

**Outcome Assurance for Agentic Systems:
How Do You Know an AI Agent Finished the Job, Correctly?**

Dr. Joe Shepherd
Independent Researcher

Author Note

Correspondence concerning this article should be addressed to Joe Shepherd. Email:
joe@drjoeshepherd.com

Abstract

AI agents increasingly perform consequential tasks that extend beyond a single model response, involving decisions, external actions, approvals, waits, retries, delegation, and state changes across multiple systems. Existing disciplines—including evaluation, runtime verification, authorization, observability, provenance, and independent verification and validation—provide important assurance mechanisms, but they establish different claims about different parts of execution. This paper proposes Outcome Assurance, a task-centered framework for determining whether an agentic task has achieved an acceptable intended outcome. The framework treats the task, rather than the model invocation or agent process, as the primary assurance boundary and distinguishes four assurance layers: Decision Assurance, Action Assurance, Execution Assurance, and End-State Assurance. Assurance claims are established from contextual requirements, admissible evidence, evaluation methods, and explicit verdicts; they may be invalidated as context changes and must be reconciled when work is delegated. Task completion is treated as a separate assurance claim established through a Completion Contract that evaluates the task’s current assurance state against task-level acceptance conditions. The central contribution is therefore not evidence-gated completion itself, but the composition semantics by which currently usable assurance claims across decisions, actions, execution, delegation, and end-state verification support verified completion. The framework also defines Outcome Reliability across task populations and presents falsifiable hypotheses and an experimental design for testing whether these composition semantics provide measurable assurance value beyond strong evidence-gated completion. The paper reports no experimental results.

Keywords: outcome assurance, agentic systems, assurance cases, completion verification, runtime verification

Outcome Assurance for Agentic Systems: How Do You Know an AI Agent Finished the Job, Correctly?

Notation and Symbols

Notation convention. The mathematical notation in this paper provides a compact representation of framework relationships and testable propositions. Unless explicitly identified as a formal entailment or equivalence, displayed implications should be interpreted as conceptual relationships rather than theorems within a complete formal calculus. Framework symbols defined below retain the stated meaning throughout the paper unless a local symbol is explicitly defined within a bounded example.

Tables 1 and 2 list the framework notation and the operators and relations used in this paper.

Table 1

Framework Notation

Symbol	Meaning
T, T_i, T_c	Task, task instance i , and task class.
T_p, T_{ch}	Parent and child tasks.
O_T, Γ_T, L_T	Task objective, constraints and completion conditions, and lifecycle.
X_t, X	Task-relevant context at time t ; stated operating conditions or profile when time indexing is unnecessary.
π	Execution trajectory or path.
N_i, R_i, L_i	Consequential node or transition, applicable assurance requirement, and assurance layer.
C_i, C_T	Assurance claim i and task-level completion claim.
E_i, E_T	Evidence for claim i and task-level completion evidence.
M_i, M_T	Evaluation method for claim i and task-level evaluation method.
B_i, B_T	Verifier trust boundary and assumptions for claim i ; task-level verifier independence and trust-boundary requirements.
V_i, V_T	Claim verdict and task-level completion verdict.
D_i, D_T	Claim disposition and task-level disposition.
P_T, p_i	Task acceptance predicates and an individual predicate.

Symbol	Meaning
$S_f, \hat{S}_t$	Final external state and observed external state at time t .
$\mathcal{E}_T$	Set of acceptable end states for task T .
θ_T, τ_T, Q_T	Acceptable outcome equivalence or tolerances, temporal semantics, and continuing obligations.
Δ_i, W_i, J_i	Claim dependency set, validity window, and invalidation or revalidation trigger set.
$\mathcal{A}_p, \mathcal{A}_{ch}$	Parent and child authority sets or envelopes.
G	Agentic configuration.
GT_i	Experimental ground truth for task instance i .
OAR_i, VC_i, TC_i, CC_T	Outcome Assurance Requirement representation, Verification Contract, temporal claim state, and Completion Contract. Contracts are specifications fixed before evaluation; the verdicts they produce are recorded separately.
R_O	Outcome Reliability.
$AC_T(t), AS_T(t)$	Dynamic assurance case (durable record of established and retained claims) and Task Assurance State (the claims in it that are currently usable), $AS_T(t) = Usable(AC_T, t)$ (Section 6.9).
$DAR_{ch}, Open_{ch}$	Delegated Assurance Result returned by child task T_{ch} , $DAR_{ch} = (Result_{ch}, AS_{T_{ch}}(t_r), Open_{ch})$, and its unresolved obligations, unmet requirements, and Indeterminate claims (Section 9.12).
$Verified(T)$	Shorthand for $VerifiedComplete(T, t)$ where the verification time is not material (used chiefly in Sections 9 and 10).
$MaterialChildren(T_p)$	Delegated children whose results or claims support a mandatory predicate or requirement of parent T_p (Section 9.8).
$B_A - B_E$	Experimental baselines: agent self-report, execution success, end-state task success, equal-specification check, and evidence-gated completion (Section 13).
$P_{process}, P_{outcome}$	Process and outcome subsets of the acceptance predicates P_T (Section 10.3).
$Pol_k, Auth_k, Ap_k, Sup_k$	Policy version, authority grant, approval, and supplier identifiers used in worked examples.

Table 2*Operators and Relations*

Symbol	Meaning
$=, \neq$	Equality and inequality.
$\Rightarrow$	Implication; in conceptual expressions, the antecedent would be sufficient for the consequent.
$\nRightarrow$	Non-implication; the antecedent alone is insufficient to establish the consequent.
$\Leftrightarrow$	Equivalence under the stated definitions or conditions.
$\wedge, \vee, \neg$	Logical conjunction, disjunction, and negation.
$\in, \subseteq$	Set membership and subset containment.

Symbol	Meaning
$\forall, \exists$	Universal and existential quantification.
$P(\cdot)$	Probability of the specified event, conditional on any stated variables.
$\models$	Semantic satisfaction or entailment; used only where formal satisfaction or entailment is intended.
$Preserves(R_{ch}, R_p^{req})$	The child requirements preserve or strengthen every mandatory parent assurance obligation allocated to the delegated work.
$Eventually_I(\phi),$ $Always_I(\phi)$	Metric temporal operators: ϕ holds at some time, or at every time, within interval I (for example $I = [0, 30m]$, measured from a stated start event).
$Usable(C_i, t)$	Claim C_i is <i>Satisfied</i> , within its validity window, has valid dependencies, and no invalidation trigger has fired at t (Section 8.11).
$Reconciled(DAR_{ch}, T_p, t)$	The child's returned assurance state has been reconciled against parent T_p 's requirements and context at time t (Section 9.8).
$\sqcap$	Conjunction of three-valued verdicts (strong Kleene): <i>NotSatisfied</i> if any input is <i>NotSatisfied</i> ; <i>Satisfied</i> only if every input is <i>Satisfied</i> ; otherwise <i>Indeterminate</i> .

1. From Model Outputs to Agentic Outcomes

The unit of assurance in AI systems is changing.

A conventional language-model interaction can often be represented as:

$$x \rightarrow M(x) \rightarrow y$$

where x is an input, M is a model invocation, and y is an output. Evaluation can therefore focus substantially on properties of y : factual accuracy, relevance, toxicity, groundedness, calibration, or agreement with a reference answer.

An agentic task is different. Its objective may persist beyond any individual model invocation and may require decisions, external actions, approvals, waits, retries, resumptions, compensation, human intervention, and delegated child tasks:

$$T \rightarrow \{d_1, a_1, w_1, d_2, a_2, \dots, a_n\} \rightarrow S_f$$

where T is the task, d_i is a decision, a_i is an action that may create an external side effect, w_i is a temporal transition or dependency, and S_f is the observed state following execution.

The task may span multiple agents, models, tools, credentials, services, and runtimes. More importantly, successful execution of its constituent operations does not establish successful completion of the task.

A tool call may succeed while its effect fails downstream. A transaction may commit while another required system remains unchanged. Authority or approval may expire while execution is suspended. A timeout may leave the status of a side effect unknown. A child task may succeed while the parent outcome remains incomplete. Every planned step may execute while the required final state still does not exist.

These are not exclusively AI problems. Distributed systems have long addressed partial failure, uncertain state, long-running transactions, compensation, retry semantics, and non-atomic workflows (Garcia-Molina & Salem, 1987). Runtime verification evaluates execution against specifications and explicitly addresses circumstances in which incomplete or uncertain evidence prevents a sound verdict (Bartocci & Falcone, 2018; Taleb et al., 2023). Agent-reliability research likewise distinguishes outcome consistency from trajectory consistency, demonstrating that variation in how an agent executes can be separated from variation in whether it succeeds (Rabanser et al., 2026).

Agentic systems bring these problems together with probabilistic decision making, dynamically generated execution paths, delegated work, and externally consequential actions. This changes the assurance question.

For a model response, one may ask:

IsOutputAcceptable(y)?

For an agentic task, the stronger question is:

HasTaskAchievedIntendedOutcome(T)?

The two questions are not equivalent.

$$\textit{SuccessfulExecution}(T) \not\Rightarrow \textit{SuccessfulCompletion}(T)$$

This paper begins from that distinction:

Execution is a process. Outcome is a state of the world produced or established by that process.

For consequential agentic work, assurance of the process cannot in general substitute for assurance of the outcome.

Existing mechanisms remain necessary. Evaluation can measure performance.

Authorization can determine whether an action is permitted. Runtime verification can evaluate execution traces against specifications. Observability can expose system behavior. Provenance can preserve relationships among evidence, actions, and actors. Independent verification can provide objective evidence outside the executor's own assertion.

The problem addressed here is instead how their evidence and decisions can be composed across the lifetime of an agentic task to establish a stronger task-level claim: that the intended outcome has actually been achieved under the requirements applicable to the work that produced it.

This paper proposes *Outcome Assurance* for that purpose.

Outcome Assurance treats the task and its intended outcome as the primary assurance boundary. Its proposed framework:

1. distinguishes *Decision, Action, Execution, and End-State Assurance* as separately testable assurance surfaces;
2. constructs assurance claims contextually at consequential transitions rather than assigning universal assurance properties to an agent;

3. preserves and reevaluates claims and evidence as tasks wait, resume, retry, delegate, and encounter changing context;
4. propagates assurance obligations across delegated work while requiring parent-level reconciliation;
5. separates end-state observation from the stronger task-level claim of verified completion;
6. represents completion through an explicit *Completion Contract*; and
7. extends instance-level Outcome Assurance into *Outcome Reliability*, which measures the rate at which a defined agentic configuration produces verified outcomes across a defined task population.

Each of these constructs draws on established mechanisms, reviewed in Section 2. The paper’s central proposition concerns their composition:

Outcome Assurance determines whether a task may be considered complete from the current assurance state supporting that outcome—not merely from evidence that the desired end state exists.

These constructs are proposed as a framework to be tested, not as experimentally established properties of agentic systems. The paper therefore specifies falsifiable hypotheses, failure conditions, and experimental methods under which the proposed distinctions should be retained, collapsed, or revised.

The central question is simple:

How do you know an AI agent finished the job, correctly?

Outcome Assurance proposes that the answer must come from the currently usable assurance state establishing the task’s outcome—not merely from the agent or execution process reporting that it is done.

2. Existing Assurance Mechanisms Address Different Questions

Outcome Assurance does not replace AI evaluation, runtime verification, authorization, observability, provenance, or software assurance. Its premise is that these disciplines answer different questions and therefore provide different classes of evidence.

A useful decomposition is:

$$\text{Permission} \neq \text{Execution} \neq \text{Evidence} \neq \text{Outcome}$$

The distinctions matter because success in one category does not establish success in another.

2.1 Evaluation

AI evaluation estimates properties of model or agent behavior under specified test conditions. The National Institute of Standards and Technology (NIST) AI Risk Management Framework explicitly treats testing, evaluation, verification, and validation as part of operationalizing trustworthy AI. It also distinguishes validity and reliability from other trustworthy-AI characteristics (Tabassi, 2023).

Agent evaluation expands the evaluated object beyond a model response. Contemporary frameworks can measure tool selection, trajectory behavior, task success, repeated-run consistency, resource consumption, robustness, and safety (Rabanser et al., 2026).

This is an important expansion, but task success as an evaluation metric and verified completion of an individual production task are different claims.

An empirical success rate estimates behavior over a population of executions. Outcome Assurance asks whether a particular task instance satisfies its required completion conditions.

Formally, an estimate such as

$$P(\text{success} \mid T \in T_c, X)$$

for task class T_c under stated operating conditions X does not establish

$$\text{Completed}(T_i)$$

for a particular task instance T_i .

2.2 Authorization and Policy Enforcement

Authorization answers whether a principal is permitted to perform an operation under some applicable authority or policy.

This is necessary for consequential agents, but authorization establishes a *precondition for execution*, not its eventual effect.

For an action a ,

$$\text{Authorized}(a) \not\Rightarrow \text{Effect}(a) = \text{IntendedEffect}(a)$$

An authorized payment can fail. An authorized record update can modify the wrong object because of an execution defect. An authorized child agent can return incomplete work. An authorized API invocation can time out after committing its side effect.

The proposed framework keeps policy, delegated authority, risk, and assurance as distinct typed results (Section 6.1).

2.3 Runtime Verification

Runtime verification provides perhaps the closest established technical analogue to part of the proposed framework.

Runtime verification dynamically evaluates execution traces against formal specifications. It generally requires a specification, instrumentation capable of producing relevant events, and a monitor capable of evaluating those events (Bartocci & Falcone, 2018).

This provides a strong foundation for conformance evaluation within Outcome Assurance (Section 6.5).

Runtime verification also exposes an important limitation: a sound verdict depends on the evidence available to the monitor. Research on uncertainty in runtime verification explicitly examines incomplete or imprecise traces and circumstances in which a monitor cannot soundly determine whether a property has been satisfied (Taleb et al., 2023).

For agentic work, this limitation becomes particularly important when a trace reports:

$$ToolCall(request) \rightarrow Success(response)$$

while the assurance requirement concerns:

$$ObservedExternalState = RequiredExternalState$$

The second proposition cannot necessarily be inferred from the first.

Outcome Assurance therefore treats execution traces as evidence, not automatically as proof of the final outcome.

2.4 Provenance

World Wide Web Consortium (W3C) PROV provides a mature conceptual model for representing provenance through *Entities*, *Activities*, and *Agents*, together with relationships including generation, derivation, use, association, attribution, and delegation. PROV explicitly supports representing responsibility and cases where one agent acts on behalf of another (Moreau & Missier, 2013).

A task may need to preserve relationships such as:

$$Evidence \leftarrow Decision \leftarrow Agent \leftarrow DelegatingPrincipal$$

or:

$$FinalState \leftarrow Action \leftarrow ChildTask \leftarrow ParentTask$$

Provenance makes such relationships representable, but provenance and assurance remain distinct.

A perfectly complete provenance record can faithfully describe an incorrect result.

Traceability establishes the basis from which a result can be reconstructed or investigated; it does not by itself establish that the result satisfies the task's acceptance conditions.

Thus:

$$Traceable(x) \not\Rightarrow Correct(x)$$

Traceability is therefore treated as a requirement on evidence within Outcome Assurance (Section 7.3).

2.5 Independent Verification and Validation

NASA defines independent verification and validation (IV&V) as a software-assurance discipline employing rigorous analysis and testing to obtain objective evidence and provide an independent assessment of critical products and processes throughout the lifecycle. Its current standard explicitly considers both nominal operation and off-nominal conditions (National Aeronautics and Space Administration, 2022).

Outcome Assurance adopts the underlying principle of evidence-backed verification independent from the actor making the claim, while applying it to a different object: dynamically executed agentic work.

The executing agent may participate in generating evidence. It should not necessarily be able to unilaterally establish that the conditions defining its own successful completion have been satisfied.

2.6 Dynamic and Runtime Assurance Cases

Assurance cases organize claims, arguments, evidence, and context into an auditable structure (Object Management Group, 2023). A substantial line of work has made such cases dynamic. Denney, Pai, and Habli proposed dynamic safety cases as a means of operationalizing through-life safety assurance (Denney et al., 2015). ENTRUST develops a partial assurance case at design time and completes it at runtime with adaptation-correctness evidence produced by self-adaptive software (Calinescu et al., 2018). ACCESS periodically evaluates an assurance case at runtime by executing evaluation rules, and reports the result to the system without taking control of it (Wei et al., 2024). For frontier AI, Cărlan et al. propose a dynamic safety case management system in which consistency checks first identify the argument elements directly impacted by a system change and then the elements indirectly impacted (Cărlan et al., 2024). Process-centric systems have long separated objective achievement from activity completion: Guard-Stage-Milestone lifecycles mark milestones achieved by conditions and allow invalidating sentries to make an achieved milestone false again (Hull et al., 2011).

These works establish mechanisms on which Outcome Assurance builds: claims, evidence, and context; runtime construction and evolution of an assurance case; runtime evaluation of its claims; and change-triggered reevaluation of dependent elements. Outcome Assurance adopts them. In the works reviewed here, the assurance object is a system, deployment, or business artifact rather than an individual agentic task whose completion must be

decided. Nor do they address how assurance returned by delegated work enters a parent’s completion decision.

2.7 Evidence-Gated Agent Completion

A recent and rapidly growing body of agent-systems work already separates an agent’s report of success from verified completion and gates completion on independent evidence. EvidenceNet is a trusted runtime assurance layer for multi-agent network operations across authority boundaries: agents cannot declare their own success, a trusted completion contract specifies the observations required, each observation is bound to its source, request, command, and network epoch, observations collected at an earlier epoch are stale, and the submission gate is fail-closed (Zhang et al., 2026). TraceGrant compiles a permission–obligation–evidence–completion contract, tracks each obligation as pending, active, verified, or waived, distinguishes authorization, successful native execution, and completion of the overall task, and accepts a success claim only when every hard obligation has valid support (Liao et al., 2026). Proof-or-Stop treats agent outputs as proposed lifecycle claims rather than lifecycle state, requires fresh evidence bound to tracked source state, enforces review as a lifecycle gate, and reports signed receipt bundles that rejected the tamper classes it tested (J. Huang et al., 2026). Verify-Gated Completion allows agents to propose completion while a read-only admission verifier decides, using claim and evidence packets and resolving ambiguous or weakly evidenced cases fail-closed (Nguyen & Tran, 2026). EviBound places an approval gate that validates acceptance criteria before execution and a verification gate that checks artifacts in an authoritative tracking store after execution (Chen, 2025). Empirically, false success—an agent asserting task completion when the environment state shows otherwise—is common, and large language model (LLM) judges detect it poorly (Advani, 2026).

These systems show that two approaches are already established or emerging: separating execution and self-report from verified completion, and gating completion on independent authoritative evidence. Outcome Assurance adopts both rather than proposing them. The remaining distinction is:

Existing completion gates determine whether sufficient evidence supports completion. Outcome Assurance additionally maintains whether the assurance state upon which completion depends remains usable across time and delegated execution.

Several of these systems include freshness rules or invalidate earlier evidence, for example when the network epoch advances (Zhang et al., 2026) or when evidence no longer matches tracked source state (J. Huang et al., 2026). In the works reviewed here, however, completion is not defined over a task-wide set of supporting assurance claims whose individual usability is selectively revalidated as their dependencies change, nor over assurance state returned by delegated work that must be reconciled before it can support a parent’s completion. Whether those composition semantics add measurable assurance value beyond a strong evidence-gated completion architecture is the experimental question of Section 13.

