications
Execution Human-led, with automated tool support	Execution Agents plan, implement, test, and repair
Control Human review and automated gates	Control Boundary constraints and risk-tiered acceptance
Quality Layered testing, analysis, and review	Quality Evidence from products, traces, and the production system
Delivery Code, documentation, and verification records	Delivery Specifications, code, traces, and acceptance evidence
Bottleneck Implementation and organizational coordination	Bottleneck Expression, context, verification, and related factors

Figure 1 Comparison of Traditional and Agentic Software Engineering Paradigms

Within an organization, different roles typically perform these functions. Business leaders define value objectives, risk appetite, and acceptable outcomes. Product and architecture personnel translate intent into enforceable boundaries and system designs. Engineering and quality personnel are responsible for the software production system, testing mechanisms, verification strategies, and stage gates. Organizational leaders establish the scope of authorization, assignment of responsibility, audit requirements, and handling of exceptions. One person may perform several functions, but every function must have an explicitly accountable owner.

Using agents must not create an accountability gap. Accountability must remain with the organizations and people identified by applicable law and organizational policies. The EU AI Act establishes requirements such as human oversight for high-risk AI systems within its scope. ISO/IEC 42001 sets out requirements for an AI management system, and the NIST AI Risk Management Framework treats governance as a cross-cutting function. [8]–[10] These sources differ in their legal status, scope of application, and implementation arrangements. They do not establish a uniform global obligation, nor do they imply that all software using agents automatically qualifies as a high-risk AI system. Autonomous execution can be authorized, but responsibility must remain clearly assigned.

Chapter 2 Why the New Paradigm Requires a New Engineering System

This chapter asks why neither stronger models nor existing software engineering processes, on their own, are sufficient to support production-grade software development with agents.

2.1 Model Capability Is Not Software Production System Capability

A capable model does not make engineering problems disappear. Production outcomes depend jointly on the following elements; omitting any one can substantially weaken overall performance:

Production outcomes = f(model, context, harness, tools, permissions and sandboxes, execution loop, testing and verification, governance mechanisms)

A runtime control harness organizes and enforces the runtime behavior and control policies of an agent system built around a model and supported by an underlying runtime environment. Its responsibilities include orchestrating model and tool calls; assembling context according to the task and policies; managing session and task state; enforcing permissions, isolation, budgets, and stopping policies; and providing mechanisms for observation, verification, checkpoints, and exception recovery. It does not require changes to model parameters. Instead, it uses context, tools, state, and feedback to turn model capabilities into controlled, verifiable system capabilities that support sustained operation and produce engineering artifacts.

A development framework provides the components and abstractions needed to build a system. A runtime provides execution mechanisms such as processes, containers, queues, networking, storage, resource limits, and scheduling. A harness can be built using a development framework and execute on a runtime, but its core responsibility is to organize system behavior and enforce runtime policies. It is not synonymous with either a development framework or the underlying runtime environment.

Context is the complete input available to a model in a single invocation. The harness selects and assembles it according to task boundaries and policies from instructions, specifications, task state, session history, memory,

knowledge, tool results, and verification results. It is the information view for the current invocation, not the entirety of the system's state, session history, or memory.

An execution loop is a feedback control process that iterates toward a task objective. It can be summarized as "observation–decision–action–verification": Observe context and environmental state, decide on the next plan or action, execute within permission and policy constraints, and incorporate results and verification feedback into subsequent decisions. Every loop should specify its purpose, progress measures, number of iterations, time and cost budgets, repeated-state detection, termination conditions for success and failure, escalation thresholds, and rules for pausing and resuming.

An execution graph uses nodes, edges, and shared state to represent execution dependencies, routing, parallel execution, and joins. Nodes may be model invocations, tool operations, deterministic checks, human reviews, or bounded subloops. A loop concerns how to continue, correct course, stop, or escalate after obtaining a result. A graph concerns which node executes next and how state and evidence propagate through dependencies. An execution graph may contain cycles or describe an acyclic workflow; this white paper therefore does not treat a loop and a graph as synonyms. Complex tasks can introduce execution graphs as needed, while simple tasks need not add graph-based orchestration.

These elements are cooperating engineering responsibilities, not isolated product layers or a strict hierarchy of nested sets. A harness typically implements the execution loop and can host the runtime control mechanisms for an execution graph. Context, tools, permissions and sandboxes, and the execution loop are listed separately in the formula to emphasize that each requires its own engineering investment. Appendix B explains the relevant terms and boundaries. [17]

The same model can produce substantially different output quality under different harnesses, context quality, and testing constraints. These differences must be measured using the same tasks and budgets. ASE's engineering priority is therefore to organize these elements into a closed feedback loop, rather than simply wait for model capabilities to improve.

Existing engineering practices support long-running execution through task decomposition, session handoff artifacts, incremental checks, and end-to-end testing. [16] This white paper groups these practices under harness and loop engineering. A harness supplies tools, context, state, and control mechanisms, and enforces policies. A loop organizes iterations of observation, decision, action, and verification, while managing budgets and termination conditions. Together, they provide ASE's execution foundation, but they must still be combined with specifications, independent acceptance, and governance across the full lifecycle. Controls for permissions, evidence, verification, and stopping should remain in the harness over the long term, independently of any particular model. Temporary scaffolding that compensates for weaknesses in current models should be retired promptly as model capabilities improve, to avoid accumulating technical debt.

2.2 The Tension Between Probabilistic Reasoning and Deterministic Engineering Accountability

LLM outputs may be nondeterministic: The same input may produce different results across runs, and a model may confidently reach a wrong conclusion. Engineering accountability, however, requires explicit assessment criteria, risk thresholds, and accountable parties. Some functionality can be accepted against deterministic conditions, while properties such as reliability also require statistical evidence. Not every engineering property can be reduced to a binary decision. The essential requirement is to manage uncertainty through mechanisms that can be checked.

This tension cannot be eliminated; it can only be managed through engineering. Three basic approaches are:

1. **Constrain probabilistic outputs with deterministic mechanisms:** Use type systems, schema validation, contract checks, static analysis, and formal methods to establish deterministic boundaries around probabilistic outputs.
2. **Establish stability through repetition and statistics:** Run nondeterministic processes repeatedly and aggregate their results statistically. Base judgments on distributions rather than individual results.
3. **Replace trust with evidence:** Do not ask an agent to certify its own correctness. Require evidence that can be independently checked.

2.3 New Failure Modes and Engineering Debt

When agents enter production, they introduce failure modes that are uncommon in traditional software engineering or differ substantially in degree:

- **Compounding deviations:** Errors in interpreting requirements or planning grow throughout a long-running task and remain hidden until its final state, increasing the cost of rework.
- **Superficial conformance:** An implementation passes tests without meeting the actual requirements, or the tests themselves are changed or bypassed, making a "pass" meaningless.
- **Hidden regressions:** A change passes every existing test but introduces defects in paths the tests do not cover.
- **Hallucinated dependencies:** The implementation references nonexistent interfaces, library functions, or configuration options, and those references are not exercised in the local environment.
- **Architectural drift:** Repeated locally optimal changes accumulate, gradually moving the system away from its intended architecture without anyone overseeing the whole.
- **Understanding and evidence debt:** Code works, but personnel cannot explain critical design decisions or reconstruct the basis for changes, undermining later maintenance, audits, and incident handling.
- **Governance debt:** Unclear permission boundaries, incomplete audit logs, and rollback paths that have not been rehearsed prevent rapid loss containment when an incident occurs.

- **Interaction and execution risks:** These include prompt injection, context contamination, state drift, unauthorized actions or irreversible side effects, unbounded retries and budget exhaustion, and dependency mismatches or conflicts among multiple agents. Controls should cover input sources, state, tools, and stopping conditions.

These failures are difficult to detect in a single check. As they accumulate, they substantially increase long-term maintenance costs and become major sources of engineering debt in the new paradigm.

2.4 Extend Traditional Software Engineering—Do Not Discard It

ASE extends traditional software engineering rather than replacing it. Practices such as version control, code review, continuous integration, layered testing, defect management, and release management remain useful, but their implementation must adapt:

Traditional Practice	Extension in ASE
Version Control	Version specifications, prompts, evaluation sets, and agent configurations as well as code
Code Review	Shift the focus of human review from implementation details toward specification conformance, architectural consistency, and completeness of evidence
Continuous Integration	Expand gates from "build and tests pass" to "specification checks, tests, evaluations, security checks, and compliance checks all pass"
Layered Testing	Add an evaluation layer for agent behavior and the software production process
Release Management	Shift release decisions from human judgment to evidence-driven automated gates combined with human release authorization based on risk

Traditional software engineering asks how to build software reliably. ASE must also address how to reliably assess the trustworthiness of software built by machines.

2.5 Part I Summary

First, the software production paradigm has undergone a structural change rooted in who executes the work and how execution is controlled, rather than in isolated efficiency gains. Second, model capability is necessary but not sufficient; software delivery capability comes from systems engineering. Third, risk spans requirements, execution, acceptance, and operation, and cannot be deferred until final acceptance. Fourth, existing engineering systems must expand while retaining the core disciplines of version control, review, testing, and release management.

The next two parts address, in sequence, what ASE is (WHAT) and which three core capabilities its first release should prioritize (HOW).

Part II What Is Agentic Software Engineering?

Chapter 3 ASE's Definition, Engineering Targets, and Boundaries

This chapter asks what ASE studies and engineers, and what falls outside its scope.

3.1 Formal Definition

Building on the one-sentence definition in Section A.2, this white paper defines ASE formally as follows:

ASE consists of software engineering methods and a software production system organized around "humans specify, agents execute." Agents perform the primary execution work of software engineering within human-defined goals, specifications, constraints, and acceptance criteria. The engineering system manages context, processes, and artifacts through executable specifications, controlled execution, testing and evaluation, and verification-based assurance, producing auditable evidence. Organizations and authorized personnel make risk-based acceptance and release decisions and hold the responsible parties accountable.

This definition emphasizes three points. First, agent execution must remain within the boundaries of human authorization; exceeding those boundaries is a failure to prevent and control. Second, evidence is as important as code. Third, human accountability is essential, but it does not require a person to click "approve" for every execution. Low-risk tasks may be released automatically under preapproved policies and gates; high-risk situations and exceptions require explicit human decisions. Preauthorization is itself an acceptance decision made by a person in advance: Authorized personnel establish acceptance criteria, release policies, sampling checks, and escalation conditions, and remain responsible for the outcomes. Automated release therefore does not change the principle that final acceptance rests with people.

3.2 Two Engineering Targets

ASE addresses two interrelated engineering targets:

The first is engineering with agents. Agents participate in requirements, design, implementation, testing, and repair to deliver conventional software or software products that incorporate AI. The product itself need not be an agent system.

The second is engineering agentic systems themselves. This covers models, harnesses, tools, context, state, memory, execution loops, collaboration, and operational governance. Agentic systems used to produce software are one type within this category.

These two targets share capabilities in specifications, testing, verification, evidence, and governance, but neither substitutes for the other. An organization using agents to produce software must assure both the quality of the delivered software and the controllability of the agentic software production system. Both assurance targets recur throughout the following chapters.

3.3 Boundaries with Related Concepts

Concept	Primary Focus	Relationship to ASE
AI4SE / AI-Assisted Software Engineering	Using AI to assist software engineering activities such as requirements, coding, and testing	Overlaps with ASE; this white paper emphasizes sustained agent execution and an end-to-end governance feedback loop
Agentic Coding	Agents perform localized coding tasks	Overlaps with ASE in coding; localized coding alone does not cover the full lifecycle
ASE (This White Paper)	A full-lifecycle engineering paradigm encompassing specifications, execution, verification, governance, and accountability	—
General-Purpose AI Agents	Task automation across domains	Shares techniques for tool use and execution loops with ASE, but is not defined around software engineering artifacts and quality
MLOps / LLMOps	Development, evaluation, deployment, and monitoring of models and LLM applications	Overlaps with ASE in models, evaluation, and operational governance; ASE also covers specifications, artifacts, and the accountability feedback loop for software production
DevOps / Platform Engineering	Delivery workflow automation and platform capability development	Provides the foundation for implementing ASE; ASE strengthens agent-oriented specifications, execution controls, and evidence requirements

Research literature also distinguishes software engineering for humans (SE for Humans) from software engineering for agents (SE for Agents), and discusses human command environments and agent execution environments. [1] That distinction concerns whom an engineering environment serves, rather than the two engineering targets defined here. Nevertheless, its premise that both people and agents need dedicated engineering support is useful.

3.4 Scope and Non-Goals of the First Release

This white paper covers ASE's definition, overall framework, capability architecture, full lifecycle, and three core capabilities for the first release: specification development, testing and evaluation, and verification and trustworthiness assurance. The first release takes software production (coding) as its initial use case. Extending the methodology to business management systems and other business domains is left for later releases.

This white paper does not cover specific algorithms or model architectures; tool or vendor selection; a complete reference implementation; multi-agent collaboration protocols; detailed deployment or runtime architectures; or specific provisions of standards and certification systems. The overall framework accommodates these topics, but their technical details are reserved for later releases.

Non-goals: This white paper does not claim that any particular product or technical approach is superior, rank model capabilities, or forecast market size. All external data cited is used solely to illustrate trends and mechanisms, not as a basis for procurement or investment decisions.

Chapter 4 ASE Reference Framework, Capability Elements, and Core Capabilities

This chapter answers: What layers, capabilities, and control mechanisms make up the new paradigm?

4.1 The Overall Loop from Requirements to Evidence

ASE operates as a closed loop. Human business intent is compiled into executable specifications, which direct agents as they plan and execute in a controlled environment. Execution produces artifacts and process traces. Testing, evaluation, and verification examine both the artifacts and the process and generate evidence. That evidence supports acceptance decisions, and the acceptance results and operational feedback flow back into specifications and knowledge to drive the next cycle of evolution.

The defining characteristic of this loop is not the degree of automation, but the presence of explicit inputs, outputs, and decision criteria at every step.

4.2 Four-Layer Reference Framework

The four-layer reference framework proposed in this white paper describes responsibilities and capabilities. It does not represent four independent technology stacks or a one-way serial pipeline:

Layer 1: Intent and Specification. This layer holds business objectives, value judgments, accountability boundaries, and their executable representations. Its outputs include requirements, task contracts, constraints, and the Definition of Done. Humans lead the confirmation of intent, while agents may assist with clarification and structuring.

Layer 2: Agent Execution. At this layer, agents use a model, a runtime control harness, tools, and context and operate within defined permission constraints as they iterate through observation, decision, action, and verification. Execution must have budgets and termination conditions, be stoppable, recover from exceptions, and preserve the traceability of critical execution traces.

Layer 3: Software Lifecycle. This layer organizes the artifacts, activities, and stage gates for definition, design, build, verification, release and operations, evolution, and retirement, maintaining consistency and enabling handoffs across stages.

Layer 4: Assurance and Governance. Testing, evaluation, verification, risk controls, and audits span the other layers to produce quality claims, evidence, and release decisions and to enforce organizational and individual accountability.

Specifications, actions, artifacts, evidence, and feedback connect the four layers. Intent constrains execution; execution and lifecycle activities produce artifacts; and the Assurance and Governance layer independently checks

each stage and feeds the results back into specifications and the production system. Evidence supports conclusions within its stated scope; it does not automatically constitute a mathematical proof.

4.3 Six Capability Domains and Their Key Elements

Table 1. Six Capability Domains, Key Elements, and Primary Outputs

Capability Domain	Key Elements	Primary Outputs
1. Requirements and Specification Engineering	Objectives, constraints, acceptance criteria, nonfunctional requirements, policies, and traceability relationships	Executable specifications, task contracts, and Definition of Done
2. Context and Knowledge Engineering	Context, sessions, state, memory, knowledge, versioning, and source attribution	Usable, traceable task context
3. Agent Execution Engineering	Models, harnesses, tools, permissions, sandboxes, execution loops, budgets and termination, and exception recovery	A controlled, stoppable execution process that preserves the traceability of critical execution traces
4. Software Artifact Engineering	Code, configuration, data, tests, documentation, dependencies, and the software supply chain	Consistent, buildable, maintainable, versioned artifacts
5. Testing, Verification, and Evidence	Testing, evaluation, test oracles, static, dynamic, and formal verification, and the evidence chain	Reviewable quality conclusions and a basis for acceptance
6. Runtime Governance and Evolution	Release gates, observability, incident response, security governance, feedback, and retirement	An operable, auditable system that supports continuous improvement

The six capability domains span the four layers and must not be treated as six nonoverlapping sequential steps. An organization may determine the depth of each capability based on risk, but it must identify who provides each capability and which artifacts and evidence are used for acceptance.

4.4 Three Core Capabilities for the Initial Release

Within the six capability domains, this white paper focuses on three cross-cutting core capabilities. Testing and evaluation, together with verification and trustworthiness assurance, support the fifth capability domain:

Capability 1: Specification Development. This capability answers what an agent must do, what it must not do, and what constitutes completion. It is the control center for the entire loop.

Capability 2: Testing and Evaluation. This capability determines how to judge reliably whether the software and its production process meet requirements. It is a major constraint on software delivery capability.

Capability 3: Verification and Trustworthiness Assurance. This capability determines how to prove or continually confirm that critical boundaries have not been crossed. It determines the risk levels of the application scenarios in which ASE can be used.

Together, the three capabilities form a minimum quality loop: specification, execution, testing and verification, evidence-based acceptance, and specification revision. The required context management, controlled execution, and runtime governance should be established in parallel. More complex capabilities, such as multi-agent systems, may be added incrementally as needed.

4.5 Five Engineering Principles

- 1. Specification-Driven Principle:** Define objectives, boundaries, and acceptance conditions before execution. An agent must not guess at unresolved critical requirements.
- 2. Risk-Constrained Principle:** Limit impact through least privilege, isolation, budgets, stop conditions, and rollback or compensating actions. Irreversible, high-risk actions should require prior approval and, when necessary, must not be executed autonomously.
- 3. Closed-Loop Evidence Principle:** Establish traceable artifacts, independent determinations, quality claims, and approval records for critical requirements. Acceptance must not rely only on the executor's self-assessment.
- 4. Human Accountability Principle:** Define the responsibilities of the organization and authorized individuals for authorization, risk acceptance, and exception handling. Automation does not eliminate accountability.
- 5. Continuous Evolution Principle:** Feed failures, operational events, and evaluation results back into specifications, tests, and governance policies, and adjust the scope of autonomy based on evidence.

4.6 Responsibility Boundaries: Humans, Agents, and the Platform

Responsibilities are divided among three parties. Humans define objectives and remain accountable. Agents interpret tasks and orchestrate the work. Humans, agents, and the platform jointly develop and use verification and rollback capabilities. No party may declare completion on its own. The specific boundaries are as follows:

Party	Responsibilities	Should Not Do
Human	Define objectives and acceptance criteria; establish constraints and permissions; accept results and authorize release based on risk; remain accountable to external parties	Use manual inspection as a substitute for all automated checks, or review every line of every output without regard to risk; critical implementations still require human review when appropriate
Agent	Plan and execute within defined boundaries; produce artifacts and process evidence; stop and request clarification when a boundary is crossed or uncertainty arises	Set value objectives independently; relax acceptance criteria independently; decide risk acceptance independently
Platform	Enforce permissions and gates; execute preapproved automated release policies; record evidence; provide stopping, recovery, and limited replay capabilities	Set objectives independently, relax risk thresholds, execute unapproved automated release policies, or assume accountability on behalf of the organization or its personnel



Figure 2. ASE Four-Layer Reference Framework and Six Capability Domains

Chapter 5 Full Lifecycle, Critical Artifacts, and the Quality Loop

This chapter answers: How does ASE cover the entire process from business intent through operational evolution?