3. The Task as the Primary Assurance Boundary

Selecting the correct assurance boundary is a design decision.

The *model* is too narrow because a task may span multiple models or model versions.

The *agent* is insufficient because one agent may execute many task classes with materially different consequences and assurance requirements.

The *execution trajectory* is insufficient because multiple trajectories may legitimately produce equivalent outcomes.

The *individual action* is too narrow because successful actions can compose into an unsuccessful task.

We therefore define the primary assurance object as the *task*.

Definition 1 — Task

A task T is a unit of intended work defined by at least:

$$T = (O_T, \Gamma_T, L_T)$$

where:

- O_T is the task objective or intended outcome;
- Γ_T is the set of task constraints and completion conditions; and
- L_T is the lifecycle through which the work progresses.

For practical implementations, the task may additionally specify initiating identity, accountable principal, applicable authority, data scope, deadlines, budgets, policies, and acceptance predicates. The proposed task model permits multiple decisions, actions, waits, retries, delegations, and validations.

3.1 Intended Outcome

The intended outcome must be distinguishable from the execution plan.

Let:

$$P_T = \{p_1, p_2, \dots, p_n\}$$

be the acceptance predicates for task T .

Let S_f be the externally observed state after execution.

In the simplest case, considering end-state predicates only, a completion condition is:

$$Verified(T) = \bigwedge_{i=1}^n p_i(S_f)$$

where every mandatory predicate must evaluate to true.

More complex tasks may use disjunctions, temporal constraints, tolerances, or acceptable equivalence classes rather than simple conjunction.

Completion is thus defined against *observed state*, not merely against exhaustion of the execution plan.

3.2 Execution Path Is Not Task Identity

Suppose two valid executions are:

$$\pi_1 = (a_1, a_2, a_3)$$

and

$$\pi_2 = (a_1, d_1, a_4, a_3)$$

If both comply with applicable constraints and produce states S_1 and S_2 belonging to the set of acceptable end states $\mathcal{E}_T$ (defined in Section 4.2),

$$S_1, S_2 \in \mathcal{E}_T$$

then the executions may both satisfy the task despite differing trajectories.

This distinction is consistent with current agent-reliability research, which explicitly separates outcome consistency from trajectory-distribution and trajectory-sequence consistency (Rabanser et al., 2026).

It also prevents an undesirable requirement that probabilistic agents reproduce identical internal execution paths when path variation is irrelevant to the intended outcome.

3.3 Consequential Nodes

Not every transition in a task warrants the same assurance cost.

We therefore define a *consequential node* as a decision or action whose result can materially affect an external system, person, resource, obligation, right, safety condition, financial state, governed record, or task-completion condition.

The task graph used by the proposed framework includes decisions, actions, waits, delegations, validations, and outcome predicates, with assurance requirements attached contextually to consequential nodes.

This leads to a further principle:

Assurance requirements attach to consequential work in context; they are not permanent attributes of the agent performing the work.

Consider the same agent performing three operations: `ReadAccount`, `RecommendRefund($100)`, and `ExecuteRefund($100)`.

The fact that one agent performs all three does not imply that all three should have identical evidence, authority, approval, verification, or stability requirements.

The object to which an assurance requirement attaches is therefore the consequential transition *within the task context*.

3.4 Why the Task Persists

Long-running execution makes task identity especially important.

At time t_0 , an agent may be authorized to perform a task using policy version Pol_1 , knowledge version K_1 , approval Ap_1 , and authority state U_1 .

Execution may then suspend.

At t_1 :

$$Pol_1 \rightarrow Pol_2, \quad K_1 \rightarrow K_2, \quad U_1 \rightarrow U_2$$

while the task itself remains:

$$T_{id} = constant$$

A robust system therefore cannot assume:

$$Assured(T, t_0) \Rightarrow Assured(T, t_1)$$

Previously established assurance conditions may require re-evaluation when execution resumes. Runtime-verification research recognizes that incomplete information can prevent sound verdicts; agentic execution makes the issue operationally significant because institutional and technical conditions may change during suspension (Taleb et al., 2023). The persistent object is therefore the task’s assurance record (the dynamic assurance case AC_T , Section 6.8) associated with the intended work, not a process or model invocation. Section 8 develops the corresponding revalidation mechanics.

4. Outcome Assurance

Definition 2 — Outcome Assurance

Outcome Assurance is the evidence-backed determination that an agentic task has reached an acceptable intended end state while satisfying the applicable assurance requirements governing the decisions, actions, and execution that produced it.

This definition, and the determination it describes, involve four requirements that should not be collapsed.

4.1 Evidence-Backed

Outcome Assurance is not established by model confidence, self-report, or unsupported evaluator judgment.

A completion claim must be supported by evidence appropriate to the predicates being evaluated.

For predicate p_i , let:

$$E_i = \{e_1, e_2, \dots, e_m\}$$

represent the evidence used to evaluate it.

Then the verdict for predicate p_i is:

$$V_i = \textit{Evaluate}(p_i, E_i, X_t)$$

where X_t contains the time-indexed context necessary to interpret both predicate and evidence.

Verification may use deterministic queries, cryptographic evidence, external systems, rules, sensors, independent models, human review, or combinations thereof. The appropriate verifier is the least probabilistic mechanism capable of evaluating the requirement, in the sense defined in Section 7.5.

4.2 Acceptable Intended End State

Outcome Assurance does not require one exact bitwise state unless the task requires it.

Let $\mathcal{E}_T$ be the set of states acceptable for task T :

$$\mathcal{E}_T = \{S: P_T(S) = \textit{true}\}$$

Completion requires:

$$S_f \in \mathcal{E}_T$$

rather than necessarily:

$$S_f = S^*$$

for one predetermined state S^* .

This distinction accommodates multiple acceptable implementations, routes, resources, timings, or representations and supports the separate population-level treatment of *Outcome Reliability* in Section 11.

4.3 Applicable Assurance Requirements

A correct final state is necessary but not always sufficient.

Consider a task whose objective is:

$$\textit{Balance}(\textit{account}) = \$900$$

An unauthorized change from \$1,000 to \$900 satisfies the end-state predicate through an unacceptable process. Conversely, an authorized, conformant process may fail downstream and leave the balance at \$1,000.

Therefore:

$$OutcomeAssured(T) \neq EndStateValid(T)$$

and:

$$OutcomeAssured(T) \neq ProcessConformant(T)$$

Instead, both are required:

$$OutcomeAssured(T)$$

$$\Rightarrow ProcessRequirementsSatisfied(T) \wedge EndStateRequirementsSatisfied(T)$$

subject to any explicitly defined exceptions, waivers, or tolerances.

The implication is one-way. Satisfied process and end-state requirements are necessary inputs, but they do not mechanically establish the outcome. The task-level determination is a separate claim evaluated under the task's Completion Contract (Section 10):

$$OutcomeAssured(T, t) \Leftrightarrow VerifiedComplete(T, t)$$

where *VerifiedComplete*(*T*, *t*) holds when the Completion Contract *CC_T* returns *Satisfied* for the completion claim at verification time *t*. The two names denote the same determination: Outcome Assurance is the framework, and *VerifiedComplete* is the task state it licenses.

Verified completion is therefore a verdict over the task's current assurance state, not merely its final observed state. The Completion Contract evaluates both the authoritative end state and the process and outcome claims on which completion relies, and admits only those claims that remain usable at verification time (Sections 6.9 and 10.14).

4.4 Independent Completion Determination

An executing agent may emit $Claim(T, Complete)$. That claim is evidence about the agent's execution state. It is not necessarily evidence that the external world satisfies the task's completion predicates.

Outcome Assurance therefore separates $ExecutionComplete(T)$ from $VerifiedComplete(T)$.

The proposed framework requires a logically independent verifier whose conclusion is not controlled solely by the executing agent's assertion.

At completion, the verifier evaluates authoritative evidence against task-level acceptance predicates and may return *Verified*, *NotVerified*, or *Indeterminate* (the completion-verification names for *Satisfied*, *NotSatisfied*, and *Indeterminate*). This adapts independent verification and validation's use of objective evidence without requiring a separate organization or physical infrastructure (National Aeronautics and Space Administration, 2022). It requires that:

$$\neg UnilateralControl(Executor, V_T)$$

That is, the executing agent cannot unilaterally determine the verifier's conclusion; the required independence should be proportional to the consequence of an incorrect completion verdict. Section 7 defines verifier independence and trust boundaries in full.

As Section 2.7 notes, independent evidence-gated completion verification is adopted here rather than proposed. Outcome Assurance adds the verifier's object: the task's current assurance state (Section 6.9), not only the evidence available at the moment of verification.

4.5 Outcome Assurance as a Layered Framework

A task may fail despite passing an earlier assurance stage. Outcome Assurance therefore decomposes the lifecycle into four layers:

$$Layers(OA) = \{DA, AA, EA, ESA\}$$

where:

Decision Assurance (DA) evaluates consequential judgments; *Action Assurance (AA)* evaluates proposed side effects and their authorization and execution requirements; *Execution Assurance (EA)* evaluates lifecycle integrity; and *End-State Assurance (ESA)* evaluates authoritative external-state observations against acceptance conditions.

The four layers are not necessarily sequential single-pass phases. A task may encounter each multiple times.

For example:

$$DA_1 \rightarrow AA_1 \rightarrow EA_1 \rightarrow Wait \rightarrow EA_2 \rightarrow DA_2 \rightarrow AA_2 \rightarrow ESA$$

A failed End-State Assurance check may also return the task to remediation rather than terminate it:

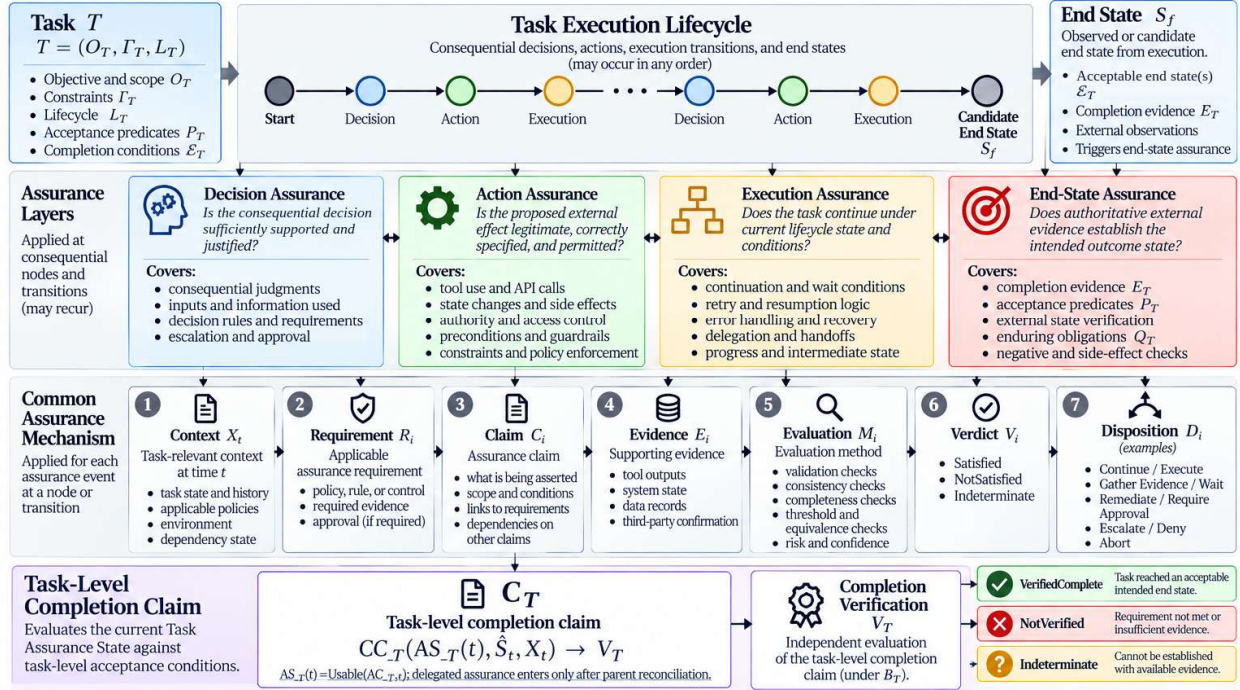
$$ESA_{fail} \rightarrow Remediation \rightarrow AA_3 \rightarrow ESA_2$$

Outcome Assurance is therefore not simply a final QA step but a lifecycle assurance model whose final claim is verified outcome completion.

Figure 1 summarizes the Outcome Assurance framework.

Figure 1*Outcome Assurance Framework*

Outcome Assurance treats the task as the primary assurance boundary. Consequential decisions, actions, execution transitions, and end states are evaluated through contextual assurance requirements. Their evidence-backed verdicts contribute to the task-level completion claim rather than functioning as a fixed sequential pipeline.

**4.6 What Outcome Assurance Does Not Claim**

Outcome Assurance does *not* imply:

- deterministic reasoning from a probabilistic model;
- identical execution trajectories;
- absence of all possible failure modes;
- that provenance proves correctness;
- that authorization proves successful execution;
- that an independently implemented verifier is necessarily correct;
- that every assurance property applies to every task;
- that a single scalar score adequately represents assurance; or
- that verified completion guarantees the outcome will remain valid indefinitely.

A completion verdict is always relative to:

$$(T, P_T, X_t, E_t, t_v)$$

where T is the task definition, P_T its acceptance predicates, X_t the applicable time-indexed context, E_t the available evidence, and t_v the verification time.

Outcome Assurance is a bounded claim supported by specified evidence under specified conditions at a specified point in time, not a declaration of absolute correctness.

Primary Proposition

For consequential agentic work, successful execution should not be treated as sufficient evidence of successful completion, and evidence that the desired end state exists is not by itself sufficient either. Assurance should be defined at the task boundary, preserve applicable requirements and evidence across the task lifecycle, and culminate in a completion verdict over the task's current assurance state: the process and outcome claims, including reconciled assurance returned by delegated work, that remain usable when authoritative observations are verified against the task's intended outcome.

This proposition is narrower than a claim that existing AI evaluation or governance is generally inadequate.

Research Hypothesis

H1: Full Outcome Assurance reduces false assurance and false completion relative to strong evidence-gated completion when failures arise from stale supporting claims, changed context, or unreconciled delegated assurance state.

Method: Construct controlled agentic tasks containing known failures at the decision, action, execution, and end-state layers, including failures introduced through claim expiry,

dependency or context change during suspension, and delegated assurance that is returned but unreconciled or later invalidated. Evaluate each task under the Section 13 baselines, including the evidence-gated completion baseline B_E (Section 13), which has no task-wide dependency tracking of supporting claims, no returned child assurance state, and no mandatory parent reconciliation.

Threshold: Outcome Assurance must reduce the false-assurance rate or false-completion rate relative to the strongest baseline— B_E and the equal-specification checker B_D in particular—by at least a prespecified minimum effect on the targeted failure classes, while keeping false rejection, indeterminacy, latency, and verification cost within prespecified budgets.

Decision trigger: If Outcome Assurance does not meet that prespecified effect against B_E within those budgets, its composition semantics add no measurable value over strong evidence-gated completion for the tested task classes, and H1 is unsupported.

Source confidence: High for the underlying engineering principles, the distinction between outcome and trajectory, and evidence-gated completion as established practice; medium for the proposition that maintaining assurance state through time and delegation adds detection value beyond it. The framework is a proposal that remains to be validated, not an experimentally established theory.

5. The Four Assurance Layers

The four layers are proposed as distinct *assurance surfaces*, not as a required four-stage workflow. One mechanism may implement checks for several layers. The decomposition is justified only if the layers establish different propositions and expose materially different failure classes.

5.1 A Common Layer Contract

Each layer instance is represented as:

$$\Lambda_k = (O_k, X_{t,k}, R_k, C_k, E_k, M_k, V_k, F_k)$$

where:

- O_k is the assurance object;
- $X_{t,k}$ is the time-indexed contextual input required to interpret that object;
- R_k is the set of applicable requirements;
- C_k is the execution-specific assurance claim;
- E_k is the admissible evidence;
- M_k is the evaluation method or warrant;
- V_k is the verdict; and
- F_k is the failure and disposition record.

The tuple is a minimum semantic contract, not a storage schema. An implementation may distribute its fields across a task ledger, policy decision point, event log, provenance graph, and verification service. It must nevertheless preserve enough information to determine what was evaluated, under which requirements, from what evidence, and with what operational consequence.

For layer k , evaluation at time t is:

$$Evaluate_k(M_k, O_k, X_{t,k}, R_k, C_k, E_k, t) \rightarrow (V_k, F_k)$$

Every layer must be able to express at least three epistemic results:

$$V_k \in \{Satisfied, NotSatisfied, Indeterminate\}$$

Section 7 defines these verdicts and keeps them distinct from operational disposition.

Requirements in R_k should be expressed as predicates or decision rules over evidence and context, not as evaluator impressions. Evidence in E_k must identify its source, observation time, scope, and relevant provenance. A model-generated explanation may be retained as evidence of what the model reported, but it is not proof that the reported reasoning caused or justifies the decision (Turpin et al., 2023).

For a task with mandatory layer instances $\mathcal{L}_T$, the framework permits an assured-completion verdict only when:

$$OutcomeAssured(T, t) \Rightarrow \bigwedge_{k \in \mathcal{L}_T} V_k = Satisfied$$

The implication is one-way. Satisfied layer verdicts are necessary under the declared assurance plan, but their correctness still depends on the adequacy of the requirements, evidence, and evaluators. Section 10 defines the separate task-level evaluation that turns these inputs into a completion verdict (see also Section 4.3).

5.2 Decision Assurance

Definition 3 — Decision Assurance

Decision Assurance is the evidence-backed determination that a consequential judgment was made from admissible and sufficiently valid inputs, under the applicable decision requirements, with uncertainty and escalation handled as required.

The assurance object is a consequential decision instance:

$$O_{DA} = (q, Alt, c, t_d)$$

where q is the question or claim decided, Alt is the set of considered alternatives when applicable, c is the selected conclusion or course, and t_d is the decision time. The object is the externally relevant judgment—not hidden model state and not unrestricted chain-of-thought.

Contextual inputs may include the task objective, facts relied upon, source provenance, current policy, risk classification, uncertainty estimates, consequence level, decision authority, and alternatives that must be considered. Applicable requirements may concern input validity, source admissibility, decision rules, required corroboration, conflicts of interest, confidence or coverage thresholds, human approval, and conditions requiring abstention or escalation.

Admissible evidence can include authoritative records, retrieved source passages, signed approvals, deterministic calculations, model outputs, evaluator results, and a trace of which evidence was used. Self-reported confidence and generated rationale may contribute to the record, but neither establishes correctness on its own. Selective-prediction research provides a formal precedent for trading coverage for controlled risk through rejection (Geifman & El-Yaniv, 2017); it does not, by itself, validate an agent’s semantic judgment or determine the correct operational escalation policy.

The verdict evaluates the declared decision requirements; its disposition may permit progression, require revision or evidence, abstain, or escalate.