5.1 Six-Stage Lifecycle

This white paper divides the ASE lifecycle into six stages:

- 1. Definition:** Define the business objectives, scope, accountability, and risks. Compile intent into specifications, task contracts, and decidable acceptance conditions.
- 2. Design:** Define the architecture, interfaces, context boundaries, and permission boundaries, and establish the verification strategy, release strategy, and rollback or compensation plan.
- 3. Build:** Agents plan, implement, test, and repair in a controlled environment, producing versioned artifacts, a dependency inventory, and change traces.
- 4. Verification:** Independently examine artifacts, execution traces, and software delivery capability; consolidate testing, evaluation, and analysis results and any formal evidence required; and address residual risk.
- 5. Release and Operations:** Authorize release according to approved policies and risk levels, and perform staged rollout, monitoring, incident response, and rollback or compensating actions.
- 6. Evolution and Retirement:** Feed failures and operational feedback back into specifications and tests, and manage knowledge updates, system migration, permission revocation, data disposition, and service retirement.

5.2 Minimum Artifacts and Stage Gates for Each Stage

Table 2. Minimum Artifacts and Stage Gates Across the Six-Stage Lifecycle

Stage	Minimum Artifacts	Stage Gate
1. Definition	Objectives and risk statement, specifications, task contract, and acceptance conditions	Scope and accountability are clear; critical requirements are decidable
2. Design	Architecture and interfaces, permission boundaries, verification plan, and rollback or compensation plan	Boundaries are enforceable; the design can be verified; approval has been obtained for irreversible risks
3. Build	Versioned artifacts, dependency inventory, execution traces, and change records	Artifacts can be built; changes are controlled; critical execution traces are traceable
4. Verification	Testing and evaluation reports, verification evidence, and a residual-risk register	Critical criteria are met; independent review is complete; authorized personnel have approved how residual risk will be handled

Stage	Minimum Artifacts	Stage Gate
5. Release and Operations	Release authorization records, operating policies, observability records, and incident records	Release is authorized according to risk; monitoring and loss-containment measures are available
6. Evolution and Retirement	Failure examples, specification and knowledge updates, and migration and retirement records	The feedback loop is closed; migration and the disposition of data and permissions have been confirmed by the accountable person

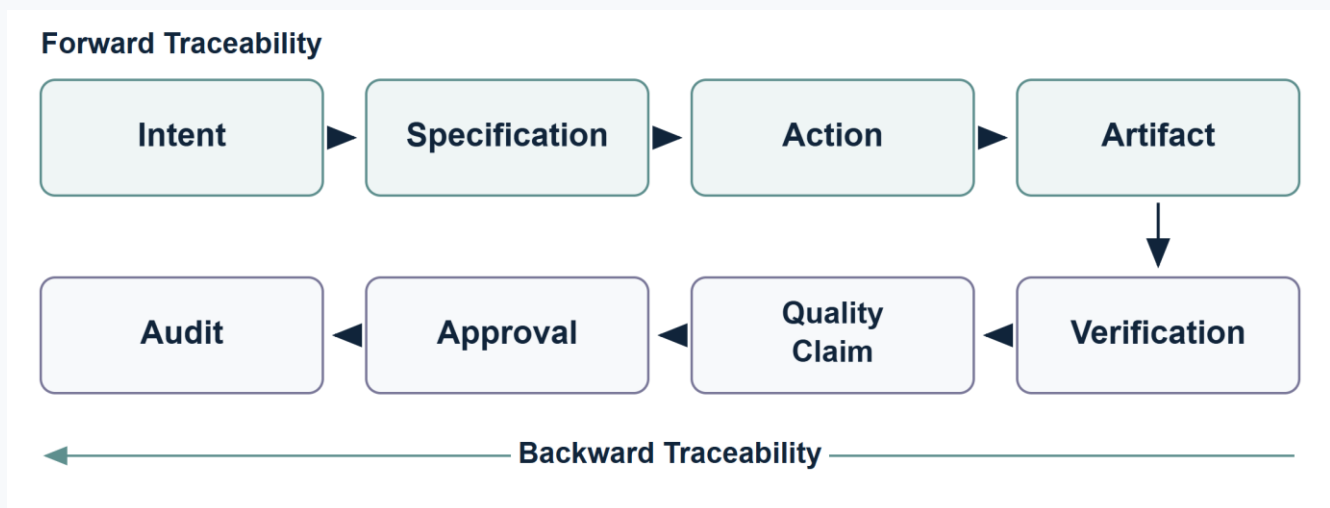
A minimum artifact is reviewable evidence that each stage should have; it need not be a separate document. Organizations may combine artifacts. For any item deemed not applicable, they should document the rationale, risk, and approver. They must not silently omit a critical control.

5.3 Evidence Chain Across the Lifecycle

The evidence chain connects requirements to conclusions. Its core requirement is bidirectional traceability:

- **Forward traceability:** Starting from any requirement, reviewers can find its acceptance criteria, corresponding test cases, execution results, related code changes, and the person responsible for acceptance.
- **Reverse traceability:** Starting from any code change, reviewers can trace it back to the task that triggered it, the corresponding requirement, the governing specification clauses, and the verification evidence.

The evidence chain can be represented as Intent → Specification → Action → Artifact → Verification → Claim → Approval → Audit. Each item of evidence should identify, at a minimum, its source, version, time, accountable person, scope of applicability, and related requirements, and should state the limits of the quality claim and any residual risk. Critical traces include tool calls, permission use, and outcomes; they need not expose the model's internal reasoning. Evidence should have integrity protection and access controls, be retained for the period required by applicable regulations and organizational policy, and be updated with each artifact version.



5.4 Quality Objectives

ASE quality objectives should cover correctness, security, reliability, controllability, traceability, and evolvability across two dimensions: the product and the production process.

Product dimension: functional correctness, performance and resource efficiency, reliability and availability, security, maintainability, and compliance.

Process dimension:

- **Reproducibility and replayability:** Hold controllable versions and inputs constant, retain records sufficient to review critical actions and determinations, and report the statistical stability of nondeterministic conclusions.
- **Explainability and controllability:** Record the authorization basis and source for critical actions, and provide mechanisms to stop execution, restrict permissions, transfer control to a human, and handle exceptions.
- **Auditability:** An authorized third party can use the evidence to reconstruct critical actions and the approval chain. Any evidence gap must be disclosed explicitly.
- **Evolvability and recoverability:** Released changes have rollback, compensation, migration, or loss-containment plans proportionate to risk, and those plans are maintained through retirement.

In nondeterministic systems, reproducibility does not mean byte-for-byte identity. It means statistical stability under controlled conditions and limited replay of observable processes. External dependencies that cannot be replayed must be identified explicitly.

5.5 Human-Agent Accountability Gates in the Lifecycle

Human-agent accountability gates enforce authorization, risk acceptance, and human takeover. The higher the risk, the lower the reversibility, and the broader the impact, the earlier the gate should occur. At a minimum, accountability gates cover specification confirmation, release authorization, and handling of operational exceptions. Low-risk matters may be decided automatically under an approved policy and be subject to spot checks; high-risk matters and exceptions require explicit approval by an authorized person. Continuous monitoring may trigger human review, but a human signature is not required after every low-risk change.

An accountability gate may take the form of a policy authorization record, an approval, or a mandatory confirmation. It must identify the accountable person, applicable scope, version, and timestamp. Authorization should be reevaluated when the risk level, permissions, or verification evidence changes materially.

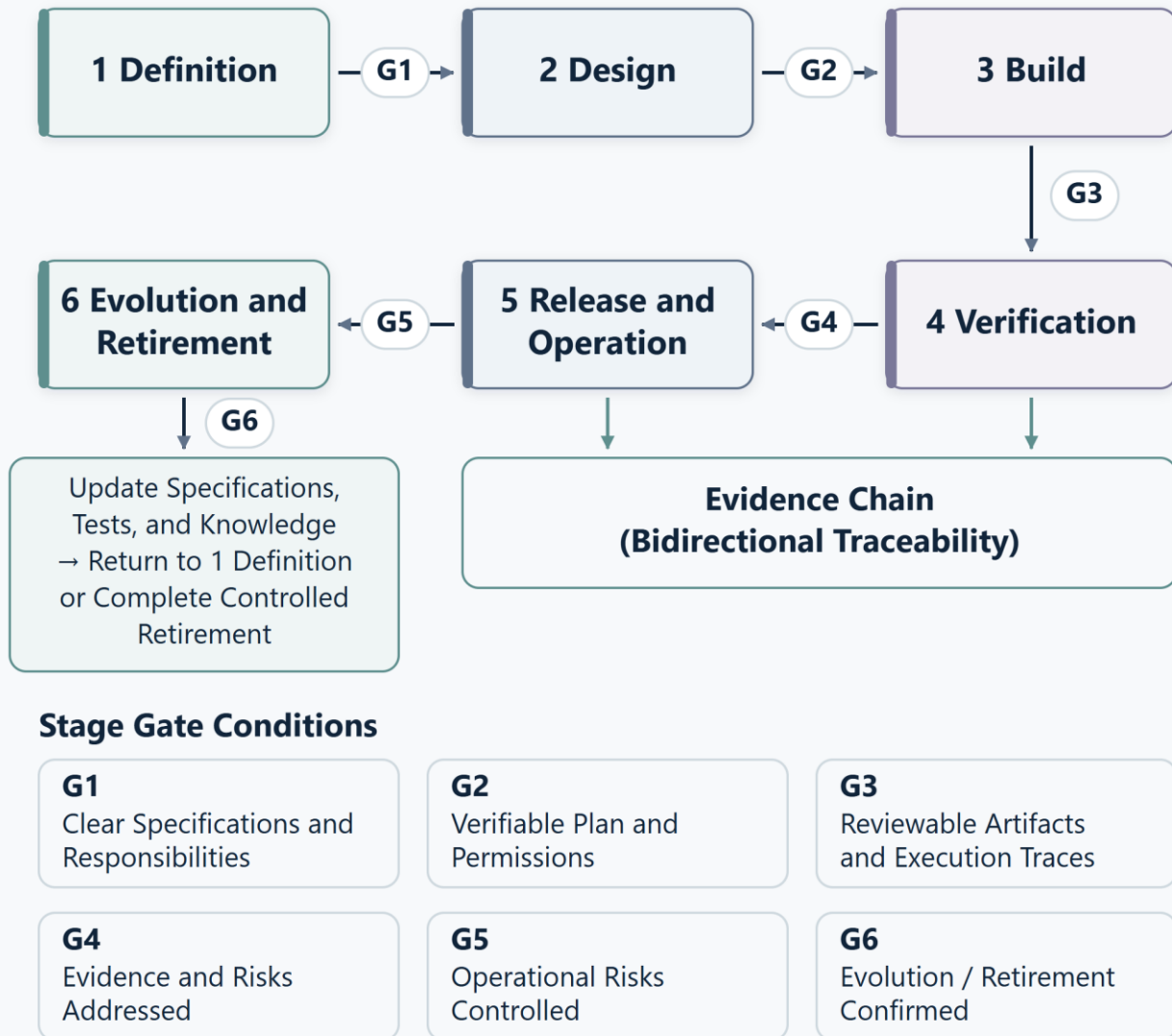


Figure 3. Six-Stage Lifecycle, Stage Gates, and Evidence Loop

Part III How to Build the Three Core Capabilities

Chapter 6 Specification Development: Converting Human Intent into Engineering Contracts Agents Can Execute

This chapter answers: How can agents know what to do, what not to do, and what counts as complete?

6.1 Upgrading Specifications from Descriptive Documents to Runtime Control Objects

Traditional software engineering expresses requirements through documents, contracts, and communication among people. In ASE, an agent may not proactively detect ambiguity or ask the right follow-up questions. Without an explicit clarification mechanism, it may fill in ambiguous requirements on its own. Specifications must therefore become runtime control objects that are continually referenced, checked, and traced during development, testing, release, and operations.

Specifications must be understandable to people, readable by machines, usable to check results, and traceable across versions. Not every natural-language requirement must be fully formalized. However, critical provisions such as safety boundaries, interfaces, and permissions should be expressed as structured contracts, policies, or assertions. Open-ended requirements should define a clear human or combined decision procedure.

6.2 Four Minimum Categories of Specifications

This white paper recommends that every task delegated to an agent include, at a minimum, specifications in the following four categories:

Category 1: Objectives and Acceptance Specifications. State what must be achieved and how it will be accepted, including business objectives, scope, functional requirements, priorities, and success criteria.

Category 2: Behavior and Quality Specifications. State the required behavior and quality boundaries, including security, privacy, performance, reliability, resource budgets, and maintainability requirements.

Category 3: Interface and Tool Specifications. Define permitted inputs and outputs, interface preconditions and postconditions, tool parameters, authorization scope, external side effects, idempotency, and error handling.

Category 4: Assurance and Accountability Specifications. Define testing and verification methods, the Definition of Done, evidence requirements, the decision maker and approver, and the conditions for stopping, clarification, human takeover, rollback, or compensation.

Together, the four categories form the task contract. The Definition of Done cannot simply state that the task is complete; it must identify the artifacts, evidence, and approval conditions that can be checked. If the contract is incomplete, it should be clarified before execution.

6.3 The Requirements Compilation Process

Requirements compilation progressively transforms business intent, rules, constraints, and examples expressed in natural language into executable, verifiable, and traceable software artifacts [18]. Drawing on the way a traditional compiler converts a high-level language into a machine-executable program, it establishes a systematic transformation path from human intent to a software system rather than merely producing a structured requirements document.

Requirements compilation can be divided into the following stages:

1. **Intent Capture and Interactive Clarification.** Gather requirements from natural-language descriptions, business documents, meeting records, existing systems, user interaction examples, prototypes, and other sources. When ambiguity, omissions, conflicts, or implicit assumptions arise, the system should actively engage users through questions, prototype feedback, example confirmation, and similar methods to clarify the actual intent incrementally.
2. **Requirements Modeling and Intermediate Representation.** Compile natural-language requirements into a machine-processable requirements model that covers entities and data relationships, business rules, state transitions, user scenarios, interface contracts, quality constraints, permissions, and cross-requirement dependencies. Like the intermediate representation (IR) in a traditional compiler, this representation is the core medium that connects natural-language intent to downstream software generation.
3. **Requirements Decomposition and Constraint Propagation.** Decompose high-level business objectives into scenarios, tasks, interfaces, data, behavior, and quality constraints, while maintaining dependencies across levels. When an upstream requirement changes, the corresponding constraints can propagate to the affected software artifacts.
4. **Software Artifact Generation.** Compile the unified requirements representation into, or use it to drive the generation of, database schemas, APIs, user interfaces, test cases, configuration, permission rules, and even implementation code. Requirements thereby become a direct input to software construction rather than a reference document for development activities.
5. **Executable Verification and Requirements Debugging.** Compile requirements into an executable set of acceptance conditions, tests, assertions, constraints, and check programs to verify whether the generated software conforms to user intent. When a test fails or behavior deviates from expectations, the system should

support backtracking to the relevant requirement, constraint, or requirements conflict so that the requirements themselves can be debugged.

6. Incremental Compilation and Bidirectional Evolution. When users change requirements, the system should not regenerate the entire system. It should analyze the impact of the change and recompile only the affected design elements, interfaces, code, and tests. Software implementation and operational feedback should also map back to the requirements model, creating a continuously evolving loop of requirements, software, and verification results.

The outputs of requirements compilation are intermediate representations, constraints, and executable artifacts that connect user intent to software implementation, not simply a better-structured requirements document. The goal is to establish the following path:



Together, these elements form a complete compilation pipeline.

Requirements compilation addresses a more fundamental software engineering problem: how to make human requirements clear while also turning them into computational objects that directly drive software generation, verification, and continuous evolution.

6.4 Specification Quality and Conflict Resolution

The following measures can be used to assess specification quality:

- **Coverage:** Does every requirement have a corresponding acceptance condition?
- **Decidability:** Does every acceptance condition have a predefined, reviewable decision procedure that is automated, manual, or a combination of both?
- **Consistency:** Do any specification items conflict with one another?
- **Traceability:** Can every specification be traced to its business source and forward to its implementation and evidence?
- **Feasibility and Priority:** Can the requirement be implemented within the available budget, technology, and permissions? Are priorities and authority for resolving conflicts clearly assigned?

Conflict resolution follows three rules. First, constraints take precedence over objectives: nonnegotiable safety, compliance, privacy, and other provisions must not be weakened to meet a functional objective. Second, explicit terms take precedence over implicit ones: conflicting provisions must be identified, and an agent must not choose among them on its own. Third, escalation takes precedence over guessing: conflicts that cannot be resolved automatically must be referred to a human decision maker, and the agent must not decide them independently.

6.5 Specification Changes and Impact Analysis

In ASE, a specification change can have a much greater impact than in a traditional model: a change in wording may alter an agent's entire execution path. Organizations must therefore establish the following change disciplines:

- **Every specification change must be versioned, with the person making the change, the time, the reason, and the scope of impact recorded.**
- **Each specification change must be analyzed for its impact on the architecture, code, tests, proof assumptions, authorizations, and operating policies.** Evidence invalidated by the change must be identified, and affected verification must be rerun. Reuse of unaffected evidence must have a documented basis.
- **For high-risk specification provisions, organizations should require change approval to prevent in-place modification during execution.**
- Specification clarification requests raised by an agent during execution should be logged and measured as signals of specification defects and used to improve specification quality.

6.6 Minimum Practice Template

Organizations can begin with the following minimum set without waiting for a complete platform:

1. **Task Contract Template:** For each task, identify the objectives and scope; inputs; behavior and quality constraints; interfaces, tools, and permissions; Definition of Done and evidence; accountable person and escalation conditions; and rollback or compensation method.
2. **Acceptance Conditions Checklist:** Provide one row for each requirement, including a unique identifier, decision method, current status, and evidence location.
3. **Specification Check Script:** Before a task starts, automatically check the specification for completeness and prevent the task from starting if a required item is missing.
4. **Specification Change Log:** Record the impact scope of each change and the results of reverification.
5. **Clarification Request Register:** Consolidate clarification requests raised by agents and use them as a feedback indicator of specification quality.

Minimum task contract example (illustrative, not a universal threshold): R-01 requires pagination to be added to an existing order-query interface while maintaining compatibility with legacy calls. Inputs include interface

specification v1, de-identified data, and a test baseline. Only the query module and tests approved through review may be modified; access to production credentials, writes to production data, and database schema changes are prohibited. The Definition of Done requires the compatibility and pagination contract tests to pass, latency under the specified load to meet an approved threshold, and an independent review to be completed. Evidence includes the commit identifier, test report, environment version, and authorization trace. If a requirements conflict, unauthorized request, or budget overrun occurs, execution stops and the interface owner decides how to proceed.