Decision Assurance is related to AI evaluation, calibration, selective prediction, and software validation (Geifman & El-Yaniv, 2017; National Aeronautics and Space Administration, 2022; Tabassi, 2023). The proposed layer narrows those ideas to an individual consequential judgment embedded in a live task. Its falsifiable object is whether the judgment satisfies explicit requirements given identified evidence.

5.3 Action Assurance

Definition 4 — Action Assurance

Action Assurance is the evidence-backed determination that a proposed or attempted external side effect is admissible as specified, within applicable authority, policy, scope, and action-level safety requirements.

The assurance object is a typed side-effect request or attempt:

$$O_{AA} = (pr, o, op, par, eff, id_a)$$

where pr is the acting principal, o the target object or resource, op the operation, par its parameters, eff the requested effect, and id_a an attempt identifier.

Contextual inputs may include delegated authority, subject and object attributes, credentials, approvals, policy version, target identity, data classification, current resource state, risk, prior attempts, and task scope. Requirements may include authorization, parameter and target validation, separation of duties, approval, least privilege, rate or budget limits, idempotency keys, concurrency preconditions, rollback or compensation capability, and prohibitions on particular side effects.

This formulation intentionally exceeds a single access-control check without absorbing the rest of the task lifecycle. Attribute-based access control determines authorization from subject, object, operation, and environmental attributes (Hu et al., 2014). Usage control extends access control with obligations, conditions, attribute mutability, and ongoing evaluation (Park & Sandhu, 2004). These are strong foundations for Action Assurance, particularly where permission can change during use. They do not establish that the requested effect occurred or that the task's objective was achieved.

Admissible evidence may include a policy decision and policy version, an authority or delegation chain, credential status, approval artifacts, validated request parameters, target-state preconditions, attempt identifiers, dispatch records, and tool or service receipts. A success response is evidence that an invocation was accepted or reported as successful; its meaning depends on the service contract and does not automatically prove the external effect.

For a proposed action, *Satisfied* means admissible to attempt under the stated conditions, not “effect achieved.” For an attempted action, *Satisfied* also establishes that mandatory action-level controls were applied. In either case, the disposition may block, narrow, reauthorize, reconcile, compensate, or escalate.

Action Assurance is bounded to one proposed side effect. Execution Assurance governs how attempts compose across the task. A retry may therefore require both an Execution Assurance decision that retry is the correct lifecycle transition and a new Action Assurance verdict that this particular attempt remains admissible. Long-running transaction literature shows why compensation and partial effects require explicit treatment rather than an assumption of atomic success (Garcia-Molina & Salem, 1987).

5.4 Execution Assurance

Definition 5 — Execution Assurance

Execution Assurance is the evidence-backed determination that the task may continue, pause, remediate, or terminate while preserving applicable lifecycle invariants across sequencing, dependencies, waits, retries, resumptions, delegation, checkpoints, and intermediate state.

The assurance object is the task’s current execution state and proposed lifecycle transition:

$$O_{EA} = (G_T, \sigma_t, \ell_t, \delta_t)$$

where G_T is the task graph, σ_t is durable task state, ℓ_t is the execution and evidence ledger, and δ_t is the proposed lifecycle transition at time t .

Contextual inputs may include completed and pending nodes, dependency states, child-task status, prior attempts, checkpoints, deadlines, budgets, policy and authority versions, outstanding compensation, and evidence freshness. Requirements may include control-flow invariants, ordering constraints, dependency satisfaction, retry and deduplication rules, checkpoint integrity, deadline and resource bounds, delegation contracts, resume-time revalidation, failure propagation, and completion protocol.

Admissible evidence includes durable events, checkpoint hashes, queue and scheduler records, attempt and idempotency identifiers, service receipts, dependency observations, child-task verdicts, compensation records, and revalidation results. Workflow-pattern research demonstrates that sequencing, synchronization, choice, and multiple-instance behavior are separate control-flow concerns (van der Aalst et al., 2003). Sagas establish the need to reason explicitly about component transactions and compensating actions in long-lived work (Garcia-Molina & Salem, 1987). Runtime verification supplies monitor semantics for evaluating observed execution against specifications, including the possibility that uncertain traces do not support a sound verdict (Bartocci & Falcone, 2018; Taleb et al., 2023).

The verdict evaluates whether the proposed lifecycle transition preserves applicable execution requirements; dispositions include continue, pause, reconcile, retry, compensate, escalate, or terminate.

Execution Assurance is not synonymous with durable execution. A durable runtime asks whether computation can be reconstructed or resumed after interruption. Execution Assurance

asks whether the governed task *should* continue under the current evidence, authority, dependencies, and constraints. Durability can preserve a stale approval, an unsafe retry decision, or a corrupted intermediate state perfectly. It is enabling infrastructure, not the assurance verdict.

This layer overlaps materially with runtime verification, workflow management, transaction processing, and continuous usage control (Bartocci & Falcone, 2018; Garcia-Molina & Salem, 1987; Park & Sandhu, 2004; van der Aalst et al., 2003). Outcome Assurance does not claim a new execution-monitoring mechanism. Its proposed contribution is to preserve the execution verdict as a typed task-level assurance result that remains distinct from action admissibility and final-state verification.

5.5 End-State Assurance

Definition 6 — End-State Assurance

End-State Assurance is the evidence-backed determination that authoritative observations of the relevant external state satisfy the task’s outcome acceptance predicates at the verification time.

The assurance object is not the executor’s completion claim. It is the set of outcome acceptance predicates evaluated over observed state:

$$O_{ESA} = (P_{outcome}, \hat{S}_t)$$

where $P_{outcome} \subseteq P_T$ is the outcome portion of the declared acceptance specification (Section 10.3) and $\hat{S}_t$ is the state reconstructed from authoritative observations at time t . Process predicates are evaluated by the other layers and by Completion Verification.

Contextual inputs may include the task objective, acceptable equivalence class, tolerances, verification window, authoritative systems of record, observation scope, consequence level, and required verifier independence. Requirements may concern observation authenticity,

freshness, coverage, cross-system consistency, temporal persistence, absence of prohibited residual effects, and independence appropriate to consequence.

Admissible evidence includes direct queries to systems of record, transaction receipts whose contracts establish committed effect, cryptographically verifiable artifacts, sensor measurements, reconciled cross-system records, independent service results, and human inspection. Execution traces and executor assertions may identify what to verify, but they are not substitutes for authoritative state observations when the acceptance predicate concerns the external world.

At this layer, *Satisfied* maps to the completion-verifier result *Verified* (Section 4.4), *NotSatisfied* to *NotVerified*, and *Indeterminate* to an unresolved evidentiary state; dispositions include complete, remediate, re-observe, reconcile, escalate, or fail.

The layer has established antecedents. Hoare-style correctness separates preconditions, execution, and postconditions (Hoare, 1969). IV&V distinguishes objective evidence about conformance from evidence that the system satisfies the right need (National Aeronautics and Space Administration, 2022). Current agent benchmarks increasingly use executable evaluators over resulting environment or database state: τ -bench compares final database state with an annotated goal state (Yao et al., 2025), while OSWorld uses task-specific execution-based evaluation scripts (Xie et al., 2024). These works support the feasibility and importance of state-based verification. They do not establish the broader four-layer decomposition proposed here.

5.6 Non-Substitutability and Independent Decision Value

The decomposition is useful only if, for each layer, failures exist for which that layer’s verdict changes a task decision that the other layers’ verdicts do not determine. Representative separating cases are listed in Table 3:

Table 3*Failure Cases With Independent Layer Detection Value*

Failure case	Layer with independent detection value	Why the other layer results can still appear satisfied
A mandatory eligibility factor is ignored, but an otherwise authorized action happens to produce an acceptable result.	Decision Assurance	The action is admissible, execution is coherent, and the final predicate can hold; none establishes that the required judgment rule was followed.
A valid decision is carried out by a principal lacking authority.	Action Assurance	The rationale, orchestration, and resulting state can all be acceptable while the side effect remains impermissible.
Individually admissible calls occur in an order that violates a cross-action invariant, or a parent advances before required child completion.	Execution Assurance	Local action checks can pass for each call, while only task-level dependency and lifecycle state make the transition invalid.
Every planned step reports success, but an authoritative system of record does not contain the required change.	End-State Assurance	Decision, permission, and trace conformance do not entail the required external state.

These cases establish conceptual separability, not empirical prevalence. They must be implemented as controlled tasks and tested against baselines rather than accepted as proof by example.

Hypothesis H2 — Layer Separability

H2: Decision, Action, Execution, and End-State Assurance identify materially distinct failure classes, and each layer has independent detection or decision value for at least one consequential task condition.

For each layer k , the minimum retention condition is the existence of a controlled failure class f_k such that evaluating k changes the correct disposition while the remaining layer verdicts, under their declared objects and evidence, do not entail that change. The empirical comparison must hold task definitions and observations constant and must not label failures solely from the proposed taxonomy.

The threshold for retaining a layer is therefore:

$$\forall k \in \{DA, AA, EA, ESA\}, \quad \exists f_k: \text{IndependentDecisionValue}(k, f_k) = \text{true}$$

If no such failure class can be demonstrated for a layer, the layer should be collapsed or reframed.

5.7 Overlap, Collapse Pressure, and Present Confidence

The literature provides substantial pressure against treating the four layers as mutually exclusive technical disciplines. Runtime verification can monitor any layer whose relevant propositions and events are observable (Bartocci & Falcone, 2018). Usage control already combines authorization, obligations, changing attributes, conditions, and ongoing control, spanning much of Action and Execution Assurance (Park & Sandhu, 2004). IV&V intentionally evaluates products and processes across a lifecycle and therefore spans process and outcome questions (National Aeronautics and Space Administration, 2022). Hoare logic relates execution to postconditions rather than treating them as unrelated objects (Hoare, 1969). A sufficiently expressive assurance case or policy-and-monitoring system could consequently implement all four surfaces without adopting four subsystems.

Recent agent systems already separate some of these surfaces: TraceGrant distinguishes authorization, successful native execution, and completion of the overall task (Liao et al., 2026), and trace-based assurance work distinguishes step contracts from trace contracts and separates contract checking from governance actions that allow, rewrite, or block (Paduraru et al., 2026). These are partial analogues of the Action, Execution, and End-State distinctions. Among the works reviewed here, none types a separate Decision Assurance surface, which is consistent with that surface being the least settled boundary below.

This literature rules out the stronger claim that the framework discovers four previously unknown mechanisms or that each layer requires separate infrastructure. It does not, however, collapse the semantic decomposition. The verdict “this action is admissible,” the verdict “this

task transition preserves lifecycle invariants,” and the verdict “the required external state now holds” have different entailments even when produced by one engine. Decision Assurance remains the least settled boundary because decision requirements may sometimes be encoded completely as action policy or evaluated only through resulting behavior. H2 therefore places the burden on empirical tests to show when a separately typed decision verdict changes an operational disposition.

The present confidence assessment is:

- *High* that Action, Execution, and End-State Assurance establish non-equivalent propositions.
- *Medium* that Decision Assurance retains independent value across a sufficiently broad set of consequential tasks rather than only in regulated or explicitly rule-governed decisions.
- *Medium* that the common layer contract $\Lambda_k = (O_k, X_{t,k}, R_k, C_k, E_k, M_k, V_k, F_k)$ is expressive enough without creating redundant layer-specific fields.
- *Open* whether one or more layers should be profiles of established runtime-verification, usage-control, workflow-assurance, or IV&V frameworks rather than components of a distinct framework.

6. Contextual Assurance Requirements

The four layers in Section 5 identify *what object is being assured*: a decision, an action, an execution transition, or an end state. They do not by themselves determine what must be shown about that object in a particular task. That determination is contextual.

For each consequential node or proposed transition, the task must establish an assurance claim under the requirements applicable at that point. The resulting chain is:

Task $\rightarrow$ *Consequential Node* $\rightarrow$ *Context* $\rightarrow$ *Requirement* $\rightarrow$ *Claim* $\rightarrow$ *Evidence*
 $\rightarrow$ *Evaluation* $\rightarrow$ *Verdict* $\rightarrow$ *Disposition*

An earlier formulation of this framework described assurance through five peer properties—Validity, Traceability, Stability, Conformance, and Boundedness. They do not form a coherent set: they mix applicability, record quality, claim evaluation, controls, and population-level behavior. This framework therefore begins with requirements and claims. For assurance requirement instance i evaluated under the task context at time t , a compact runtime representation of the Outcome Assurance Requirement (OAR) instance is:

$$OAR_i(X_t) = (L_i, N_i, R_i, C_i, E_i, M_i, V_i, D_i)$$

where:

- L_i is the assurance layer;
- N_i is the consequential node or transition;
- R_i is the applicable requirement;
- C_i is the assurance claim for this execution;
- E_i is the admissible evidence;
- M_i is the evaluation method or warrant;
- V_i is the epistemic verdict; and
- D_i is the operational disposition.

All elements are interpreted under X_t , the time-indexed context that binds the task, identities, authority, policy, versions, relevant state, and other conditions needed to interpret the requirement and evidence. This is the runtime representation and evaluation of the requirement, not merely the requirement predicate itself. Like Λ_k , the tuple is a semantic record, not a storage schema. Its purpose is to prevent implementations from collapsing distinct questions: what must

be true, what is asserted, what supports that assertion, how support is evaluated, what conclusion follows, and what the task should do next.

This structure draws on established assurance-case practice rather than claiming the concepts of claim, evidence, argument, and context as novel. The Object Management Group’s Structured Assurance Case Metamodel defines assurance cases in terms of auditable claims, arguments, evidence, and contextual information, with explicit relations from evidence and subordinate claims to the claims they support (Object Management Group, 2023). Runtime and dynamic assurance-case work already applies such structures during operation and reevaluates them as systems change (Calinescu et al., 2018; Cârlan et al., 2024; Denney et al., 2015; Wei et al., 2024). Here, contextual assurance claims at consequential transitions populate the task’s assurance case and the Task Assurance State derived from it (Sections 6.8–6.9), which ultimately supports or fails to support task completion (Section 10).

6.1 Assurance Requirements

An assurance requirement states *what must be true* for a consequential node or transition to be acceptable. It is not a generic aspiration such as “be safe” or “be reliable.” It must identify a testable obligation over a defined object, context, and time.

A requirement may govern any of the four assurance layers. Examples include:

- a Decision Assurance requirement that an eligibility judgment use the policy version effective at decision time and consider every mandatory factor;
- an Action Assurance requirement that a proposed transfer remain within delegated authority, target the identified account, include a valid approval, and use an idempotency key;

- an Execution Assurance requirement that a parent task not advance until specified child tasks have completed, or that a retry occur only after the prior attempt's status is reconciled; and
- an End-State Assurance requirement that an authoritative system of record show the intended change within a specified observation window and that no prohibited residual effect remain.

Requirements may originate in task acceptance criteria, law, regulation, contract, organizational policy, delegated authority, safety analysis, service contracts, user instructions, or technical invariants. Their origin matters because it determines who may interpret, waive, revise, or retire them. A requirement record should therefore bind at least the requirement text or predicate, its source and version, its scope, its applicability rule, its criticality, and any permitted exception or waiver authority.

Policy, authorization, delegated authority, risk, and assurance remain distinct typed controls. A policy engine may return `Allow`; an authority service may confirm that a principal possesses a delegated right; and a risk process may classify expected exposure. Those results can be inputs to an assurance requirement, but none substitutes for the others. An assurance result cannot enlarge authority, a low-risk classification cannot waive a mandatory requirement, and authorization does not establish that an attempted effect occurred.

Operational bounds should be expressed in the requirement they constrain. A deadline is an Execution Assurance requirement over time. A retry limit is a transition invariant. A spending limit is a resource or authority constraint. A data-scope restriction is an Action Assurance requirement and potentially a separate authorization result. A requirement to stop when evidence

is insufficient belongs to the evaluation and disposition rules. This typed treatment replaces *Boundedness* as an undifferentiated assurance property.

6.2 Assurance Claims

An assurance claim states what is asserted to be true for this execution under an applicable requirement. Requirement and claim are related but not interchangeable.

If R_i states:

A refund above \$500 requires approval from an active finance manager before execution.

then the corresponding task-instance claim C_i may state:

Refund attempt $A-184$ for \$725 was approved before dispatch by principal $P-17$, who was an active finance manager with approval authority for this account at the approval and dispatch times.

The requirement is reusable and normative. The claim is instance-specific and truth-valued. It binds the rule to an identified action, actors, objects, versions, and times. A single requirement may produce many claims across a task, and one consequential node may require several claims at different layers.

Claims must be externally evaluable. They should concern observable decisions, declared inputs, authority, actions, task state, or resulting external state—not hidden model activations or unrestricted chain-of-thought. Generated explanations can be unfaithful to the factors that produced an answer (Turpin et al., 2023). The assurance record should instead preserve the applicable rule, external inputs and sources, authority and approval artifacts, action and transition records, observations, and evaluator results needed to assess the claim.

A claim also has a temporal scope. “Approval is current,” “the dependency is complete,” and “the target state holds” are assertions at specified times or over specified intervals. A satisfied claim at t_0 cannot be reused at t_1 unless the requirement and evaluation method permit reuse and the relevant conditions remain satisfied.

6.3 Context and Applicability

Context determines why a requirement, authority, policy, approval, source, or evidence item applies *here and now*. It is upstream of the verdict rather than a universal assurance property.

The time-indexed context X_t may include:

- task identity, objective, acceptance predicates, and consequence level;
- the consequential node and proposed transition;
- subject, object, resource, tenant, jurisdiction, and organizational boundary;
- policy, requirement, model, tool, schema, and knowledge versions;
- delegated authority, credential, approval, and separation-of-duties state;
- relevant external and intermediate state;
- deadlines, budgets, retry counts, prior attempts, and outstanding compensation;
- evidence observation times, freshness windows, and systems of record; and
- assumptions, exceptions, and waiver state.

Applicability is the act of binding general rules and artifacts to this context. An approval may be authentic yet apply to another account. A credential may be unexpired yet prohibit the requested purpose. A policy may be current but not govern the relevant jurisdiction. A source may be authoritative for identity but not for settlement status. A previous observation may be accurate but too stale for a transition whose preconditions can change rapidly.

Distributed-systems and security standards provide familiar precedents. Leases make grants explicitly time-bounded (Gray & Cheriton, 1989). Public Key Infrastructure using X.509 (PKIX) path validation evaluates not only cryptographic signatures but also validity intervals, usage constraints, policies, and revocation state (Cooper et al., 2008). W3C PROV uses *validity* in yet another sense: the logical consistency of a provenance history (Cheney et al., 2013). The framework therefore avoids a generic *Validity* property and records the specific applicability predicates that must hold.

Applicability may change without changing the artifact itself. A task that waits, resumes, crosses an organizational boundary, changes executor, or delegates work must reevaluate context-sensitive inputs when their continued applicability is not guaranteed. This is why durable storage of an earlier *Satisfied* verdict is not enough: the task must also preserve the conditions under which that verdict was established.