Traceability example: R-01, "Remain compatible when no pagination parameter is supplied" → T-01, legacy-client contract test → V-01, test report containing the commit and environment versions → C-01, "Compatible within the declared interface scope" → A-01, accountable person's approval or preauthorized gate record. A failure should trigger revision of the specification or implementation; the contract must not be relaxed autonomously.

Capability summary: Specification development addresses the risks that intent will be misunderstood and that objectives and boundaries will drift during execution. Inputs include business intent, constraints, organizational policies, and existing artifacts. Outputs include executable specifications, task contracts, acceptance conditions, and traceability relationships. The requirements owner confirms critical interpretations, and specification changes are subject to authorized approval.

Chapter 7 Testing and Evaluation: Examining Outcomes, Processes, and Software Delivery Capability

This chapter addresses the following question: How can we reliably determine whether the software and its production process meet requirements when agents are nondeterministic?

7.1 Define the Test Objects

ASE should examine software outcomes, execution processes, and software delivery capability. These fall into five categories of test objects:

Object 1: Models and agents. Use representative tasks to assess their planning, tool use, instruction following, and ability to refuse actions outside authorized boundaries, and record the operating conditions and scope of applicability.

Object 2: Tools and components. Use unit, contract, permission, and fault tests to examine the behavior of interfaces, sandboxes, state stores, and dependent components.

Object 3: Execution traces. Examine task progress, use of authorization, state transitions, stop conditions, and exception recovery, and identify unauthorized test modifications or side effects.

Object 4: Software artifacts and the environment's end state. Examine code, configuration, data, and observable post-deployment state rather than relying only on the agent's completion message.

Object 5: Business outcomes and software delivery capability. Determine whether user goals have been achieved, and measure success rates, rework, human intervention, cost, latency, and severe failures to assess whether the system can deliver sustainably.

The five categories provide different kinds of evidence. Correct artifacts do not offset unauthorized behavior; an agent's self-reported completion does not mean that the environment has reached the intended end state; and a single success does not demonstrate sustainable software delivery capability. [16]

7.2 Layered Testing and Evaluation Methods

The following four methods should be used in combination rather than treated as strictly mutually exclusive testing layers:

1. Unit and contract testing: Verify functions, modules, and interface contracts. Agents can generate many such tests, but the tests must be driven by specifications, and the validity of the tests themselves must be checked.
2. Integration and system testing: Verify collaboration among modules and end-to-end behavior. These tests are a primary means of finding latent regressions.
3. Property and invariant testing: Check properties such as idempotency, conservation, and permissions under generated or selected inputs and states, and use fault injection, adversarial examples, and long-horizon scenarios to search for counterexamples. Passing these tests supports only the scope covered; it does not prove that a property always holds.
4. Behavioral and capability evaluation: Run representative task sets repeatedly, and combine regression replay, shadow execution, and staged-release experiments to evaluate system performance. Prespecified safety and capability thresholds can serve as release gates. Trend analysis supports long-term improvement but cannot replace mandatory gates.

Property testing can systematically generate boundary cases, but it must still complement explicit test cases, integration tests, and end-to-end tests. Methods should be selected according to risk, decidability, and coverage objectives. The fact that code is generated by a probabilistic model does not necessarily reduce the value of conventional testing.

7.3 Test Oracles: Who Decides What Is "Correct"?

A test oracle provides the basis for deciding whether the behavior of the object under test is correct. In ASE, the source of each oracle must be explicitly designed. Each of the following five sources has limitations:

Test Oracle	Applicable Scenarios	Primary Limitations
Assertions derived from specifications	Functional and contract verification	Depend on specification quality; assertions are ineffective when specifications are ambiguous
Existing implementation as a reference	Refactoring and migration	The existing implementation may contain defects that will be inherited
Properties and invariants	Concurrency, distributed systems, and state machines	Require abstraction expertise; the properties themselves may be incomplete
Human annotation and expert review	Subjective quality, user experience, and compliance	Expensive, not scalable, and subject to differences in judgment
Statistical and distributional comparison	Nondeterministic output and generative quality	Can provide confidence estimates, but not deterministic conclusions

Core discipline: An agent's self-evaluation must never be the sole basis for release. Prefer deterministic judgments from compilers, schemas, rules, tests, and property assertions. For open-ended tasks, combine reference implementations, differential results, independent model scoring, and expert sampling, and calibrate the scorer. Producers and evaluators should be separated in their permissions, data, and evaluation processes; merely changing the model does not establish independence. When no reliable test oracle is available, narrow the task scope, lower the level of autonomy, or escalate to human judgment. [16]

7.4 Statistical Discipline and Reproducibility

The capability and reliability of a nondeterministic system cannot be inferred from a single success. Experiments under fixed conditions are used to reproduce failures, whereas repeated-sampling experiments estimate variability and failure rates; the two should be distinguished. The statistical plan must specify the estimand, task-sampling method, and repeated-run procedure.

- **Repeated runs:** For nondeterministic capabilities, conduct repeated trials over the task distribution and report the number of runs, success rate, variability, confidence intervals, and tail failures, together with sample dependence, retry count, and budget. Interval estimates must account for tasks and other correlated units; repeated attempts on the same task must not be treated as new independent tasks.
- **Fixed, controlled conditions:** Record the versions of the model, harness, prompts, tools, data, environment, parameters, specifications, and evaluation set. Fix the source of randomness when it is controllable, but recognize that a random seed does not ensure complete reproducibility across platforms or service versions.
- **Prevent contamination:** Separate development, regression, and held-out evaluation sets; protect hidden test cases; record exposure of public tasks; and update them regularly. When the training data of a closed-source model cannot be verified, disclose the uncertainty about contamination rather than claiming complete isolation.

- **Isolation and protection:** Keep offline evaluation separate from development and production environments, and protect hidden tests and evaluators. Controlled production experiments, such as shadow and staged-release trials, require separate authorization as well as data boundaries, monitoring, and loss-containment measures.
- **Distinguish trends from hard limits:** Use mean capability for trend analysis, but immediately block and investigate severe safety failures, unauthorized actions, or data leakage. An average success rate cannot replace an assessment of the upper bound on severe-failure risk.

Evaluation conclusions should state the environment, task distribution, and limitations. [3][16] Quality, cost, latency, human intervention, rollback, and severe failures should also be recorded to avoid optimizing only for average success rate. The absence of observed severe failures in a finite sample does not mean that the true risk is zero.

7.5 Evidence-Driven Release Gates

A release gate should consist of a set of automatically checkable conditions, including the following:

Gate Item	Decision Criteria
Specification conformance	Critical requirements meet their thresholds; unmet noncritical requirements have explicit, authorized residual-risk treatment
Test pass status	Applicable tests pass; nondeterministic capabilities meet prespecified statistical thresholds
Coverage	Changed areas meet organization-defined coverage requirements
Security and compliance	Applicable security, dependency, license, and sensitive-data rules pass, with no unresolved severe issues
Process compliance	Critical traces are complete, with no unauthorized actions, leakage of hidden evaluations, or unauthorized weakening of tests
Recovery and loss containment	Rollback, compensating actions, or loss-containment measures match the risk, and required drills have been completed
Authorized release	Low-risk changes are released under preauthorized policies; high-risk changes and exceptions are approved by designated personnel

Release gates must default to deny. Exceptions require explicit authorization and must record the rationale, accountable owner, residual risk, expiration, and compensating measures. No exception may release conduct prohibited by law or waive nonwaivable safety limits. When a gate fails, the work must return to the specification, implementation, or verification stage for correction; unlimited retries may not be used to select an accidental success.

7.6 Minimum Practice Template

1. Testing baseline: Select unit, contract, integration, end-to-end, and property tests according to task risk; explain any items that do not apply; and incorporate the selected tests into the release gate.

2. Behavioral evaluation set: Begin with a pilot using a representative set of normal, boundary, and failure tasks, then expand it according to risk coverage and statistical requirements. A small task set can provide a starting point, but it cannot demonstrate system safety with respect to low-probability, severe failures.
3. Anti-tampering checks: Detect weakened assertions, silenced errors, and access to hidden evaluations. Permit test maintenance approved through independent review, but prohibit agents from concealing defects by changing the decision criteria.
4. Repeated-run report template: Record versions, tasks, number of trials, success rate and confidence intervals, failure categories, cost, latency, human intervention, and severe failures.
5. Gate checklist: A pre-release checklist that supports check-off, trace retention, and audit.

Example minimum evaluation matrix (object / method / test oracle / evidence): model and agent / repeated runs on representative tasks / approved scoring rules / trial-distribution report; tool component / contract testing and fault injection / interface assertions / component test records; trace / adversarial scenarios and replay / permission and stop policies / action audit; artifact and end state / integration and end-to-end testing / acceptance assertions / versioned results; business outcome / shadow or staged-release experiment / business objectives and expert confirmation / operational metrics and approval records.

Example gate: For the specified release candidate, environment, and task scope, all critical contracts must pass, no severe safety failure may remain unresolved, and regression metrics must meet preapproved thresholds. An independent evaluator checks the completeness of the evidence, and the risk owner approves acceptable residual risk. The evidence, scope of applicability, responsible parties, and pass/fail status are all retained with the release record.

Capability summary: Testing and evaluation address the risk that incorrect outcomes, out-of-bounds processes, or unsustainable software delivery capability will nevertheless be judged deliverable. Inputs include specifications and acceptance conditions, artifacts, execution traces, evaluation sets, and test oracles. Outputs include testing and evaluation reports, statistical reports, and gate records. An independent evaluator checks the evidence, while the risk owner approves residual risk and is accountable for the release decision.