6.4 Evidence

Evidence consists of claim-relevant observations or artifacts, including authoritative records, approvals, service receipts, database or sensor observations, task events, calculations, independent evaluations, human findings, and cryptographic proofs. Its record must preserve the attributes needed to determine applicability and admissibility. Provenance and traceability describe how evidence was produced and related; they do not establish the substantive claim (Hasan et al., 2009; Meliou et al., 2010; Moreau & Missier, 2013):

$$\text{Traceable}(E_i, C_i) \not\Rightarrow C_i$$

When traceability is mandatory, it becomes an explicit requirement with its own claim and verdict. Section 7 provides the canonical treatment of evidence, admissibility, provenance, and integrity.

6.5 Evaluation and Evidentiary Sufficiency

Evidence establishes a claim only through an explicit method or warrant. Evaluation may be deterministic, formal, statistical, semantic, external-system based, human, or composite; the selection rule is to use the least probabilistic evaluator capable of soundly evaluating the requirement. Conformance is the result of that evaluation, not a separate peer property:

$$\text{Conforms}(N_i, R_i, E_i, X_t) \Leftrightarrow \text{Evaluate}(M_i, C_i, E_i, X_t) = \text{Satisfied}$$

This relation cannot establish unstated needs or conformance outside the method's declared coverage (Gray et al., n.d.). Section 7 defines evidentiary sufficiency, evaluator competence, and the Verification Contract that binds claims, evidence, methods, and trust assumptions.

6.6 Assurance Verdicts

Every requirement instance supports three epistemic verdicts:

$$V_i \in \{\text{Satisfied}, \text{NotSatisfied}, \text{Indeterminate}\}$$

Section 7 is the canonical definition of *Satisfied*, *NotSatisfied*, and *Indeterminate*, including evidentiary insufficiency and verdict scope. These claim-level verdicts do not establish population-level *Outcome Reliability*.

6.7 Disposition

A verdict states what the evidence establishes; a disposition states what the task does next. Section 7 defines this distinction and its operational consequences; the OAR_i record keeps both V_i and D_i in the dynamic assurance case (Section 6.8).

6.8 Dynamic Assurance Cases

An agentic task does not receive one static assurance certificate at its beginning or end. It dynamically constructs an assurance case as consequential work occurs.

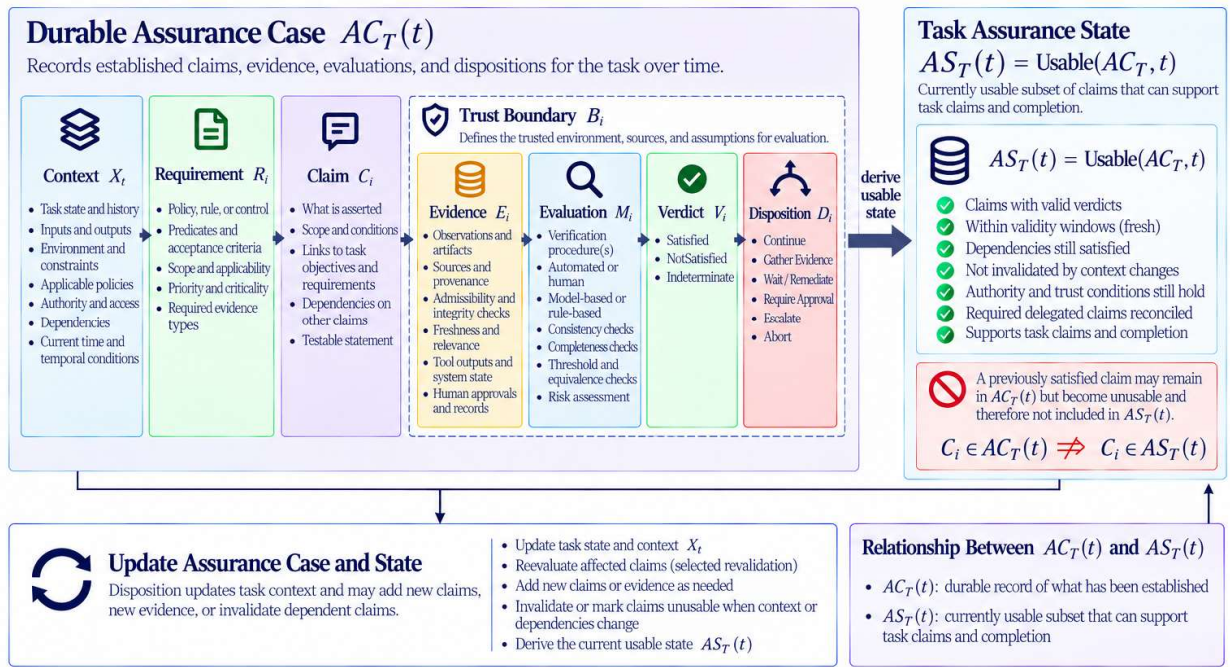
At node N_i , the task resolves applicable requirements from X_t , instantiates claims, obtains admissible evidence, applies evaluation methods, records verdicts, and follows dispositions. The resulting objects become part of the durable task-assurance record. Later nodes may depend on them, refine them, contradict them, or require them to be reevaluated.

Figure 2 shows the dynamic assurance case and the Task Assurance State derived from it.

Figure 2

The Dynamic Assurance Case

A task's assurance record is durable over time, but only a subset of established claims remains usable at any moment. The Dynamic Assurance Case records what has been established in the durable assurance case and derives the current Task Assurance State from the claims that remain usable under current context, dependencies, policy, authority, and trust conditions.



Let the assurance record at time t be:

$$AC_T(t) = \{OAR_1(X_{t_1}), OAR_2(X_{t_2}), \dots, OAR_n(X_t)\}$$

This set is not merely an event log. It preserves the inferential structure needed to determine which claims were established, from which evidence, under which requirements, methods, and contexts, and with what operational consequences. Provenance records how artifacts and activities relate; the assurance case records what those relationships were taken to

establish and why. Durable workflow research shows that execution and recovery semantics can be preserved across heterogeneous tasks (Zhuang et al., 2023); the assurance record adds the contextual claim and warrant needed to decide whether a recovered transition should proceed.

The case evolves in at least four ways:

1. **Accumulation.** New consequential nodes create new requirement and claim instances. A final completion claim may depend on satisfied claims across Decision, Action, Execution, and End-State Assurance.
2. **Refinement.** A broad claim may be decomposed into subordinate claims, or an Indeterminate result may trigger collection of evidence that supports a later reevaluation.
3. **Invalidation.** New evidence may contradict an earlier observation, an action may fail after acceptance, or a requirement may be revised so that a prior verdict can no longer support continuation.
4. **Reevaluation.** Resume, delay, delegation, policy change, authority change, model or tool change, evidence expiry, object-state change, or transfer across an organizational boundary may require a claim to be evaluated again under a new X_t .

Reuse, delegation, and completion all consume this evolving case. The central result is: An agentic task is assured by dynamically constructing and reevaluating a contextual assurance case for its consequential transitions, not by assigning the agent or task a fixed set of generic assurance properties.

As Section 2.6 notes, dynamic assurance cases are an adopted foundation, not this paper's contribution. What this paper adds is the use of the task's assurance case to derive the current assurance state on which completion depends.

6.9 Task Assurance State

The assurance case $AC_T(t)$ is a durable record. It retains what was established, from which evidence, and under which context, including claims that no longer hold. That history is necessary for audit and assurance replay (Section 8.13), but it is not what the task may rely on now. The framework therefore distinguishes the case from the *Task Assurance State*:

$$AS_T(t) = Usable(AC_T, t) = \{OAR_i \in AC_T(t) : Usable(C_i, t)\}$$

where $Usable(C_i, t)$ is the claim-usability predicate of Section 8.11: the claim's verdict is `Satisfied`, t lies within its validity window, its dependencies remain valid, and none of its invalidation triggers has fired. For claims returned by delegated work, usability additionally requires parent reconciliation (Section 9.8). $AC_T(t)$ records what has been established and retained over the task lifecycle; $AS_T(t)$ represents what may currently be relied upon to support task claims at time t .

Writing $C_i \in AC_T(t)$ for the claim of a record $OAR_i \in AC_T(t)$, membership in the case does not imply membership in the state:

$$C_i \in AC_T(t) \not\Rightarrow C_i \in AS_T(t)$$

A previously satisfied claim may thus remain in $AC_T(t)$ while failing any of these usability conditions. Claims whose current verdict is `NotSatisfied` or `Indeterminate` remain in $AC_T(t)$ and inform disposition, but they are not in $AS_T(t)$.

The Task Assurance State connects the remaining sections. Section 8 defines how it changes over time, Section 9 defines how delegated assurance may enter it, Section 10 evaluates completion against it, and Section 11 defines reliability over the completion verdict it supports.

Research Hypothesis

H3: For an individual agentic task, task-scoped contextual assurance claims—each supported by admissible evidence and an explicit evaluation method—improve diagnostic precision, disposition accuracy, or safe reuse under changed context relative to a simpler fixed-property representation of the same task.

Dynamic, context-bound assurance claims are established in runtime assurance-case work (Calinescu et al., 2018; Cărlan et al., 2024; Denney et al., 2015; Wei et al., 2024). H3 does not test whether such claims are possible, only whether scoping them to the consequential transitions of a task improves these decisions. Section 13.4 specifies the test. H3 survives only if contextual claims materially improve diagnostic precision, disposition accuracy, or safe reuse under changed context; otherwise the simpler fixed-property representation should be used.

7. Evidence and Verification

Outcome Assurance requires more than recording that an agent acted or preserving a trace of its execution. For each consequential claim, the system must establish what evidence is admissible, how that evidence is evaluated, and what assumptions underlie the resulting verdict.

This requirement follows directly from established assurance practice. Systems and software assurance distinguishes an assurance *claim* from the *evidence* supporting that claim and from the argument connecting the two. ISO/IEC/IEEE 15026 identifies claims, argumentation, and evidence as foundational assurance concepts. Provenance standards provide mechanisms for reconstructing how evidence was produced, while independent verification and validation emphasizes objective evidence and independence from the process being assessed. Outcome Assurance applies these principles dynamically to individual agentic tasks rather than treating assurance solely as a static property of a system before deployment (International Organization

for Standardization, 2022, 2025; Moreau & Missier, 2013; National Aeronautics and Space Administration, 2022).

For a claim C_i , Outcome Assurance therefore requires an explicit relationship among the claim, its admissible evidence, the method used to evaluate that evidence, and the trust assumptions under which the verdict is accepted. A useful abstraction, which this paper calls a *Verification Contract*, is:

$$VC_i = (C_i, E_i, M_i, B_i)$$

where:

- C_i is the assurance claim;
- E_i is the admissible evidence set;
- M_i is the evaluation method; and
- B_i is the verifier trust boundary and associated assumptions.

The contract is specified before evaluation. Evaluating it within the current task context X_t produces the verdict V_i , which is recorded with the contract but is not part of its specification:

$$V_i = Evaluate(C_i, E_i, M_i, B_i, X_t)$$

The tuple is the contract's core; Section 7.14 lists the fuller set of fields an implementation may specify. The term denotes an implementation abstraction rather than a new theory of evidence: the underlying claim-evidence-argument structure is established assurance practice.

7.1 Evidence Is Claim-Relative

Evidence has no universal assurance value independent of the proposition it is intended to support.

An HTTP 200 OK response may provide strong evidence that a service accepted a request. It is not necessarily sufficient evidence that the requested real-world effect occurred. A payment API may acknowledge a request while settlement remains pending. A database service may accept an update while a downstream system remains inconsistent. A model may report that it performed a task while the state defining successful completion does not exist.

Evidence must therefore be evaluated relative to a specific assurance claim:

$$\text{Supports}(E, C \mid X, M)$$

where E is the evidence, C is the claim, X is the applicable context, and M is the evaluation method.

NASA's software assurance practice similarly treats objective evidence as documented facts demonstrating that a particular required activity or property has been confirmed; the appropriate evidence depends on what is being established. Independent verification and validation then uses such evidence to increase confidence that critical systems satisfy their requirements and intended purpose (National Aeronautics and Space Administration, 2022).

Outcome Assurance therefore makes no general claim that more evidence produces greater assurance. Ten weak observations may remain insufficient for a claim that one authoritative observation could establish. Conversely, one authoritative source may be insufficient where corroboration or separation of duties is required.

Evidence sufficiency is a property of the relationship:

$$(C, E, M, X)$$

not of evidence quantity alone.

7.2 Evidence Admissibility

Before evidence can support an assurance claim, it must be admissible for that claim under the current context.

An evidence item can be represented abstractly as:

$$e = (\text{source}, \text{observation}, \text{time}, \text{scope}, \text{provenance}, \text{integrity})$$

with an admissibility predicate:

$$\text{Admissible}(e, C, X_t)$$

Admissibility conditions may include:

- whether the source is authoritative for the asserted fact;
- whether the observation concerns the correct entity, transaction, or task;
- whether the evidence remains temporally valid;
- whether its provenance is available and internally consistent;
- whether its integrity can be established;
- whether it was created under the required authority or process;
- whether the evidence is sufficiently independent from the actor making the claim;
- whether the evidence falls within the validated scope of the evaluation method;

and

- whether required corroborating evidence is present.

Admissibility becomes particularly important in long-running tasks. Evidence that was admissible at time t_0 may cease to be admissible at t_1 because an approval expires, policy changes, an authoritative source is superseded, or the task context changes.

Thus:

$$\text{Admissible}(e, C, X_{t_0}) \not\Rightarrow \text{Admissible}(e, C, X_{t_1})$$

An asynchronous task must therefore preserve not only evidence but also the conditions under which that evidence was considered admissible.

7.3 Provenance and Traceability

Provenance is a necessary component of evidence qualification but is not equivalent to proof of the underlying claim.

The W3C PROV model represents entities, activities, agents, derivations, associations, delegation relationships, usage, generation, and temporal ordering. Its formal constraints allow a provenance instance to be evaluated for internal consistency and provide a basis for reasoning about how an artifact or observation was produced (Cheney et al., 2013; Moreau & Missier, 2013).

For Outcome Assurance, provenance can establish relationships such as:

$$Evidence \leftarrow ObservationActivity \leftarrow AuthoritativeSystem$$

or:

$$Decision \leftarrow Evidence \leftarrow SourceVersion$$

and, in delegated work:

$$ChildEvidence \leftarrow ChildTask \leftarrow ParentDelegation$$

This information is essential for reconstruction, audit, applicability analysis, and causal investigation.

However:

$$ValidProvenance(E) \not\Rightarrow True(E)$$

A provenance graph may provide a consistent account of how a false measurement was generated. Similarly, complete traceability can faithfully describe a decision that was wrong.

Evidence traceability should therefore be treated as a structural requirement on evidence, not as a substitute for evaluation of the assurance claim.

7.4 Evidence Integrity and Evidentiary Truth

Cryptographic signatures, hashes, append-only logs, and trusted execution mechanisms can establish properties such as:

- the identity of an issuer;
- that an artifact has not been altered;
- that evidence is bound to a particular task or context;
- that events occurred in an authenticated sequence; or
- that a later artifact commits to an earlier evidence set.

These mechanisms can be valuable components of an assurance architecture, but they establish properties of the evidence artifact rather than the truth of the proposition it contains.

Formally:

$$Integrity(e) \not\Rightarrow Accuracy(e)$$

A signed incorrect statement remains incorrect.

Outcome Assurance therefore distinguishes *evidence integrity* from *claim satisfaction*.

This distinction is especially important if assurance claims are later materialized as signed portable assertions. Such an artifact can prove that a specified verifier asserted a claim under specified conditions. It cannot, by cryptographic construction alone, establish that the claim was objectively true.

7.5 Evaluation Methods

Evidence becomes assurance evidence only through an evaluation method that connects the evidence to the claim.

The evaluation method M_i specifies how admissible evidence is interpreted and what constitutes sufficient support for a verdict. Depending on the claim, the method may be:

- a deterministic predicate;
- a calculation;
- a schema or structural validation;
- a formal policy or rule evaluation;
- an observation of an authoritative external system;
- a temporal or state-transition predicate;
- a statistical test;
- a calibrated threshold;
- a semantic evaluator;
- domain-expert judgment;
- human approval; or
- a composite of multiple methods.

No single evaluator type is appropriate for all claims.

Where a claim can be established using deterministic observation, adding a probabilistic evaluator generally introduces unnecessary uncertainty. Where the claim itself is semantic or judgment-dependent, deterministic checks may be incapable of evaluating the actual requirement.

This suggests the following design principle:

Use the least probabilistic evaluation mechanism capable of evaluating the assurance claim that actually matters.

Here, *least probabilistic* means the evaluator class whose error on the claim that matters is smallest and best characterized. A deterministic evaluator qualifies only when it evaluates that claim rather than a proxy for it.

A deterministic test of an inadequate proxy is not superior to a probabilistic evaluation of the correct claim.

For example, testing:

apiResponse.status = 200

deterministically does not establish:

customerReceivedRefund = true

if those propositions are not equivalent.

7.6 Evidentiary Sufficiency

An evaluation method must also define what constitutes sufficient evidence.

Evidence may fail to be sufficient because it is:

- incomplete;
- stale;
- outside the evaluator's validated scope;
- contradictory;
- insufficiently independent;
- produced by an untrusted source;
- missing required corroboration;
- subject to excessive measurement uncertainty; or
- incapable of distinguishing satisfaction from non-satisfaction.

Established assurance-case practice treats this relationship between claim, evidence, context, and argument as central to justified confidence. ISO/IEC/IEEE 15026 explicitly grounds assurance in supporting claims through argumentation and evidence rather than in evidence accumulation alone (International Organization for Standardization, 2022).

Outcome Assurance does not assign a universal numeric measure to evidentiary sufficiency. The sufficiency conditions should instead be defined by the Verification Contract for the claim being evaluated.

For one claim, a single authoritative system-of-record query may be sufficient. For another, multiple independent observations may be required. For a third, current evidence may be incapable of supporting any sound conclusion.

7.7 Assurance Verdicts

A binary success/failure model is insufficient for runtime assurance because available evidence may not permit a sound conclusion.

Outcome Assurance therefore uses at least three assurance verdicts:

$$V_i \in \{Satisfied, NotSatisfied, Indeterminate\}$$

Satisfied means the available admissible evidence meets the evaluation conditions required to establish the claim.

NotSatisfied means the available evidence establishes that the required claim does not hold.

Indeterminate means the available evidence is insufficient to soundly establish either satisfaction or non-satisfaction.

Indeterminate is not equivalent to failure.

Examples include:

- an authoritative external system is temporarily unavailable;
- settlement has not yet reached a state that can be observed;
- two admissible authoritative sources conflict;
- required corroborating evidence has not arrived;
- the evaluator is outside its validated scope;
- an approval's continued applicability cannot yet be established; or
- an asynchronous external process remains pending.

This distinction is consistent with the broader verification principle that an incomplete observation may not support a sound binary verdict (Bauer et al., 2011; Taleb et al., 2023). It also prevents an agentic system from converting uncertainty into false confidence simply because execution requires a next step.

When several verdicts must hold together, as when a claim depends on subordinate claims, they combine by the strong Kleene conjunction $\sqcap$ defined in Table 2. This matches the three-valued verdict domain used in finite-trace runtime verification (Bauer et al., 2011).

7.8 Verdict and Disposition

A verdict describes the epistemic state of an assurance claim. A disposition determines what the system does next.

Let:

$$D_i = f(V_i, R_i, X_t)$$

where R_i is the applicable assurance requirement and X_t the current context.

Possible dispositions include:

- Execute;
- Continue;

- Wait;
- GatherEvidence;
- RetryObservation;
- Remediate;
- RequireApproval;
- Escalate;
- Deny;
- Abort; or
- FailClosed.

For example:

Indeterminate $\rightarrow$ *Wait*

may be correct for an eventually consistent external system.

The same verdict may produce:

Indeterminate $\rightarrow$ *RequireApproval*

for a time-sensitive consequential decision.

Separating verdict from disposition prevents operational policy from contaminating the assurance determination itself, for example by silently rewriting an `Indeterminate` result as `Satisfied` because the task is inconvenient to delay.

7.9 Conflicting Evidence

Outcome Assurance must also define how conflicting admissible evidence is treated.

Suppose:

Supports($e_1, C \mid X, M$)

while:

$$\text{Supports}(e_2, \neg C \mid X, M)$$

and both evidence items are admissible.

The system cannot soundly infer:

$$V = \text{Satisfied}$$

unless its evaluation method contains an explicit conflict-resolution rule.

Such rules may use:

- authoritative-source precedence;
- temporal precedence;
- jurisdictional precedence;
- source specialization;
- corroboration;
- additional observation;
- reconciliation procedures;
- expert review; or
- explicit escalation.

Provenance can identify the origins and relationships of the conflicting observations but does not by itself determine which should govern (Cheney et al., 2013; Moreau & Missier, 2013).

Absent a justified resolution mechanism, the appropriate verdict is normally

`Indeterminate` rather than averaging contradictory evidence or allowing the executing agent to select whichever observation favors completion.

7.10 Temporal Evidence and Eventual Consistency

Evidence can also be incomplete because the world has not yet converged to an observable final state.

Suppose an action occurs at t_0 , but the corresponding authoritative state is only expected to appear within thirty minutes.

Then:

$$\neg \text{Observed}(\text{ExpectedState}, t_0)$$

does not imply `NotSatisfied` if the requirement is temporal:

$$\text{Eventually}_{[0,30m]}(\text{ExpectedState})$$

An Outcome Assurance evaluator may therefore produce:

$$\text{Indeterminate} \rightarrow \text{Wait} \rightarrow \text{Reobserve}$$

until either `ExpectedState` is observed within the permitted interval or the temporal requirement expires.

This behavior is particularly relevant to asynchronous agents interacting with eventually consistent distributed systems. It also reinforces why assurance verdicts and task lifecycle state must remain distinct.

7.11 Verification Independence

For consequential claims, assurance may require evidence or evaluation that is sufficiently independent of the executing agent.

Independence should not be defined simply as using a second model.

NASA IV&V distinguishes technical, managerial, and financial independence because the purpose of independent verification is to prevent the producing process from being the sole evaluator of its own acceptability. Outcome Assurance applies the underlying principle at

runtime, without requiring the same organizational structure for every task (National Aeronautics and Space Administration, 2022).

A verifier is considered sufficiently independent when the executing agent cannot unilaterally:

- determine the authoritative evidence;
- alter the applicable acceptance predicate;
- modify the evaluation method;
- change the required threshold;
- fabricate the external state being observed; or
- issue the final assurance verdict.

These conditions group into at least three dimensions of independence.

Evidence independence concerns whether the actor making the claim controls the evidence intended to verify it.

Evaluation independence concerns whether that actor can alter the logic or method used to determine satisfaction.

Verdict independence concerns whether the actor can issue or override its own acceptance decision.

A deterministic local calculation may be sufficient for a low-consequence arithmetic claim. Confirmation of a financial settlement may require observation from an external system of record. A regulated or safety-critical claim may require independent human or organizational authority.

Thus verifier independence is contextual rather than binary.

AI-control research studies the same separation in adversarial form, using a less capable trusted model to monitor a more capable untrusted one (Greenblatt et al., 2024). Recent agent completion gates likewise separate the proposing agent from a read-only or trusted verifier (Nguyen & Tran, 2026; Zhang et al., 2026).

7.12 Protecting the Verification Channel

Independence also requires protecting the verification mechanism from the actor being evaluated.

If executing agent Ex has modification authority over the acceptance predicates P_T , evaluation method M , or authoritative evidence E_a , then the verification process contains a direct optimization path:

$$Ex \rightarrow \{P_T, M, E_a\}$$

A stronger architecture constrains that relationship:

$$WriteAccess_{Ex}(P_T) = false$$

$$WriteAccess_{Ex}(M) = false$$

and, where feasible:

$$WriteAccess_{Ex}(E_a) = false$$

The executing agent may generate candidate evidence, but it should not have unilateral control over the authoritative observations used to certify its own successful completion. This is the runtime equivalent of avoiding self-certification.

It is especially important for capable agents because optimization against an evaluator can produce apparent task success without satisfaction of the underlying objective; documented agent evaluations show models exploiting scoring gaps in this way (Hamin & Edelman, 2025).

The correct control is therefore not merely a stronger evaluator but a protected evaluation channel whose trust assumptions are explicit.

7.13 Verifier Trust Boundaries

Independent verification does not eliminate the need to trust something.

Every verification chain eventually terminates in assumptions about components such as:

- an authoritative system of record;
- an approved rule engine;
- a calibrated measurement device;
- a trusted cryptographic root;
- a validated verifier implementation;
- an accepted external service;
- a qualified human reviewer; or
- a formal system and its axioms.

Trying to verify every verifier recursively produces an infinite regress rather than greater practical assurance.

Outcome Assurance instead requires the trust boundary to be explicit.

An evaluation method can therefore be represented with metadata such as:

$$M_i = (\text{method}, \text{trustRoot}, \text{scope}, \text{version}, \text{limitations})$$

For example:

method:

Query purchase-order system of record

trust root:

Production procurement database

scope:

Purchase-order existence and approval status

version:

verifier-policy-7

limitations:

Does not establish counterparty receipt

This allows a later reviewer to understand precisely what the assurance verdict does and does not establish.

The approach is consistent with the broader assurance principle that assurance provides *justified confidence*, not absolute certainty. ISO/IEC/IEEE 15026 bases assurance on claims supported through argumentation and evidence rather than on proof of universal correctness (International Organization for Standardization, 2022, 2025).

7.14 Verification Contracts

The concepts above can be collected into a runtime *Verification Contract* associated with an assurance claim.

A Verification Contract may specify:

claim

applicable context

admissible evidence classes

authoritative evidence sources

provenance requirements

evaluation method

verifier identity and version

trust assumptions
 required independence
 threshold or acceptance rule
 validity interval
 permitted verdicts
 indeterminate disposition
 failure disposition

The Verification Contract does not prescribe a specific technology. It may be implemented through policy, schema, code, rule systems, workflow definitions, formal specifications, or combinations of these.

Its function is to make explicit what would otherwise remain implicit:

What evidence would cause this system to believe that this particular claim has been established?

7.15 Relationship to Outcome Assurance

Sections 5 and 6 established the assurance layer and the contextual assurance requirement. This section completes the claim-evaluation mechanism:

$$\textit{AssuranceLayer} \rightarrow \textit{Requirement} \rightarrow \textit{Claim} \rightarrow \textit{VerificationContract} \rightarrow \textit{Verdict} \\ \rightarrow \textit{Disposition}$$

The Verification Contract converts observations into a bounded, inspectable claim by recording the applicable requirement, admissible evidence, evaluation method, trust assumptions, verdict, and disposition. Logs, provenance, and policy decisions can supply inputs; the contract specifies why the available evidence was sufficient—or insufficient—to establish the assurance claim governing the task transition.

Section 7 Proposition

The section supports the following proposition:

A defensible task-level assurance verdict requires a context-specific verification contract that identifies the claim, admissible evidence, evaluation method, trust assumptions, and required independence before determining whether that claim is Satisfied, NotSatisfied, or Indeterminate.

The proposition deliberately stops short of claiming that the resulting verdict is objectively infallible. Assurance terminates at explicit trust assumptions. Its purpose is to make those assumptions and the evidentiary basis for the verdict visible, bounded, and reviewable.

Research Hypothesis

H4: Context-specific Verification Contracts that identify the claim, admissible evidence, evaluation method, trust assumptions, and required independence reduce false assurance compared with agent self-verification or evaluation over unprotected, non-authoritative evidence.

The claim–evidence–argument structure and contract-based specification underlying the Verification Contract are established (Benveniste et al., 2018; Object Management Group, 2023). H4 tests whether making their binding explicit for each claim—including the trust boundary and required independence—reduces false assurance. Section 13.5 specifies the test. H4 survives only if the stronger contract conditions reduce false assurance by a prespecified material amount without converting most cases to rejection or Indeterminate.

8. Assurance Through Time

Outcome Assurance is not static. A claim that was adequately supported at one point in a task may no longer be usable later in the same task.

This is particularly important for agentic work because execution may be suspended for minutes, hours, or days while awaiting an approval, external event, scheduled time, human response, rate-limit reset, dependency, or delegated subtask, as in long-running monitoring-agent

settings (Maldaner et al., 2026). During that interval, the computational state may remain durable while the institutional and evidentiary conditions supporting prior assurance claims change.

That is:

$$Satisfied(C, t_0) \not\Rightarrow Satisfied(C, t_1)$$

A workflow can be restored to exactly the point at which it stopped without establishing that it remains justified to continue from that point.

Outcome Assurance therefore distinguishes *durable execution state* from *durable assurance state*.

A durable workflow answers:

Where did execution stop, and how can it continue?

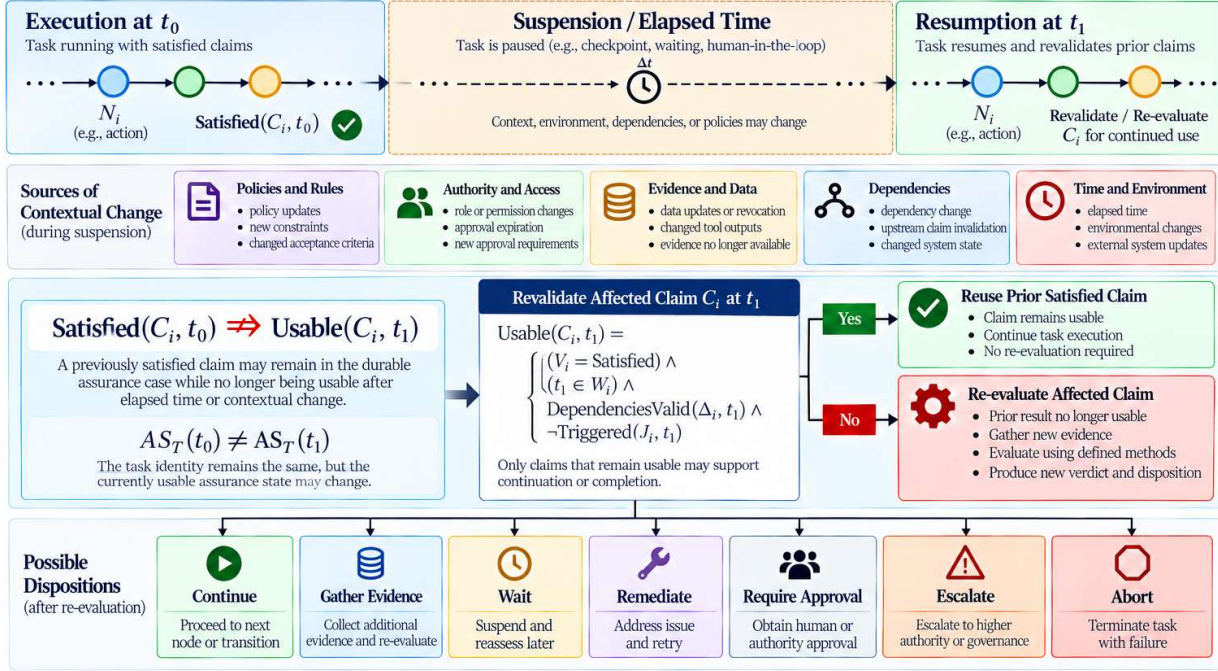
Temporal assurance adds:

Which claims justified continuation at that point, and do those claims remain usable now?

Figure 3 illustrates assurance through time.

Figure 3*Assurance Through Time*

A task can be suspended and later resumed, but prior assurance results do not automatically remain usable. Changes in time, context, policies, authority, evidence, or dependencies can invalidate previously satisfied claims. Upon resumption, the task selectively revalidates affected claims and re-evaluates those that are no longer usable before continuing execution.

**8.1 Assurance Claims Are Temporally Scoped**

An assurance verdict is established under a particular context.

Suppose a purchase task is evaluated at time t_0 using:

- policy version Pol_3 ;
- delegated authority $Auth_7$;
- approval Ap_{19} ;
- transaction amount of \$4,000;
- supplier Sup_8 ; and
- supporting evidence E .

A claim such as:

$$PurchaseAuthorized = Satisfied$$

is therefore conditional on those dependencies.

A more accurate representation is:

$$C = \text{PurchaseAuthorized} \mid \{Pol_3, Auth_7, Ap_{19}, Amount = 4000, Supplier = Sup_8, E\}$$

If the task resumes later, one or more of those dependencies may no longer hold.

For example:

- the approval may have expired or been revoked;
- the amount may have changed;
- the supplier may have become prohibited;
- authority may have been withdrawn;
- policy may have changed;
- supporting evidence may have become stale;
- an external side effect may have occurred while the task was suspended.

The earlier verdict remains historically valid as evidence of what was established at t_0 . It is not automatically a valid basis for execution at t_1 . OAuth standards provide a concrete analogue: credentials carry scope and lifetime, yet authorization may be revoked before nominal expiration (Hardt, 2012; Lodderstedt et al., 2013).

Outcome Assurance therefore requires a distinction between:

historical satisfaction — the claim was established under the context then in force; and

current usability — the claim remains an acceptable basis for the next governed transition.

8.2 Expiration Alone Is Insufficient

A simple implementation might associate every assurance claim with a time-to-live. That is useful but incomplete.

Consider an approval that is valid for 24 hours. Two hours after it is issued, the transaction amount changes from \$4,000 to \$40,000.

The approval has not expired temporally, but the conditions under which it supported the original claim have materially changed.

Similarly, an authority grant may remain within its nominal validity window while being explicitly revoked. A policy may remain versioned and available while becoming inapplicable because the task moves to another jurisdiction or affected population.

Claim reuse must therefore consider both time and semantic dependencies.

A general form is:

$$\begin{aligned} & \textit{Reusable}(C, t) \\ &= \textit{TemporalValidity}(C, t) \wedge \textit{DependencyValidity}(C, t) \wedge \textit{ContextCompatibility}(C, t) \end{aligned}$$

This is stronger than an expiration check:

$$\textit{now} < \textit{expiresAt}$$

8.3 Assurance Dependency Graphs

To determine whether previously satisfied claims remain reusable, an Outcome Assurance system can explicitly record the conditions on which each claim depends.

For a claim C_i :

$$\textit{Deps}(C_i) = \{d_1, d_2, \dots, d_n\}$$

For example:

Claim:

PurchaseAuthorized

Dependencies:

Policy Pol3

Authority Auth7

Approval Ap19

Amount = \$4,000

Supplier = Sup8

Account = AC2

A change to one of those dependencies can trigger reevaluation of only the claims that rely on it.

If *Changed* represents the set of materially changed contextual elements, then:

$$\text{InvalidatedClaims} = \{C \mid \text{Deps}(C) \cap \text{Changed} \neq \emptyset\}$$

This produces a dependency-aware revalidation model:

Context Change

↓

Identify affected dependencies

↓

Identify dependent assurance claims

↓

Invalidate or re-evaluate those claims

↓

Continue only when required claims are usable

This is preferable to either extreme:

- trusting every prior verdict indefinitely; or
- reevaluating every assurance claim after every suspension.

The dependency graph is an optimization, not a relaxation of the assurance boundary.

Where dependency coverage cannot be established confidently, the system should broaden revalidation rather than assume reuse is safe.

8.4 Assurance Checkpoints

Long-running work requires checkpoints. For Outcome Assurance, checkpointing must preserve more than execution position.

An assurance checkpoint may include:

- task identity and version;
- current execution position;
- satisfied, unsatisfied, and indeterminate claims;
- dependencies of reusable claims;
- evidence references;
- evidence validity conditions;
- policy and knowledge versions;
- authority and delegation state;
- approval status;
- known external side effects;
- side effects whose outcome is uncertain;
- outstanding requirements;
- pending child tasks;
- retry state;
- idempotency identifiers; and
- conditions that require reevaluation before continuation.

The purpose of the checkpoint is not merely to answer:

What should execute next?

It must also support:

What did the system previously establish, why did it establish it, and what must still be true for that result to remain usable?

The difference between these questions matters most after failure recovery.

Computational recovery may recreate the workflow state exactly while some assurance conditions have become stale or invalid (Temporal Technologies, n.d.).

8.5 Re-Entry Assurance

A task resuming from a waiting, queued, failed, or suspended state should not necessarily return directly to execution.

Instead, the transition back into active work can pass through a *Re-entry Assurance Gate*.

Executing

↓

Waiting / Queued

↓

Resume Event

↓

Re-entry Assurance

↓

Continue | Revalidate | Wait | Escalate | Abort

The gate determines what must be established before execution is permitted to continue.

At minimum, it can determine:

1. which assurance claims the next transition depends on;
2. which of those claims remain reusable;
3. which relevant dependencies changed during suspension;
4. whether required evidence remains admissible;
5. whether approvals remain valid;

6. whether authority remains applicable;
7. whether external side effects occurred during suspension;
8. whether child-task state changed; and
9. whether the next proposed transition remains permitted under current context.

The objective is not to replay the entire assurance process mechanically.

Instead:

$$\text{Resume}(T, t_1) \Rightarrow \text{Revalidate}(\text{RequiredClaims}(T, t_1))$$

rather than:

$$\text{Resume}(T, t_1) \Rightarrow \text{Revalidate}(\text{AllHistoricalClaims}(T))$$

8.6 Unknown Side Effects

Retries introduce a particularly difficult temporal state: the system may not know whether an external action actually occurred (Featonby, 2021).

Suppose an agent submits a \$500 refund and loses its network connection before receiving the response.

The observable client state is not necessarily `Failed`. It may be `OutcomeUnknown`.

A useful action-state model therefore distinguishes at least:

$$\text{ActionState} \in \{\text{NotIssued}, \text{Succeeded}, \text{Failed}, \text{OutcomeUnknown}\}$$

If the original request actually succeeded, a retry could create a duplicate side effect. If it did not succeed, refusing to retry could leave the task incomplete.

When an authoritative observation is available, the preferred transition is therefore:

$$\text{OutcomeUnknown} \rightarrow \text{ObserveExternalState} \rightarrow \text{RetryOrContinue}$$

rather than:

$$\text{OutcomeUnknown} \rightarrow \text{Retry}$$

Unknown side-effect state is consequently an assurance condition, not merely an execution exception.

8.7 Idempotency and Retries

Idempotency reduces the risk associated with uncertain retries.

An operation is idempotent in the relevant effect semantics when repeated application does not create additional material effects beyond the intended logical operation.

For a pure function this is:

$$f(f(x)) = f(x)$$

For side-effecting requests the relevant property differs: repeated requests carrying the same idempotency key produce at most one material effect. Services often approximate this property with stable request identifiers so that multiple transport attempts resolve to one logical transaction (Featonby, 2021).