Chapter 8 Verification and Trustworthiness

Assurance: Constraining High-Risk Behavior with Layered Evidence

This chapter addresses the following question: Beyond testing, how can we prove or continuously confirm that agents and software have not crossed critical boundaries?

8.1 Distinguish "Building the Right Thing" from "Building the Thing Right"

Quality assurance addresses two distinct questions:

- **"Building the thing right" (Verification):** Does the implementation satisfy the specification—that is, was it implemented according to the specification?
- **"Building the right thing" (Validation):** Do the specification and final software meet real user needs and the intended use? This means confirming both that the specification is worth implementing and that the delivered result solves the business problem.

Testing, reviews, user trials, and analysis can provide evidence for verification and validation in different ways. Even if an erroneous specification is implemented and verified correctly, the delivered product may still be wrong. The alignment between the specification and business intent must therefore be validated continuously, and operational results must be confirmed as fit for their intended use. An agent may either misunderstand a specification or execute a flawed one; it cannot be assumed to implement requirements faithfully and correctly.

8.2 Risk Levels Determine the Strength of Assurance

The strength of assurance should match the level of risk. Low-risk changes should not incur excessive cost, while high-risk changes should not rely on testing alone.

Table 3. Risk Levels, Verification Methods, Evidence Requirements, and Human Gates

Risk Level	Typical Scenarios	Primary Verification Methods	Evidence Requirements	Human Gate
R1 Low	Internal tools, documentation, and styling changes	Unit tests and automated gates	Applicable checks, policy authorization, and records of critical actions	Preauthorized automatic release, with spot checks and escalation of anomalies
R2 Medium	Internal business functionality and non-core services	Unit, integration, and property testing; static analysis	Tests, independent review, fault injection, and trace evidence	Review by a designated owner; staged rollout and rollback according to risk

Risk Level	Typical Scenarios	Primary Verification Methods	Evidence Requirements	Human Gate
R3 High	Customer-facing core functionality, data processing, and permission changes	Contract checks, policy and data-flow analysis, and runtime monitoring; formalization of critical properties as needed	Complete evidence chain and independent review	Mandatory human approval for release
R4 Very High	Funds and payments, human safety, critical infrastructure, and irreversible operations	Modeling or proof of critical properties, independent verification and validation, and constrained execution	Applicable formal evidence and assumptions, independent review, and exercise records	Two-person or multilevel approval; staged rollout and loss containment; full autonomy prohibited when necessary

Table 3 presents an illustrative classification, not a legal one, and scenario names do not automatically determine risk levels. Each organization should classify risk based on the combined scope of impact, data sensitivity, permissions, reversibility, cost of failure, and applicable laws and regulations, then encode the corresponding permissions and gates. An agent may identify escalation signals under approved rules, but it may not lower a risk level or relax a boundary on its own.

8.3 Layered Verification Mechanisms

The following four complementary types of checks should be combined and reviewed throughout by independent personnel or experts:

- 1. Type and schema layer:** Use type systems, schemas, and interface contracts to eliminate structural errors. Critical interfaces should receive priority, and structural validity must be distinguished from semantic correctness.
- 2. Static analysis layer:** Static analysis, symbolic execution, and abstract interpretation can be used to find defects or, within the capabilities and assumptions of the tools, prove specific properties. These methods may produce false positives and are subject to incompleteness and path limitations. Reports must distinguish proved results, counterexamples, and unresolved outcomes.
- 3. Dynamic verification layer:** Use testing, fuzzing, property testing, and runtime assertions to check execution behavior. For actions that pass through controlled tool channels and are observable and decidable, combine sandbox, permission, state, and budget monitoring to intercept them before side effects occur whenever possible, while addressing monitoring blind spots and failure modes.
- 4. Formal verification layer:** Produce machine-checkable proofs of critical properties over explicit models and assumptions, or perform model checking within declared state or search bounds.

The cost and strength of assurance of these mechanisms do not increase in a strict sequence. Static analysis and runtime verification may themselves use formal methods. The mechanisms should be combined according to risk, coverage, and feasibility rather than using any one method to replace other evidence.

8.4 Priorities for Formalization

The cost of formal methods varies by object, property, and tool. An entire agentic system should not be required to undergo complete formalization in a single effort. Priorities should include:

- **Concurrent and distributed protocols:** Use state machines and temporal properties to check for deadlocks, consistency, and liveness. Industrial practice has demonstrated the value of formal modeling in finding defects caused by complex interleavings, but it does not replace implementation testing. [4][15]
- **Security and permission boundaries:** Unauthorized access, information flow, authorization preconditions, and temporal requirements such as "execution may occur only after approval."
- **Agent workflows and state machines:** Determine whether execution loops or multi-agent collaboration necessarily reach an allowed terminal state; whether they can deadlock, run out of control, or bypass mandatory gates; and whether every critical action necessarily produces a correlated audit record that cannot be omitted.
- **Resources, termination, and side effects:** Budget ceilings, stop conditions, conservation of funds, idempotency, compensating actions, and conditions that trigger irreversible operations.
- **Compilers, runtimes, and critical infrastructure:** Select critical modules for verification according to their impact surface, drawing on the bounded-assurance practices of seL4 and CompCert. [5][6]
- **Consistency among specifications, evidence, and implementation:** Whether critical clauses have been implemented, and whether the evidence required for approval is complete and version-matched.

User experience and actual user needs still require user validation and expert judgment. Formal methods can also be applied to formalizable interface, state, or accessibility constraints; such constraints should not be categorically excluded merely because they concern the "interface."

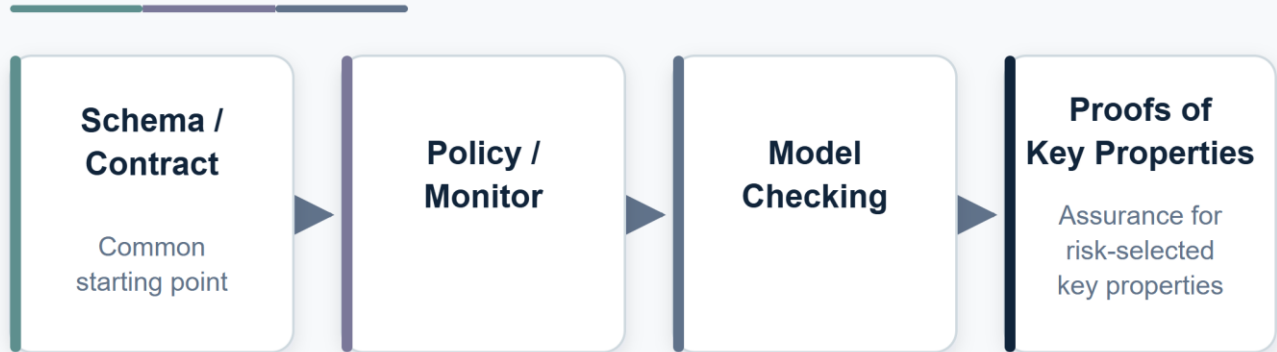
8.5 Available Formal Methods and a Progressive Adoption Path

Available formal methods include:

- **Schema and contract validation:** Use these as a progressive starting point for checking structure and local constraints. Passing validation does not prove the correctness of the system as a whole.
- **Model checking:** Exhaustively check temporal properties over finite-state models; this is well suited to protocols and state machines.
- **Theorem proving:** Construct checkable proofs in a formal system. The scope of assurance depends on the properties, assumptions, correspondence to the implementation, and trusted computing base; cost depends on the specific task.
- **Abstract interpretation and static proof:** Automatically derive program properties, with comparatively good scalability.

- **Runtime verification:** Compile critical properties into monitors that continuously check them during execution and trigger a response when a violation occurs.

The recommended progressive path is:



Organizations need not begin with complete proofs. Schema validation and runtime monitoring can materially increase assurance. Model checking and proofs of critical properties can then be introduced incrementally as specifications mature and teams develop the necessary capabilities.

8.6 Overall Assurance Model

This white paper proposes an overall ASE assurance model: formal core + statistical shell + runtime evidence.

- **Formal core:** Prove or check critical properties within the scope of formal models and assumptions, and enforce their boundaries through implementation-correspondence checks and mandatory controls. This does not provide an unconditional guarantee for the entire real-world system.
- **Statistical shell:** Evaluate observable behavior and residual uncertainty on representative tasks, producing confidence-qualified conclusions, including empirical evidence related to safety.
- **Runtime evidence:** Continuously observe deviations under real-world conditions and use pre-execution interception, alerts, staged rollout, and incident response to test and revise existing conclusions.

Together, these three elements support bounded quality claims. Formal proofs cannot replace validation of real needs; statistical results cannot guarantee behavior in uncovered cases; and runtime observation cannot cover risks that are unmonitored or have not yet occurred. Assurance should therefore be built from complementary evidence and residual-risk management.