Outcome Assurance should preserve the relationship among:

- logical operation;
- individual attempts;
- idempotency identifier;
- observed external effects; and
- resulting task state.

Retries should therefore remain causally linked:

Logical Operation R1

```

├─ Attempt 1
├─ Attempt 2
└─ Attempt 3

```

rather than appearing as unrelated actions.

This allows the assurance system to reason about:

- duplicate execution;
- parameter drift between attempts;
- cumulative cost;
- changed authority;
- changed evidence;
- changed policy;
- eventual external effect; and
- retry limits.

Idempotency improves safe execution but does not establish successful outcome.

$$Idempotent(a) \not\Rightarrow CorrectOutcome(a)$$

An operation can reliably perform the wrong action exactly once.

Idempotency therefore supports Execution Assurance. It does not replace End-State Assurance.

8.8 Compensation and Remediation

Long-running tasks frequently cross system boundaries where atomic rollback is unavailable.

When a task partially succeeds, recovery may require compensating work (Garcia-Molina & Salem, 1987).

For example:

1. reserve inventory;
2. charge a payment method;
3. create a shipment;

4. fail before final confirmation.

Recovery may require:

- cancel shipment;
- issue refund;
- release inventory.

These operations do not erase the original history. They create a new state intended to compensate for it.

Thus:

Compensation $\neq$ Undo

A compensation should therefore be treated as a new consequential action or task with its own assurance requirements.

The same principle applies to remediation more generally.

If an end-state verifier determines that the task outcome is not satisfied, remediation may:

- repeat an action;
- reverse an action;
- compensate for an action;
- repair external state;
- gather additional evidence;
- obtain new approval;
- delegate corrective work; or
- escalate to a human.

Those corrective actions remain governed work. They do not bypass Outcome Assurance merely because they were triggered by an earlier failure.

8.9 Temporal Acceptance Predicates

Not all task requirements can be evaluated at a single instant.

Some are point-in-time predicates:

$$\textit{Status} = \textit{Approved}$$

Others are temporal.

A bounded eventuality may require:

$$\textit{Eventually}_{[0,30m]}(\textit{PaymentSettled})$$

An invariant might require:

$$\textit{Always}_{[t_{start}, t_{end}]}(\textit{Spend} \leq \textit{Budget})$$

A composite requirement might state:

$$\textit{Approval} \Rightarrow \textit{Eventually}_{[0,24h]}(\textit{RecordCreated})$$

These requirements have different assurance semantics (Bauer et al., 2011).

For:

$$\textit{Eventually}_{[0,30m]}(\textit{PaymentSettled})$$

the absence of settlement after ten minutes does not establish failure.

The appropriate verdict may be *Indeterminate*, with disposition *Wait* (cf. Section 7.10).

After thirty minutes, *NotSatisfied* requires admissible evidence that settlement did not occur within the interval—such as an authoritative query showing the payment unsettled at the deadline—unless another applicable rule extends or modifies the deadline. The mere absence of an observation leaves the claim *Indeterminate*.

8.10 Completion Validity and Outcome Persistence

A verified task outcome does not necessarily remain true forever.

Suppose a task provisions access for a contractor and End-State Assurance verifies at time t_0 that:

$$\textit{AccessProvisioned} = \textit{true}$$

The contractor is terminated five hours later.

That later event does not imply that the original task was incorrectly completed. It creates a new condition that may require new work:

$$\textit{ContractorTerminated} \rightarrow \textit{RevokeAccess}$$

Outcome Assurance should therefore distinguish:

completion validity — whether the task's acceptance conditions were satisfied when completion was verified;

from:

outcome persistence — whether some condition is required to continue holding after completion.

If persistence matters, it should be specified explicitly.

For example, with t_{term} the end of employment:

$$\textit{Always}_{[t_c, t_{term}]}(\textit{AccessWithinPolicy})$$

is a different requirement from:

$$\textit{AtCompletion}(\textit{AccessProvisioned})$$

Outcome Assurance therefore establishes claims according to their declared temporal semantics. A completion receipt should not be interpreted as an indefinite guarantee unless continued validity is itself part of the requirement.

8.11 Temporal Assurance State

The temporal state of a claim can be represented as:

$$TC_i = (C_i, V_i, \Delta_i, W_i, J_i)$$

where:

- C_i is the assurance claim;
- V_i is its current verdict;
- Δ_i is its dependency set (written $Deps(C_i)$ in Section 8.3);
- W_i is its validity window; and
- J_i is its invalidation or revalidation trigger set.

A claim is usable at time t only when:

$$\begin{aligned} & Usable(C_i, t) \\ &= (V_i = Satisfied) \wedge (t \in W_i) \wedge DependenciesValid(\Delta_i, t) \wedge \neg Triggered(J_i, t) \end{aligned}$$

This representation is intentionally operational rather than axiomatic. Its purpose is to expose the conditions under which an existing assurance result may be reused rather than recomputed.

For some claims, W_i may be effectively unbounded while dependency events remain the primary invalidation mechanism. For others, strict temporal expiration may dominate. For high-consequence claims, both may apply.

Applied to every claim in the task's assurance case, this predicate yields the Task Assurance State of Section 6.9, $AS_T(t) = Usable(AC_T, t)$. The central temporal distinction is:

$$Satisfied(C_i, t_0) \not\Rightarrow Usable(C_i, t_1)$$

and therefore:

$$AS_T(t_0) \neq AS_T(t_1)$$

may hold while the task identity T_{id} remains unchanged. A resumed task continues the same work against a possibly smaller, or differently composed, set of usable claims, and any completion verdict it later reaches is evaluated against AS_T at verification time (Section 10.14).

8.12 Dependency-Aware Revalidation

Dependency-directed invalidation is not introduced here. Truth maintenance systems record the justifications on which each belief depends and retract only the beliefs whose support is withdrawn (de Kleer, 1986; Doyle, 1979). Dynamic safety-case management identifies the argument elements directly and then indirectly impacted by a system change (Cârlan et al., 2024), and revocation work for long-running agents requires the dependents of a retired authorization root to be effect-closed, terminal, or independently rebound (Zhu & Wang, 2026). Outcome Assurance applies the same pattern to a narrower object: the assurance claims whose usability may affect task completion.

Re-evaluating every assurance claim after each wait, retry, or resume is potentially safe but inefficient.

A more selective mechanism uses the assurance dependency graph.

Let:

$$Changed(X_{t_0}, X_{t_1})$$

identify material contextual changes between suspension and resumption.

Then:

$$Revalidate = \{C_i: Deps(C_i) \cap Changed \neq \emptyset\}$$

Only claims whose supporting assumptions may have changed require reevaluation, subject to any task-specific mandatory revalidation rules.

This approach can substantially reduce evaluation cost in long-running systems while preserving the assurance boundary.

However, selective revalidation depends on correct dependency modeling.

Where the system cannot determine whether a contextual change affects a claim, conservative behavior is required:

$$\textit{DependencyUncertain} \Rightarrow \textit{BroaderRevalidation}$$

This fallback assumes the system can tell when its dependency model is uncertain. A dependency that was never recorded produces no uncertainty signal. Selective revalidation is therefore only as safe as dependency capture is complete; unrecorded dependencies remain a limitation of the mechanism rather than a case it handles.

8.13 Execution Replay and Assurance Replay

Durable workflow systems commonly support replay of recorded execution history to reconstruct computational state (Temporal Technologies, n.d.).

Outcome Assurance requires a distinct capability.

Execution replay asks:

What did the workflow previously do, and what execution state follows from that history?

Assurance replay asks:

What assurance claims permitted those transitions, what evidence and context supported them, and would those claims still be applicable under a specified context?

These functions are related but not interchangeable.

Execution replay may establish:

At event 428:

Task was waiting for supplier confirmation.

Assurance replay may additionally establish:

At event 428:

Purchase approval was satisfied under policy Pol3,
 authority Auth7,
 approval Ap19,
 transaction amount \$4,000,
 and supplier Sup8.

That historical reconstruction is useful for audit and incident analysis.

But it still does not establish:

Purchase approval remains usable now.

This is the distinction drawn in Section 8.1 between historical satisfaction and current usability:

$$HistoricalAssurance(C, t_0) \neq CurrentAssurance(C, t_1)$$

Assurance replay reconstructs the former. Re-entry assurance establishes the latter.

8.14 Relationship to Contextual Assurance and Verification

Sections 6 and 7 defined contextual assurance requirements and Verification Contracts.

Temporal assurance adds the missing lifecycle dimension.

Section 6 asks:

What claim must be established for this consequential transition?

Section 7 asks:

What evidence and evaluation method establish that claim?

Section 8 asks:

For how long, and under what changing conditions, may that established claim continue to be relied upon?

The resulting lifecycle is:

Requirement → Claim → Verification → Satisfied → TemporalValidity
→ Reuse or Reevaluation

For long-running agentic work, the final two steps are necessary; without them, the system implicitly and unjustifiably assumes that historical assurance state remains valid indefinitely.

Section 8 Proposition

A previously satisfied assurance claim may be reused by a long-running agentic task only while the temporal, contextual, and dependency conditions under which it was established remain applicable. Restoration of execution state alone is insufficient to justify continuation. The usable subset of the task's assurance record is the Task Assurance State against which continuation and completion are evaluated.

This proposition implies a practical implementation requirement: durable agentic systems should preserve not only execution history but also assurance claims, their evidence, dependencies, validity conditions, and invalidation triggers.

Research Hypothesis

H5: For long-running agentic tasks, selective revalidation of the assurance claims that support task completion achieves failure detection equivalent to or better than full revalidation at materially lower cost, without increasing false completion beyond a prespecified margin.

Dependency-directed invalidation and revalidation are established mechanisms (Section 8.12). H5 does not test whether they work in principle; it tests whether restricting revalidation to completion-supporting claims preserves detection and completion correctness at lower cost.

A future empirical test should compare:

1. resumption without assurance revalidation;
2. complete revalidation of every claim after every resumption; and
3. selective revalidation of completion-supporting claims whose dependencies changed.

Injected changes should include authority revocation, policy change, approval expiration, transaction mutation, knowledge update, external state change, model or tool update, and uncertain side effects, together with matched changes that affect no completion-supporting claim.

The framework should retain selective revalidation only where its failure detection is non-inferior to full revalidation within a prespecified margin, its false-completion rate exceeds that of full revalidation by no more than a prespecified margin, and it materially reduces verification cost (Section 13.6). Where dependency coverage is uncertain, full or broader revalidation should remain the safe fallback.

9. Assurance Through Delegation

Agentic systems frequently decompose a parent task into work performed by child agents. Delegation can improve specialization, parallelism, and scale, but it also introduces an assurance boundary. The child receives only part of the work, operates under a derived authority, and returns a result that the parent may use in a larger outcome.

Authority and assurance compose differently across that boundary:

Authority attenuates. Assurance obligations do not.

Neither half of this line is original to this paper; Sections 9.3 and 9.5 identify its precedents. The rule this section adds is:

Delegated assurance must reconcile before it can support parent completion.

Authority determines what the child is permitted to do. Assurance determines what must be established before the child's work may support the parent outcome. These concerns interact, but neither can substitute for the other. A child may remain within its authority and still produce evidence insufficient for the delegated claim. It may also produce a substantively correct result through an unauthorized process.

Thus:

$$AuthorityValid(T_{ch}) \not\Rightarrow OutcomeAssured(T_{ch})$$

and:

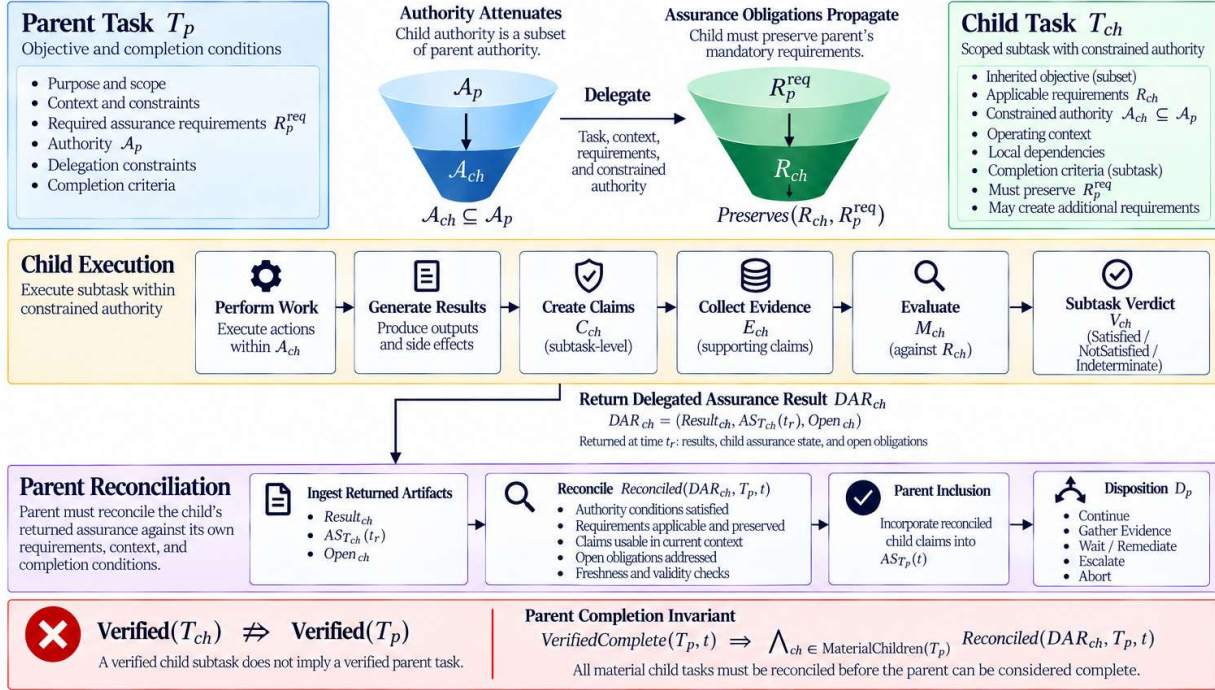
$$GoodResult(T_{ch}) \not\Rightarrow AuthorizedExecution(T_{ch})$$

Outcome Assurance therefore treats delegation as the transfer of bounded work, not as the automatic transfer or discharge of the parent's assurance obligations.

Figure 4 illustrates assurance through delegation.

Figure 4*Assurance Through Delegation*

When a task delegates work to a subtask, authority attenuates (the child has a subset of the parent's authority), the mandatory assurance obligations propagate (the child must preserve the parent's required assurance requirements). The child produces results, claims, and evidence, which the parent must reconcile against its own requirements. Verification of the child does not imply verification of the parent.

**9.1 Delegation Decomposes Work, Not Parent Assurance**

Suppose parent task T_p is to onboard a supplier and child task T_{ch} is to perform sanctions screening. The parent can delegate execution of the screening. It does not thereby eliminate the parent requirement that the supplier pass a valid sanctions screening before onboarding completes.

In general:

$$Delegate(T_{ch}) \not\Rightarrow Discharge(Assurance_{T_p})$$

unless the governing institutional model explicitly defines a transfer of accountability and the parent's acceptance conditions recognize it. Contemporary analysis of multi-agent accountability identifies authorization dilution, intent drift, provenance loss, and responsibility

laundering as failure modes introduced by delegation chains (Bahidika, 2026). These failures are not resolved merely because every child reports completion.

The parent remains responsible for determining what the child result establishes for the parent task.

9.2 A Delegation Needs Separate Authority and Assurance Contracts

A defensible delegation boundary carries two logically distinct contracts:

Authority contract — what the child is permitted to do; and

Assurance contract — what the child must establish and return before the delegated work can be accepted.

The delegation contract can therefore be represented as:

$$DC = (AC, SC)$$

where AC is the authority contract (distinct from the assurance case AC_T of Section 6.8) and SC is the assurance contract.

The authority contract may constrain:

- permitted and prohibited actions;
- data and tool scope;
- financial or resource limits;
- validity periods;
- redelegation rights; and
- maximum delegation depth.

The assurance contract may specify:

- the child objective;
- applicable acceptance predicates;

- mandatory assurance requirements;
- required evidence and sources;
- evaluation method and verifier assumptions;
- provenance and independence requirements;
- return structure;
- completion dependencies; and
- failure and cancellation semantics.

Separating these contracts makes two independent questions inspectable: whether the child was authorized to act and whether the delegated outcome was adequately assured.

9.3 Authority Must Monotonically Attenuate

Monotonic attenuation is established prior art in capability systems and is increasingly explicit in agent-specific authorization work. Macaroons provide a foundational example of delegable credentials whose caveats attenuate and contextually confine authority (Birgisson et al., 2014). User Controlled Authorization Network (UCAN) defines attenuation as constraining capabilities along a delegation chain and requires each direct delegation to restate or diminish the authority it received (UCAN Working Group, n.d.). Microsoft’s Agent Governance Toolkit similarly describes parent-to-child delegation as passing a subset of capabilities and enforcing narrowed scope across delegation hops (Microsoft, 2026).

Current agent-specific proposals follow the same invariant. The active Internet-Draft *An Attenuated Delegation Profile for Automated Agents* specifies a chain in which each link can only narrow scope and validity conditions, with verification of every link rather than only the endpoint (Hassan, 2026). The Agentic Principal Chain proposed by Muruaga carries and restricts delegated scope and budgets across a session (Muruaga, 2026). Agent-delegation frameworks

likewise require a delegatee's permission to be restricted to a strict subset of the delegator's authority and pass chains of attestations back to the delegator (Tomašev et al., 2026). These are current works rather than settled standards, but they reinforce the established security principle:

$$\mathcal{A}_{n+1} \subseteq \mathcal{A}_n$$

for each delegation hop. A child must not acquire authority that the parent did not possess or was not permitted to delegate.

9.4 Assurance Cannot Simply Be Inherited

A parent may provide the child with assurance claims established earlier in the task. Those claims are not automatically satisfied for the child context.

Suppose the parent has established:

$$C_p = \textit{DataSourceApproved}$$

A child may rely on that claim only when:

1. the claim applies to the delegated task;
2. its temporal and dependency conditions remain valid;
3. the child is permitted to rely on it;
4. the assurance contract does not require independent re-collection or re-verification; and
5. the original claim's scope includes the child's execution context.

Therefore:

$$\textit{Satisfied}(C_p) \not\Rightarrow \textit{Satisfied}(C_{ch})$$

without an explicit applicability relationship. Consistent with Section 8, a reusable parent claim is an assurance dependency supplied to the child. It may be consumed only when:

$$\textit{Applicable}(C_p, T_{ch}, X_{t,ch}) = \textit{true}$$

Delegation must therefore carry not only instructions but also the assurance context on which the child is allowed to rely.

9.5 Mandatory Assurance Obligations Must Not Be Weakened

If a parent requirement states that a sanctions screening must be current within 24 hours, a child cannot silently weaken the requirement to 30 days. Let $R_p^{req} =$

$RequiredSubset(R_p, T_{ch})$ be the mandatory parent requirements allocated to child task T_{ch} .

Then the child’s requirements must satisfy:

$$Preserves(R_{ch}, R_p^{req})$$

where `Preserves` means that the child requirements preserve or strengthen every mandatory assurance obligation applicable to the delegated work. A child may add requirements or use an equivalent formulation, but it must not weaken the obligations imposed by the parent task. This paper does not define a general strength ordering over requirements. `Preserves` can be checked mechanically only for restricted requirement forms—such as freshness windows, monetary bounds, or required source sets—where strengthening reduces to a parameter comparison. For other forms it is a reviewed and recorded judgment rather than an automated check.

This produces a paired monotonicity rule:

$$\mathcal{A}_{ch} \subseteq \mathcal{A}_p$$

while:

$$Preserves(R_{ch}, R_p^{req})$$

Non-weakening obligation propagation under delegation also has direct precedent. Delegation-aware runtime contracts for LLM multi-agent systems require workflow-contributing obligations to be inherited or transformed and never dropped, require every parent limit to

reappear at an equal or smaller value, and revoke delegated contracts transitively (Bamil, 2026). Contract-based design supplies the general theory of composing and refining assume-guarantee contracts (Benveniste et al., 2018; Clarke et al., 1989; Dewes & Dimitrova, 2025; Saoud et al., 2021; Šljivo et al., 2017). Outcome Assurance adopts this invariant.

9.6 Contract-Based Composition Provides the Formal Precedent

The separation of assumptions and guarantees is not new. Compositional model checking and assume-guarantee reasoning establish system properties by checking component properties under explicit environment assumptions (Clarke et al., 1989). Recent work applies quantitative assume-guarantee contracts to multi-agent systems with local task specifications and shared system constraints (Dewes & Dimitrova, 2025). Safety-case research similarly uses assumption-guarantee safety contracts, contextual assumptions, and supporting evidence to construct higher-level assurance arguments from component-level artifacts (Šljivo et al., 2017).

A delegated child task can therefore be approximated as:

$$Contract_c = (Assumptions_c, Guarantees_c)$$

The child establishes specified guarantees under stated assumptions. The parent must determine both that the assumptions held and that the guarantees are sufficient for the parent claim.

Outcome Assurance does not replace this theory. It applies the compositional pattern to dynamic agent tasks in combination with authority boundaries, runtime evidence, and temporal applicability, each of which has its own precedent (Sections 9.3, 2.6, and 8.12), and adds the explicit parent reconciliation rule of Section 9.8.

9.7 Child Completion Does Not Imply Parent Completion

A child may fully complete its delegated scope while the parent task remains incomplete:

$$Verified(T_{ch}) \not\Rightarrow Verified(T_p)$$

For supplier onboarding, verified sanctions screening says nothing by itself about tax validation, payment setup, contract approval, or the supplier's final status in the system of record. Even if all delegated subtasks are verified, the parent may still have integration predicates that no child owns.

Parent completion therefore requires:

$$VerifiedComplete(T_p, t) \Rightarrow ParentPredicates \wedge \bigwedge_{ch \in MaterialChildren(T_p)} Reconciled(DAR_{ch}, T_p, t)$$

rather than merely:

$$Verified(T_p) \Leftrightarrow \bigwedge Verified(T_{ch})$$

unless the parent acceptance predicates explicitly define completion that way. Section 9.8 defines *MaterialChildren*(T_p) and *Reconciled*(DAR_{ch}, T_p, t).

9.8 Parent Reconciliation Is Mandatory

The parent task requires a *Reconciliation Gate* after delegated work returns:

Child completes

↓

Child returns:

outcome

assurance claims

evidence and verification references

authority and delegation provenance

exceptions and unresolved requirements

↓

PARENT RECONCILIATION

↓

Accept | Revalidate | Remediate |

Redelegate | Escalate | Reject

At this gate, the parent determines:

1. whether the child performed the intended delegated task;
2. whether its authority was valid throughout execution;
3. whether inherited mandatory assurance obligations were preserved;
4. whether the returned evidence is admissible for the parent claim;
5. whether child outputs are internally consistent and compatible with other child results;
6. whether returned claims remain temporally usable;
7. whether the child guarantee supports a parent requirement; and
8. whether parent-level or independent verification remains necessary.

This gate is the delegation-specific rule of Outcome Assurance. Let

$MaterialChildren(T_p)$ be the delegated children whose results or claims support a mandatory parent predicate or requirement, and let DAR_{ch} be the Delegated Assurance Result returned by child T_{ch} (Section 9.12). Reconciliation against the parent at time t requires at least:

$$\begin{aligned}
 Reconciled(DAR_{ch}, T_p, t) \Rightarrow & \text{AuthorityValid}(T_{ch}) \wedge \text{Preserves}(R_{ch}, R_p^{req}) \\
 & \wedge \forall C \in Relied(DAR_{ch}): \text{Applicable}(C, T_p, X_t) \wedge \text{Usable}(C, t) \\
 & \wedge Open_{ch} \cap Mandatory(T_p) = \emptyset
 \end{aligned}$$

where $Relied(DAR_{ch})$ is the set of returned child claims on which the parent relies and $Mandatory(T_p)$ is the set of mandatory parent obligations. An open child obligation that is mandatory for the parent prevents reconciliation unless the parent itself discharges it, which removes it from $Open_{ch}$. Child assurance state enters the parent's Task Assurance State only through reconciliation:

$$C \in Relied(DAR_{ch}) \wedge C \in AS_{T_p}(t) \Rightarrow Reconciled(DAR_{ch}, T_p, t)$$

and parent completion requires:

$$VerifiedComplete(T_p, t) \Rightarrow \bigwedge_{ch \in MaterialChildren(T_p)} Reconciled(DAR_{ch}, T_p, t)$$

Reconciliation is evaluated at time t , not once at return. A child claim reconciled earlier ceases to support the parent if it later becomes unusable (Section 8). The listed conditions are necessary rather than sufficient; items 1–8 above and the parent’s Verification Contract may impose more.

9.9 Authority Provenance and Assurance Provenance Are Distinct

Delegation creates two provenance chains.

Authority provenance answers:

Who authorized this agent to act, through what chain, and with what scope?

Assurance provenance answers:

Which task produced this claim, using what evidence and evaluation, and how is the claim consumed by the parent assurance case?

Cryptographic delegation mechanisms can make authorization chains tamper-evident.

The Human Delegation Provenance protocol, for example, proposes signed, append-only records of delegation hops (Dalugoda, 2026). Such a chain can establish who authorized whom under a stated scope. It does not establish that the delegated work was performed correctly.

Conversely, a verified child result does not establish that the child was authorized to produce it. A delegation record should therefore preserve both lineages:

Principal $\rightarrow$ Authority Chain $\rightarrow$ Child Agent

Child Task $\rightarrow$ Assurance Claims $\rightarrow$ Evidence / Verification
 $\rightarrow$ Parent Assurance Case

When the parent uses a child conclusion, the lineage should remain:

$$ParentClaim \leftarrow ChildClaim \leftarrow ChildEvidence$$

The parent should not relabel child evidence as though the parent produced it, unless the parent independently collected or verified that evidence. This preserves the provenance distinctions introduced in Section 7 and W3C PROV (Moreau & Missier, 2013).

9.10 Redelegation Must Preserve Both Invariants

Redelegation compounds authority and assurance risks. For a chain:

$$A \rightarrow B \rightarrow C \rightarrow D$$

each hop must verify:

$$\mathcal{A}_{ch} \subseteq \mathcal{A}_p$$

and:

$$Preserves(R_{ch}, R_p^{req})$$

An authoritative-source requirement, 24-hour freshness condition, or independent-verification obligation must not disappear because an intermediate agent delegates again. The child may strengthen or operationally transform a requirement, but equivalence or strengthening must be demonstrable.

Preserving the requirement does not require every descendant to duplicate evidence collection. Existing evidence or a previously satisfied claim may be reused when its Verification Contract permits reuse and the applicability conditions from Sections 7 and 8 remain satisfied:

$$RequirementPropagation \neq EvidenceReplication$$

Portable Assurance Records (Section 12) can make such reuse efficient by carrying signed, context-bound references to assurance state. They are an implementation pattern, not a core mechanism of this section.

9.11 Delegation Should Carry an Assurance Envelope

A useful implementation abstraction is a *Delegation Assurance Envelope*:

DAE

$= (TaskContract, AuthorityEnvelope, AssuranceObligations, ReusableClaims, EvidenceObligations, \dots)$

It carries:

- the work being delegated;
- the child's attenuated authority;
- mandatory assurance obligations that must be preserved;
- upstream claims the child may reuse, with applicability conditions;
- evidence and verification obligations; and
- the outcome, assurance state, and failure information that must be returned.

The Delegation Assurance Envelope is a conceptual package, not a proposed interchange standard. Its purpose is to make the required boundary state explicit.

9.12 Delegated Work Should Return Structured Assurance State

Natural-language output alone is insufficient for deterministic parent reconciliation. A child return should expose structured assurance state, for example:

```
child_task_id
child_task_version

outcome
completion_verdict
```

```

assurance_claims[]
evidence_refs[]
verification_refs[]

authority_chain_ref
delegation_ref

unmet_requirements[]
indeterminate_claims[]
exceptions[]

child_tasks[]

```

The schema is implementation-dependent; the requirement is that delegated work return assurance state, not merely task output.

The return must give the parent enough information to verify authority, evaluate evidence, preserve provenance, identify unresolved obligations, and determine an explicit disposition.

Implementation-neutrally, the return is a *Delegated Assurance Result*:

$$DAR_{ch} = (Result_{ch}, AS_{T_{ch}}(t_r), Open_{ch})$$

where $Result_{ch}$ is the child's outcome and completion verdict; $AS_{T_{ch}}(t_r)$ is the child's Task Assurance State (Section 6.9) at return time t_r —the child claims that were usable when the result was returned, with their evidence and verification references, dependencies, validity conditions, and authority and delegation provenance; and $Open_{ch}$ records unmet requirements, unresolved obligations, `Indeterminate` claims, and exceptions. The schema above is one

serialization of this triple. The parent consumes it only through the reconciliation rule of Section 9.8.

9.13 Failure, Cancellation, and Orphaned Work Require Explicit Semantics

Child termination is not equivalent to successful completion. A delegated task may be:

$ChildState \in \{Verified, NotVerified, Indeterminate, Failed, Cancelled, Expired\}$

Parent policy determines the consequence. *Indeterminate* may cause the parent to wait or gather evidence. *Failed* may permit an alternative delegation. *NotVerified* may require remediation, escalation, or abort.

Cancellation must also propagate deliberately. If a parent terminates while a child continues, the system may create orphaned authority, orphaned work, or orphaned side effects. It must determine whether the child's authority remains valid, whether execution should be revoked, how outstanding assurance obligations are closed, and whether completed side effects require compensation.

A result returned after parent cancellation may be locally valid yet unusable by the terminated parent. An external action performed after cancellation may require a new governed compensation task. The delegation tree must therefore carry cancellation, revocation, expiry, and late-result semantics as well as success semantics.

9.14 Delegation Topology Should Be Acyclic by Default

Compositional reasoning becomes more demanding when components depend recursively on one another's guarantees. Assume-guarantee research distinguishes the conditions sufficient for directed acyclic interconnections from the stronger semantics required for general interconnections containing cycles (Saoud et al., 2021).

Most delegation should therefore form a tree or directed acyclic graph:

Parent → Children → Descendants

If agent *A* delegates to *B*, *B* to *C*, and *C* delegates back to *A*, assurance obligations and guarantees may become circular. Cyclic delegation should be rejected by default or supported only by an explicit composition model that can resolve those dependencies.

Delegation depth may also be bounded:

$$DelegationDepth \leq d_{max}$$

because deeper chains increase latency, context drift, provenance complexity, the difficulty of authority reasoning, and the risk that mandatory assurance obligations are lost or transformed incorrectly (Hassan, 2026; Muruaga, 2026).

9.15 Structured Constraints Are Stronger Than Inferred Intent Alone

Intent preservation across natural-language delegation is not reliably deterministic. SentinelAgent separates deterministic controls such as authority narrowing, policy preservation, reconstructibility, containment, action conformance, and output-schema conformance from probabilistic intent preservation; its reported intent control degrades substantially under sophisticated paraphrasing attacks (Patil, 2026).

This supports a conservative design rule: semantic intent evaluation may be useful as a secondary control, but it should not replace structured task scope, prohibited actions, acceptance predicates, authority constraints, and assurance requirements.

The parent should communicate what the child must do and establish through explicit contract fields wherever those fields can be defined.

9.16 The Parent Remains the Integration and Assurance Boundary

The central architectural conclusion is:

A child assures the delegated work. The parent assures what that work means to the parent outcome.

Formally, child assurance establishes a claim about T_{ch} :

$$ChildAssurance(T_{ch})$$

The parent then evaluates:

$$Supports(ChildAssurance(T_{ch}), ParentClaim)$$

using the parent's Verification Contract, contextual requirements, and current assurance state. The parent owns integration across child results, resolution of conflicts and missing dependencies, evaluation of parent-only predicates, and the final completion determination.

This yields the delegation lifecycle:

PARENT TASK

↓

Define delegated subtask

↓

Construct Delegation Assurance Envelope

┌ task contract

┌ attenuated authority

┌ non-weakened assurance obligations

┌ reusable claims and applicability conditions

┌ evidence obligations

└ return contract

↓

CHILD TASK

↓

Child Outcome Assurance

↓

Structured child result

↓

PARENT RECONCILIATION

├ authority verification

├ assurance verification

├ evidence admissibility

├ temporal validity

└ parent applicability

↓

Accept | Revalidate | Remediate |

Redelegate | Escalate | Reject

Section 9 Proposition

Delegation may transfer bounded authority and responsibility for performing work, but it does not automatically transfer or discharge assurance. Authority must monotonically attenuate across delegation boundaries, while mandatory assurance requirements must not be weakened. A parent task may rely on a child result only after reconciling the child's returned assurance state—authority, assurance claims, evidence, open obligations, and verified outcome—against the parent's own requirements and current context.

The proposition can be summarized in three doctrine lines. The first restates established principles (Sections 9.3 and 9.5); the second and third state the delegation rule this paper adds:

Authority attenuates. Assurance obligations do not.

Delegated assurance must reconcile before it can support parent completion.

A child assures the delegated work. The parent assures what that work means to the parent outcome.

Research Hypothesis

H6: Mandatory parent reconciliation of returned child assurance state detects delegation-induced assurance failures that authority attenuation and non-weakening obligation propagation alone do not detect.

A future empirical test should compare:

1. ordinary task delegation;
2. delegation contracts with authority attenuation and non-weakening obligation propagation, in which the parent accepts a child result on the child's own verification (the strong baseline, reflecting the established mechanisms of Sections 9.3 and 9.5); and
3. the same delegation contracts plus a returned Delegated Assurance Result and mandatory parent reconciliation (Sections 9.8 and 9.12).

Injected conditions should concentrate on failures that can survive condition 2: a child claim that was usable at return but becomes unusable before parent completion; a verified child result whose supporting claims do not apply in the parent context; an unresolved child obligation that is mandatory for the parent; an `Indeterminate` child claim relied on by a parent predicate; mutually inconsistent sibling results; and a result produced after authority expiry or parent cancellation. Controls should include authority amplification, a dropped assurance requirement, and unauthorized redelegation, which condition 2 should already detect.

Evaluation should measure incorrectly accepted child results, false parent completion, failure localization, incorrectly rejected valid delegated work, and reconciliation overhead. H6 is supported only if condition 3 reduces incorrectly accepted child results or false parent completion relative to condition 2 by a prespecified material amount, without an unacceptable increase in incorrectly rejected valid delegated work (Section 13.7). If condition 2 produces

equivalent parent decisions and failure localization, the reconciliation rule should be removed or narrowed. Thresholds are set per task class before testing, not after results are observed.

10. Completion Verification

Outcome Assurance ultimately requires a task-level answer: when is the whole job actually done? Termination, successful tool use, returned child tasks, and satisfied intermediate claims can coexist with an incomplete or unacceptable outcome. Completion is therefore a separate assurance claim about the task's acceptance conditions and observed end state.

Let $C_{complete}(T)$ denote the claim that task T has satisfied its completion conditions; the Completion Contract below writes the same claim as C_T . Neither

$$ExecutionState(T) = Finished, \quad AgentReports(T) = Done, \quad \forall i, AssuranceClaim_i = Satisfied$$

necessarily establishes $C_{complete}(T)$. The first says that execution stopped. The second is a self-report. The third says that the selected subordinate claims passed. None, by itself, proves that the task's intended outcome was achieved under acceptable conditions.

Thus:

$$Finished(T) \not\Rightarrow VerifiedComplete(T)$$

and:

$$AssuredTransitions(T) \not\Rightarrow AssuredOutcome(T)$$

This section develops the task-level verification required to bridge that gap. Independent, evidence-gated completion verification is itself established in recent agent systems (Chen, 2025; J. Huang et al., 2026; Liao et al., 2026; Nguyen & Tran, 2026; Zhang et al., 2026). This section's additional requirement is stated in Section 10.14.

10.1 Completion Must Be Specified Before It Can Be Verified

Established acceptance practice begins with criteria defined before the acceptance decision. NASA guidance defines acceptance criteria as the conditions a system or component must satisfy to be accepted, requires the development team and customer to define and document those criteria during planning, and evaluates obtained results against expected results or agreed tolerances (National Aeronautics and Space Administration, n.d.). NASA's systems-engineering guidance likewise distinguishes product verification against specified requirements from product validation against stakeholder expectations and intended use (National Aeronautics and Space Administration, 2016).

For an agentic task, these ideas imply a *Completion Contract* established before execution where practical and revised only through a governed change process. The contract answers:

What must be observed, and under what conditions, before this task may legitimately be called complete?

The Completion Contract is not a new assurance theory. It is a task-level specialization of the Verification Contract introduced in Section 7:

$$CC_T = (C_T, P_T, E_T, M_T, B_T, \theta_T, \tau_T, Q_T)$$

where:

- C_T is the task-level completion claim;
- P_T is the set of acceptance predicates;
- E_T identifies admissible evidence and authoritative observations;
- M_T is the evaluation method;
- B_T specifies required verifier-independence and trust-boundary conditions;

- θ_T defines tolerances and acceptable outcome equivalence;
- τ_T defines temporal semantics; and
- Q_T identifies continuing obligations.

Evaluating the contract produces the completion verdict V_T and task disposition D_T , which are recorded with it but are not part of its specification.