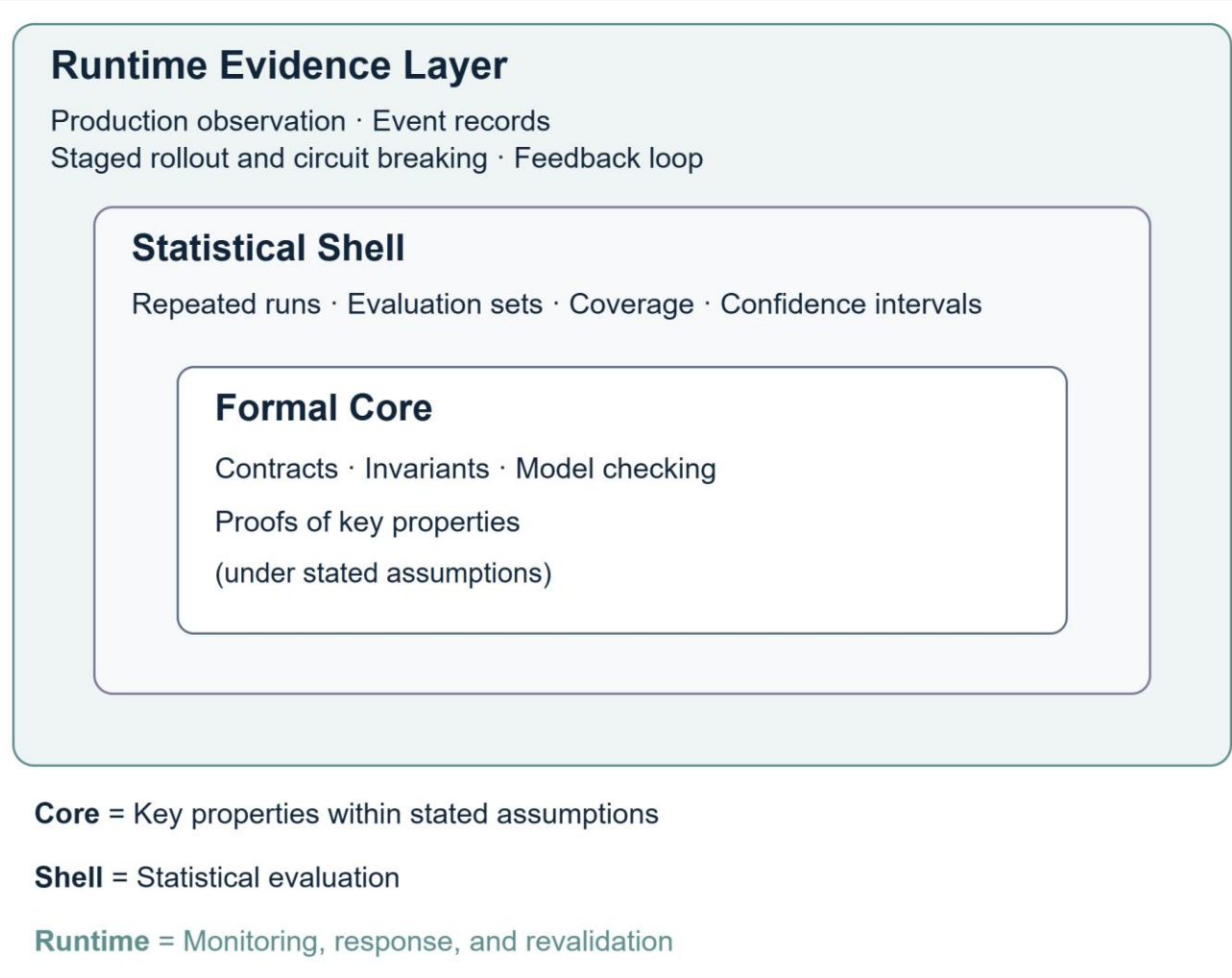


Figure 4. Trustworthiness Assurance Model: "Formal Core + Statistical Shell + Runtime Evidence"

8.7 Boundaries of Formal Verification and Honest Claims

When formal verification is introduced, its limitations must be stated explicitly:

- **Conclusions cover only explicitly stated properties and assumptions.** Formal analysis may reveal conflicts in a specification, but it cannot automatically ensure that the specification is complete or faithful to business intent expressed in natural language.
- **The object of proof is usually a model rather than the implementation.** Unless a rigorous correspondence is established between the model and the implementation, proof conclusions cannot be transferred directly to the actual system.
- **Cost depends on the object and objective.** Lightweight contracts, model checking, and complete machine-checked proofs differ greatly in effort; no unqualified multiplier should be used to characterize all formal-method practices. [7]

- **The toolchain itself can fail.** The trustworthiness of proof assistants and solvers is a prerequisite for trusting proof conclusions.

Quality claims must be bound to the versions of the specification, model, and implementation, and must list the verified properties, tools and configurations, assumptions, uncovered items, and residual risks. A claim must also explain the correspondence between the model and the implementation. When the specification, model, tools, or environment changes, the validity of earlier conclusions must be reassessed. The phrase "formally verified" must not be detached from these boundaries and used as a promise of overall system safety.

8.8 Minimum Practice Template

1. Critical-property list: Identify a small number of properties to verify first based on risk, and identify the owner, formalization target, assumptions, and enforcement point for each. Do not impose a universal minimum count.
2. Contract and schema validation: Prioritize critical cross-service interfaces as the starting point for structured checks. Passing a structural check is not equivalent to proving overall functional correctness.
3. Runtime monitors: Convert executable critical properties into monitoring or interception rules. For irreversible side effects, prioritize blocking before execution and provide timeout, takeover, and failure-handling mechanisms.
4. Risk-classification table: Use Table 3 to establish the organization's risk levels and corresponding assurance requirements, and embed them in the release process.
5. Claim template: Standardize the wording of external quality claims and require every claim to include the scope of proof and the portions not covered.

High-risk example (payment-system change): Treat "deployment requires prior approval" and "repeated requests must not cause duplicate charges" as critical properties. Model or prove the authorization state machine and idempotency, and use concurrency, retry, and fault-injection tests to check the implementation. Isolate production credentials; allow deployment only after two-person approval; use policy enforcement to intercept unauthorized actions before execution; and monitor amount and state anomalies at runtime, pausing subsequent operations when they occur. A charge that has already occurred must not be assumed to be reversible; reconciliation, compensating actions, and human handling are required. Bind the evidence package to the versions of the specification, code, model, tests, approvals, and runtime, and explicitly state assumptions about the unmodeled payment network and environment.

Capability summary: Verification and trustworthiness assurance address the risks that critical boundaries will be breached and that a local proof will be extrapolated into a claim of overall system safety. Inputs include a list of critical properties, risk classification, formal models and assumptions, and runtime policies. Outputs include verification evidence, monitoring and interception rules, and quality claims that state assumptions and uncovered items. Authorized personnel approve release according to the risk level, and the accountable owner signs external quality claims.

Conclusion: Starting with a Minimal Observable and Verifiable Closed Loop

C.1 Shared Priorities for the Industry

This white paper proposes three shared priorities:

First, recognize the change in who performs the work. The key question is not simply whether an agent can write code, but whether the resulting software can be verified, operated, and maintained and whether accountability is clearly assigned. ASE aims to retain human participation while redefining the responsibilities of people, agents, and engineering platforms. People set goals and retain accountability; agents interpret and orchestrate the work; and verification and rollback become shared capabilities.

Second, broaden investment beyond individual model capabilities to the closed-loop engineering system. The extent to which model capabilities translate into software delivery capacity depends on the coordinated development of specifications, context, execution control, testing, verification, and governance.

Third, start with a minimal closed loop rather than full autonomy. Begin with a use case whose risks can be controlled, boundaries are clear, and changes can be rolled back. Establish a minimal loop of specifications, execution, evidence, and acceptance that can be observed, measured, and improved. Then gradually expand autonomy and use-case complexity.

Autonomous execution can be authorized, but accountability cannot be left undefined. Identify the people responsible for confirming requirements, accepting risk, and handling exceptions. Organizations and their authorized personnel retain the responsibilities assigned by applicable law and organizational policy.

C.2 Topics for Future Editions

Given the scope and length of this first edition, the framework identifies the following topics without developing them in detail. Later technical editions or focused reports will examine them more fully:

- Specification languages and requirements compilation; engineering methods for context, state, memory, and knowledge
- Reference architectures and capability requirements for agent harnesses
- Design patterns for execution loops and execution graphs, including exception recovery
- Roles, communication protocols, and conflict resolution in multi-agent collaboration
- Industry use cases; mapping the 5×5 AI-native software development maturity model to ASE capability domains; business-oriented L1–L5 automation levels; and the development of testbeds and evaluation benchmarks
- Engineering practices for deployment and operation, staged rollouts, and real-time circuit breaking
- Potential connections to interface interoperability, standards, certification, and compliance frameworks

The proposed directions involving standards and certification are long-term possibilities, not established industry consensus. This white paper does not present them as settled conclusions.

References

- [1] A. E. Hassan, H. Li, D. Lin, et al., “Agentic Software Engineering: Foundational Pillars and a Research Roadmap,” arXiv:2509.06216, 2025, v3, updated June 24, 2026 (preprint). <https://arxiv.org/abs/2509.06216v3>
- [2] C. E. Jimenez, J. Yang, A. Wettig, et al., “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” ICLR, 2024. <https://arxiv.org/abs/2310.06770>
- [3] X. Deng, J. Da, E. Pan, et al., “SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?” arXiv:2509.16941v2, 2025 (preprint; its contamination-resistant design does not establish that all contamination risks have been eliminated). <https://arxiv.org/abs/2509.16941v2>
- [4] C. Newcombe, T. Rath, F. Zhang, et al., “How Amazon Web Services Uses Formal Methods,” Communications of the ACM, 2015. <https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods>
- [5] G. Klein, J. Elphinstone, G. Heiser, et al., “seL4: Formal Verification of an OS Kernel,” SOSP, 2009. <https://sel4.systems/Research/pdfs/seL4-formal-verification-operating-system-kernel.pdf>
- [6] X. Leroy, “Formal Verification of a Realistic Compiler,” Communications of the ACM, vol. 52, no. 7, pp. 107–115, 2009. <https://xavierleroy.org/publi/compcert-CACM.pdf>
- [7] G. Bernardi, A. Francalanza, M. Peressotti, and M. R. Mousavi, “Software is infrastructure: failures, successes, costs, and the case for formal verification,” arXiv:2506.13821, 2025, v4, updated January 29, 2026 (preprint). <https://arxiv.org/abs/2506.13821v4>
- [8] European Parliament and Council of the European Union, Regulation (EU) 2024/1689 (Artificial Intelligence Act), 2024. Distinguish the regulation's entry into force from the application dates of its individual provisions, and consult the current official consolidated text. <https://eur-lex.europa.eu/eli/reg/2024/1689/oj>
- [9] ISO/IEC, ISO/IEC 42001:2023, Information technology — Artificial intelligence — Management system, 2023. <https://www.iso.org/standard/42001>
- [10] NIST, Artificial Intelligence Risk Management Framework (AI RMF 1.0), NIST AI 100-1, 2023. <https://doi.org/10.6028/NIST.AI.100-1>
- [11] NIST, Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile, NIST AI 600-1, 2024. <https://doi.org/10.6028/NIST.AI.600-1>
- [12] ISO/IEC/IEEE 12207, Systems and software engineering — Software life cycle processes, and the ISO/IEC/IEEE 29119 Software testing series. These standards families provide background. A project citing them must specify the applicable edition, part, and clauses; citing a standard does not by itself establish conformity.
- [13] RTCA DO-178C / EUROCAE ED-12C; ISO 26262; IEC 61508; EN 50128. These provide background from high-assurance domains. Current editions, supersession relationships, applicable industries, and approval requirements must be checked individually. They cannot be applied directly as a basis for ASE compliance.
- [14] Cyberspace Administration of China and six other agencies, Interim Measures for the Management of Generative Artificial Intelligence Services, effective August 15, 2023. Applicability depends on the users served, how the service is provided, and the relevant jurisdiction. The measures should not be treated as imposing uniform obligations on all internal ASE activities.

- [15] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. <https://lamport.azurewebsites.net/tla/book.html>
- [16] Anthropic Engineering, “Effective harnesses for long-running agents,” November 26, 2025; “Demystifying evals for AI agents,” January 9, 2026. These describe a vendor's engineering practices, not cross-industry standards.
<https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents> ;
<https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents>
- [17] Agentic AI Harness Engineering and Loop Engineering: Why, What, and How to Build High-Performance, Reliable, Consistent, and Trusted Agent Systems. Deep Research Report, Version 4.1, July 2026. Supporting research document: `Agentic_AI_Harness_and_Loop_Engineering_Deep_Dive_v4.1.md`. For terminology, see Sections 7.5–7.11, 8.4, and 9.6.
- [18] W. Kong, Y. Lin, X. Teoh, D. M. Nguyen, R. Ren, J. Chang, H. Hu, and H. Chen, “Compiling Large Multi-Modal Requirement Documents into Runnable Software Systems,” International Symposium on Software Testing and Analysis (ISSTA 2026), 2026.

Note on references: These sources illustrate research, mechanisms, and engineering practices. Their inclusion does not imply that this white paper adopts all their conclusions or recommends any product. The framework, classifications, and examples in the text are proposals made in this white paper; they are not presented as broadly validated industry standards. The applicable editions, scope, and implementation status of standards and regulations must be checked. Claims that could not be adequately verified have been removed, qualified, or identified in review comments.

Appendix A: Checklist for a Minimal ASE Closed Loop

Use this checklist to assess an initial ASE closed loop. Failure to meet a core requirement (●) should block progression. If an item is genuinely not applicable, record the reason, associated risk, and approver. A not-applicable designation must never be used to bypass a legal prohibition or a nonnegotiable safety boundary.

Specifications (●)

- ☐ The task has all four specification categories: goals and acceptance; behavior and quality; interfaces and tools; and assurance and accountability.
- ☐ Every requirement has a predefined, reviewable decision procedure that is automated, manual, or a combination of both.
- ☐ Specifications are version-controlled, and changes are traceable.
- ☐ Specifications pass automated completeness checks before execution begins.

Execution (●)

- ☐ Agents run in a sandbox with only the minimum necessary permissions, which can be revoked at any time.
- ☐ Critical actions, tool calls, permission use, and decision records are traceable. The scope and limitations of replay are documented.
- ☐ Agents have explicit mechanisms for stopping execution and requesting clarification.

Testing and Evaluation (●)

- ☐ Risk-appropriate test baselines are in place and enforced through independent release gates.
- ☐ Controls check for unauthorized weakening of tests and access to hidden evaluations. Legitimate test maintenance requires independent approval.
- ☐ Critical conclusions involving nondeterministic capabilities or estimated proportions are supported by repeated trials and statistical reports appropriate to the sampling structure.
- ☐ Development, regression, and held-out evaluation sets are isolated. Contamination risks and the limits of unverifiable training data are recorded.

Verification and Assurance (● for Critical Controls; Rigor Based on Risk)

- ☐ Critical properties are identified by risk, with explicit assumptions, verification scope, accountable owners, and enforcement points.
- ☐ Interfaces between services use schema validation and contract checks.
- ☐ Monitorable critical properties have blocking, alerting, and response controls. Irreversible side effects are prevented before execution wherever possible.
- ☐ The organization has established a risk-classification matrix and embedded it in the release process.

Evidence and Acceptance (●)

- ☐ Intent, specifications, actions, artifacts, verification, claims, approvals, and audits form a versioned evidence chain.

- Gates deny by default. Exceptions require recorded authorization, and nonwaivable boundaries cannot be bypassed by exception.
- Low-risk work is released under preauthorized policies. Designated personnel approve high-risk work and exceptions.
- Risk-appropriate rollback, compensating-action, or loss-containment plans are in place and have been rehearsed as needed.

Governance (●)

- Responsibilities of people, agents, and platforms are clearly defined and documented.
- External quality claims state the scope of proof, tools used, assumptions, and what has not been covered.
- Measures cover success rates, rework, cost, latency, human intervention, rollback, and severe failures.
- Post-release incidents, failure cases, and evaluation feedback inform specifications and tests. Migration and retirement include data disposition and permission revocation.

Appendix B: Terminology

This table defines the terms and conceptual boundaries used in this white paper's discussion of agent execution engineering, for consistent use in Section 2.1 and subsequent reports. The paper's Chinese renderings of these terms are working conventions, not established industry standards. [17]

Term	Definition	Boundaries and Usage
Framework Development framework	Provides developers with components and abstractions for building systems, including model adapters, tool wrappers, workflows, and state access.	Supports system construction and exposes design choices. A harness may be implemented with a framework, but the terms are not interchangeable.
Runtime	Provides execution mechanisms such as processes, containers, queues, networking, storage, resource limits, and scheduling.	Focuses on execution mechanisms. A harness uses those mechanisms to organize behavior and enforce operating policies.
Harness Runtime control harness	Organizes and enforces the runtime behavior and control policies of an agent system built around a model and supported by an underlying runtime environment.	Coordinates context, tools, state, permissions, budgets, and recovery. It is neither a development framework nor the underlying runtime environment, and it does not require changes to model parameters.
Context	The complete set of visible input information selected and assembled for a single model invocation.	An information view for the current invocation, not the entirety of task state, session history, memory, or knowledge.
State Task state	A state object that records current task progress, artifacts, pending work, and control conditions.	Critical state should be persisted and governed by update rules. Only the portion needed for the current decision enters the context.
Session	Records the event history, interactions, and associated artifacts of an ongoing task, supporting handoffs, checkpoints, and recovery.	More than a chat log, and distinct from a worker process or context. Persistent records can span model invocations and processes.
Memory	Information retained from past observations and experience for reuse in subsequent tasks.	Not the entire session history. Distinguish candidate memories from validated memories, and manage their provenance, scope, currency, and invalidation conditions.
Knowledge	Governed information approved for broader reuse, with documented provenance and boundaries on applicability.	Memory can become knowledge through validation and governance. Knowledge must still be filtered by permissions and relevance before entering the context.
Loop Execution loop	An iterative process that uses observations, verification, and progress feedback to decide whether to continue, correct, pause, resume, escalate, or terminate execution.	Describes feedback control over time. It requires budgets and disciplined stopping rules; it does not describe relationships between nodes or an execution topology.
Graph Execution graph	A graph of nodes, edges, and shared state that represents execution dependencies, routing, parallelism, joins, and state transfer.	May contain bounded loops or be acyclic. Nodes need not be agents. This meaning applies to runtime orchestration in this white paper.

Term	Definition	Boundaries and Usage
Relationship Graph	Explicitly represents relationships among participants, goals, tasks, artifacts, evidence, permissions, and other entities.	Clarifies the broader meaning of graph. A relationship graph is not automatically an executable workflow, and a graph structure is not synonymous with a graph database.

Terminology note: In the Chinese edition, Harness, Framework, and Runtime are rendered as “运行控制框架,” “开发框架,” and “运行时,” respectively. Loop and Graph are rendered as “执行循环” and “执行图” and are not interchangeable; do not use “Loop / Graph” as though they were synonyms. Outside execution orchestration, Graph may refer more broadly to a relationship graph; its meaning must be clear from the context.

Closing Note

This first edition is a discussion draft intended to help develop industry consensus. The framework and methods will be refined through practice. We invite research institutions, enterprises, open-source communities, and standards organizations to contribute appropriately de-identified, reviewable use cases, task contracts, failure cases, evaluation methods, and verification examples, and to help test the boundaries of the framework's applicability.



OPEN AGENTIC INDUSTRY CONSORTIUM
开放智能体产业联盟

Build neutral infrastructure and a global collaboration ecosystem for agentic AI

Open Agentic Industry
Consortium (OAIC)

www.oaic.ai