The contract is evaluated over the task's current assurance state, the authoritative observed state, and the applicable context:

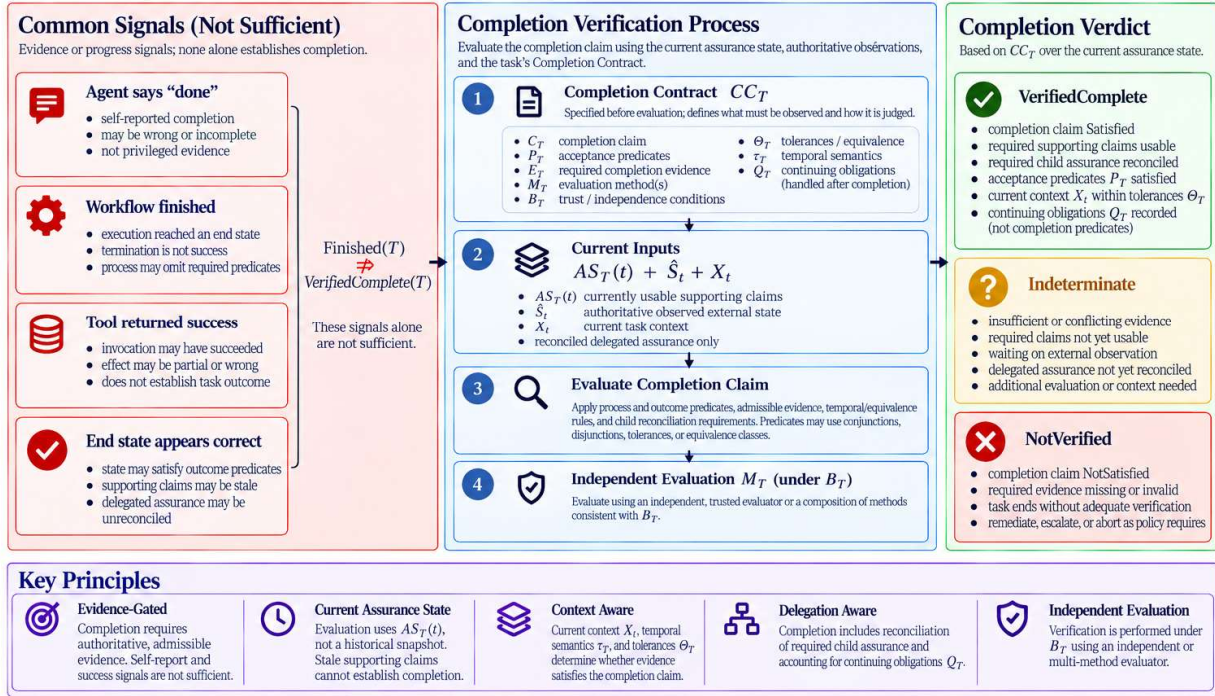
$$CC_T(AS_T(t), \hat{S}_t, X_t) \rightarrow V_T$$

where $AS_T(t)$ is the Task Assurance State (Section 6.9), $\hat{S}_t$ is the state reconstructed from authoritative observations (Section 5.5), and X_t is the time-indexed context. The expression is shorthand for an evaluation that applies every field of CC_T ; it does not replace those fields.

Figure 5 summarizes completion verification.

Figure 5*Completion Verification*

A task is not complete simply because an agent reports it is done, a workflow has finished, a tool returned success, or the final state appears correct. Completion is established only when the Completion Contract evaluates authoritative observations together with the task's current usable assurance state under the applicable context and trust conditions.



Specifying this contract in advance prevents the executing agent from redefining success after observing what happened. Where requirements legitimately evolve, the contract must preserve versioning, authority, provenance, and the reason for the change.

10.2 Acceptance Predicates Define the Outcome

Let:

$$P_T = \{p_1, p_2, \dots, p_n\}$$

be the predicates that must be evaluated for task T . For a supplier-onboarding task, they might include:

Supplier record exists

Supplier status = ACTIVE

Tax validation = CURRENT

Payment details = VERIFIED

Contract acknowledgment = RECEIVED

If every predicate is mandatory, the basic rule is:

$$VerifiedComplete(T) \Leftrightarrow \bigwedge_{i=1}^n p_i(S)$$

where S is the relevant observed state. A Completion Contract may instead allow alternatives:

$$p_1 \wedge (p_2 \vee p_3)$$

tolerances:

$$|Observed - Target| \leq \epsilon$$

acceptable equivalence classes defined by θ_T :

$$S_{observed} \in \mathcal{E}_T$$

or bounded temporal conditions:

$$Eventually_{[0,30m]}(PaymentSettled)$$

The required outcome should be expressed independently of one expected execution path.

A task to book an approved hotel in a designated area below a price ceiling may admit several materially equivalent reservations. Completion should recognize every state inside the declared acceptable class rather than require one arbitrarily selected hotel.

10.3 Process Predicates and Outcome Predicates Remain Distinct

Some completion requirements concern how the task was performed:

required approval obtained

no prohibited tool used

authority valid at commitment

mandatory child task verified

Others concern what state resulted:

payment settled

record active

refund posted

customer balance corrected

The Completion Contract should distinguish $P_{process}$ from $P_{outcome}$ and may require both:

$$VerifiedComplete(T) \Leftrightarrow Satisfied(P_{process}) \wedge Satisfied(P_{outcome})$$

together with the contract’s evidence, temporal, independence, and attribution conditions.

This separation prevents two opposite errors: accepting a correct state produced through a prohibited process, and accepting a conforming process that failed to produce the required state. NASA’s distinction between verification and validation provides an established analogue (National Aeronautics and Space Administration, 2016). End-State Assurance evaluates authoritative external-state predicates; Completion Verification evaluates the whole task-level completion claim, including all applicable process and outcome obligations.

Current agent evaluation reflects the same distinction. STATE-Bench requires an agent to follow applicable procedures and, for state-mutating tasks, updates a task-local environment whose final state is checked by deterministic assertions (Liu & Yadav, 2026). NVIDIA’s agent-evaluation guidance separately describes step-level process scoring and end-to-end outcome scoring over the final environment state (Abbassi et al., 2026). These benchmarks do not establish a general runtime assurance model, but they demonstrate why process evidence and final-state evidence answer different questions.

10.4 Authoritative State Is Preferred Over Agent Self-Report

Where an intended outcome is externally observable, completion should be established from authoritative state rather than from the executing agent’s narrative about that state. The statement:

“The supplier was successfully onboarded.”

is weaker than observations such as:

```
VendorMaster.Supplier123.status = ACTIVE
TaxSystem.Supplier123.validation = CURRENT
PaymentSystem.Supplier123.account = VERIFIED
ContractSystem.Supplier123.signature = RECEIVED
```

STATE-Bench’s deterministic state assertions and NVIDIA’s final-environment scoring both illustrate this preference for inspecting the resulting world rather than accepting a plausible final answer (Abbassi et al., 2026; Liu & Yadav, 2026). The doctrine is:

Agent output may describe completion. Authoritative state establishes completion.

This is a preference, not an assumption that every task exposes a queryable system of record. When authoritative state is unavailable, the Completion Contract must identify what alternative evidence is admissible, how it will be evaluated, and which trust assumptions remain. Self-report may be part of that evidence, but it is not privileged merely because it was produced by the executing agent.

10.5 Distributed Outcomes Require Partial-Completion Handling

Many consequential tasks produce a state vector rather than a single final variable. Supplier onboarding may modify enterprise resource planning, procurement, tax, banking, identity, and document systems:

$$S_T = (s_1, s_2, \dots, s_n)$$

The completion rule then evaluates:

$$P_T(S_T) = \text{true}$$

across the required observations. If the ERP record is active, payment setup is verified, and tax status is current, but a mandatory contract signature is missing, the task is partially fulfilled but not complete.

The verifier therefore needs explicit partial-completion semantics. It should identify satisfied, unsatisfied, and unobserved predicates; preserve the side effects already produced; and select a disposition such as waiting, gathering evidence, remediation, compensation, escalation, or failure. It must not collapse a mostly correct distributed state into success simply because a majority of operations completed.

10.6 Child Completion Is Evidence for Parent Completion

Section 9 established:

$$\text{Verified}(T_{ch}) \not\Rightarrow \text{Verified}(T_p)$$

A verified child claim may satisfy or support one parent predicate:

$$p_j(T_p) \leftarrow \text{VerifiedClaim}(T_{ch})$$

but the parent still owns $C_{complete}(T_p)$ and evaluates its own Completion Contract. As in Section 9.7, the parent may retain predicates that no child owns.

Child completion is evidence for, not a substitute for, parent completion.

The parent therefore reconciles required child receipts, verifies that their claims remain applicable, and evaluates the composed outcome against parent-level acceptance conditions.

10.7 Causal Attribution Is Contextual

An observed target state does not always establish that the governed task produced it. Suppose a task is authorized to reduce an account balance from \$1,000 to \$900. The agent issues

a \$100 credit while an unrelated reconciliation process independently applies the same adjustment. At verification, the target balance is present, but the end state alone does not identify which process caused it.

For tasks where provenance of the change matters, the Completion Contract may require:

$$DesiredState \wedge CausallyAttributedTo(T)$$

Relevant evidence may include transaction identifiers, correlation IDs, ledger entries, idempotency keys, event lineage, provenance records, or other system-specific causal links. Database-causality research formalizes why an observed answer and responsibility for that answer are distinct questions (Meliou et al., 2010).

Causal attribution is not universal. For some tasks, the final state alone is sufficient. For financial commitments, regulated actions, controlled changes, or disputes over responsibility, the organization may need evidence that the authorized task produced the outcome. The Completion Contract determines whether attribution is required and what evidence can establish it.

10.8 Termination Conditions Differ From Success Conditions

A task may terminate because it completed, failed, became impossible, lost authority, was cancelled, reached a deadline, or was superseded. Therefore:

$$Terminated(T) \not\Rightarrow VerifiedComplete(T)$$

Outcome Assurance should preserve distinct terminal states:

$$TerminalState(T)$$

$$\in \{VerifiedComplete, NotVerified, Failed, Cancelled, Expired, Superseded\}$$

`VerifiedComplete` means that the completion claim was satisfied under the Completion Contract. `NotVerified` means the task ended without an adequate verified completion claim.

Failed, Cancelled, Expired, and Superseded record different reasons execution or responsibility ended and support different remediation, retry, compensation, and audit semantics.

Indeterminate normally remains a verification verdict rather than a terminal task state.

Policy may direct the task to wait, reobserve, gather evidence, escalate, or eventually terminate as NotVerified.

The paper uses several result vocabularies. They map as shown in Table 4:

Table 4

Mapping of Result Vocabularies

Object	Values	Relationship
Claim verdict V_i (Section 7.7)	Satisfied, NotSatisfied, Indeterminate	Epistemic result for one claim.
Completion verdict V_T	Satisfied, NotSatisfied, Indeterminate	Claim verdict for C_T ; Satisfied licenses VerifiedComplete.
Verification labels (Sections 4.4, 5.5)	Verified, NotVerified, Indeterminate	The same three values, named for end-state and completion verification.
Terminal task state	VerifiedComplete, NotVerified, Failed, Cancelled, Expired, Superseded	Task lifecycle outcome; Indeterminate is not terminal.
Child task state (Section 9.13)	Verified, NotVerified, Indeterminate, Failed, Cancelled, Expired	Child outcome as seen by the parent; Indeterminate means the child's completion verdict is unresolved.
Action state (Section 8.6)	NotIssued, Succeeded, Failed, OutcomeUnknown	Execution status of one attempt, not an assurance verdict.

10.9 Missing Predicates Must Not Be Inferred

Suppose four predicates are required and the verifier observes:

$$p_1 = \text{true}, \quad p_2 = \text{true}, \quad p_3 = \text{true}, \quad p_4 = \text{unknown}$$

The verifier cannot soundly conclude `VerifiedComplete=true` unless the Completion Contract makes p_4 optional or provides another sufficient rule. The appropriate completion verdict is `Indeterminate`, or `NotSatisfied` when admissible evidence establishes that the predicate does not hold. The operational disposition then follows the contract.

This treatment is consistent with finite-trace runtime verification, which distinguishes definitive satisfaction or violation from an inconclusive verdict when a finite observation cannot yet determine the property (Bauer et al., 2011). Operational urgency must not turn missing evidence into presumed success.

10.10 Not Every Desirable Property Is a Completion Predicate

Finite observation places a boundary on what can be established at completion time. A bounded requirement such as:

$$\textit{Eventually}_{[0,30m]}(\textit{PaymentSettled})$$

can be evaluated after the settlement occurs or the interval expires. An unbounded property such as:

$$\textit{Always}_{[t_c,\infty)}(\neg \textit{SecurityViolation})$$

cannot generally be proven for all future time from a finite execution prefix. Runtime-verification semantics make this limit explicit: some properties have finite prefixes that determine satisfaction or violation, while others remain inconclusive under the available trace (Bauer et al., 2011).

The Completion Contract must therefore choose among three treatments:

1. restrict the property to a finite assurance interval;
2. transfer it into ongoing monitoring; or
3. acknowledge that final satisfaction cannot be established at task completion.

Completion predicates are conditions that must be established before `VerifiedComplete` may be issued. *Continuing obligations* survive task completion and are assigned to monitoring, another task, or another responsible principal. For example, correctly provisioning an employee's

access may be a finite completion predicate, while keeping that access within policy for the duration of employment is an ongoing obligation.

This distinction extends Section 8’s separation between completion validity and outcome persistence.

10.11 Completion Is Compositional but Not Mechanically Inferred

The task-level assurance case $AC_T(t)$ (Section 6.8) may contain claims, evidence, child receipts, execution history, and end-state observations. These elements support the completion claim, but their accumulation is not itself the completion rule. Instead, the completion verdict is evaluated over the usable subset of the case:

$$V_T = Evaluate(C_{complete}, AS_T(t), \hat{S}_t, CC_T)$$

Completion has its own claim, admissible evidence, evaluation logic, temporal conditions, and required independence. Satisfied action, execution, decision, and end-state claims are therefore inputs to a task-level evaluation whose acceptance semantics the Completion Contract defines.

The degree of verification independence is likewise contextual. An immutable authoritative query outside the executing agent’s control may be adequate for one completion claim, while another may require an independent model, human, organization, or validation system. Section 7’s distinctions among evidence, evaluation, and verdict independence apply without recreating a separate theory of independent completion verification.

10.12 Completion Receipts Attest to the Verdict

After the completion verifier issues a `Satisfied` verdict, the system may create a completion receipt binding information such as:

`task_id`

`task_version`

completion_claim

completion_verdict

acceptance_predicates

predicate_results

authoritative_observations

required_child_receipts

process_assurance_refs

exception_or_waiver_refs

verification_method

verifier_identity

trust_profile

evidence_root

issued_at

validity_semantics

continuing_obligations

The ordering is essential:

Verify $\rightarrow$ *IssueReceipt*

not:

IssueReceipt $\rightarrow$ *AssumeVerified*

A completion receipt attests that an identified verifier issued a verdict for a specified claim under stated conditions. It preserves provenance and supports later audit or reuse. It does not create the truth of the claim, cure inadequate evidence, or replace the Completion Contract.

10.13 Verified Completion Requires a Guarded State Transition

The runtime should treat the end of execution as a candidate for verification, not as authorization to mark the task complete:

Agent execution ends

↓

Candidate completion

↓

Completion verification

↓

[	
Satisfied	→ VerifiedComplete
NotSatisfied	→ Remediate / Fail / NotVerified
Indeterminate	→ Wait / Gather Evidence / Escalate
]	

The governing invariant is:

$$TaskState \rightarrow VerifiedComplete$$

only if:

$$V_T = Satisfied \quad \text{for} \quad C_{complete}(T)$$

under the current Completion Contract. The executing agent may submit a completion assertion and candidate evidence, but it must not unilaterally cause the verified terminal transition unless the contract explicitly makes that actor's assertion sufficient.

10.14 Completion Is a Verdict Over Current Assurance State

The Completion Contract's inputs are not only the final observed state. A completion verdict at time t is supported by claims in $AS_T(t)$: the claims on which the contract's process predicates, attribution conditions, and child-dependent predicates rely, including decision, action, execution, and delegated-assurance claims, and which remain usable at t .

Let $Supporting(CC_T)$ be the claims in $AC_T(t)$ on which the contract's process predicates, attribution conditions, and child-dependent predicates rely. Then:

$$\begin{aligned} VerifiedComplete(T, t) &\Rightarrow Supporting(CC_T) \\ &\subseteq AS_T(t) \wedge \bigwedge_{ch \in MaterialChildren(T)} Reconciled(DAR_{ch}, T, t) \end{aligned}$$

Consequently:

$$EndStateSatisfied(T, t) \wedge ExecutionFinished(T) \not\Rightarrow VerifiedComplete(T, t)$$

whenever a required supporting assurance claim is no longer usable or required delegated assurance has not been reconciled. Two cases illustrate the distinction. First, a sanctions-screening claim that must be current within 24 hours (Section 9.5) supports a supplier-onboarding predicate. Before verification the freshness window lapses, or the sanctions-list version on which the claim depends changes. The supplier record is `ACTIVE` and execution has finished, but the supporting claim is no longer in $AS_T(t)$; completion cannot be verified until the claim is revalidated or the task remediates. Second, a child returns a verified result whose supporting claims depend on a supplier status that changes after return. The child's own verdict stands as a historical record, but reconciliation against the parent fails at t , and the parent predicate that relies on the child is unsupported.

Evidence-gated completion systems already require sufficient, fresh evidence for completion (Chen, 2025; J. Huang et al., 2026; Liao et al., 2026; Nguyen & Tran, 2026; Zhang et

al., 2026). Outcome Assurance additionally requires that the assurance state supporting completion be usable at verification time, maintained through selective revalidation (Section 8) and delegated reconciliation (Section 9). Whether this adds measurable value beyond strong evidence-gated completion is the subject of H7.

Section 10 Proposition

Completion is a separate task-level assurance claim. A task may transition to `VerifiedComplete` only when its Completion Contract establishes the required process and outcome predicates from admissible evidence under the applicable temporal, independence, equivalence, and attribution conditions. Execution termination, satisfied intermediate claims, verified child tasks, and agent self-report are individually insufficient unless the Completion Contract explicitly defines them as sufficient. Evidence that the end state exists is likewise insufficient when a supporting assurance claim is no longer usable or required delegated assurance has not been reconciled: completion is evaluated against the task's current assurance state.

Research Hypothesis

H7: Completion evaluation over the current Task Assurance State reduces false completion relative to strong evidence-gated completion (B_E , Section 13) in tasks that contain temporal invalidation of supporting claims or delegated assurance dependencies.

A future empirical test should compare:

1. agent self-report;
2. workflow or step completion;
3. state-only completion;

4. evidence-gated completion B_E ; and
5. Completion Contract evaluation over $AS_T(t)$.

Section 13.8 adds successful tool calls and the equal-specification checker B_D to these conditions.

Injected conditions should include failures that any strong completion gate should catch—all planned steps executing while the final state is wrong, a correct final state produced through a prohibited process, a missing predicate in a distributed outcome, unavailable evidence, and a false agent success report—and failures that target the difference between conditions 4 and 5: a supporting process claim that expires or is invalidated by a dependency change after it was satisfied but before verification; a child claim that becomes unusable after return; an unresolved mandatory child obligation; a verified child result that does not apply in the parent context; and an `Indeterminate` supporting claim. Controls should include an eventual state not yet observable, alternate outcomes inside the declared equivalence class, and a continuing obligation incorrectly treated as a completion blocker.

Evaluation should measure false-positive completion, false-negative completion, time to verdict, and handling of unresolved or indeterminate evidence. H7 is supported only if condition 5 reduces false completion relative to B_E by a prespecified material amount on the targeted conditions, while correctly accepting alternate outcomes inside the declared equivalence class and keeping false rejection and indeterminacy within prespecified budgets (Section 13.8). If B_E performs equivalently, completion over current assurance state adds no measurable value for that task class.

11. Outcome Reliability

Outcome Assurance determines whether a particular task reached a verified acceptable outcome. Outcome Reliability addresses a different question:

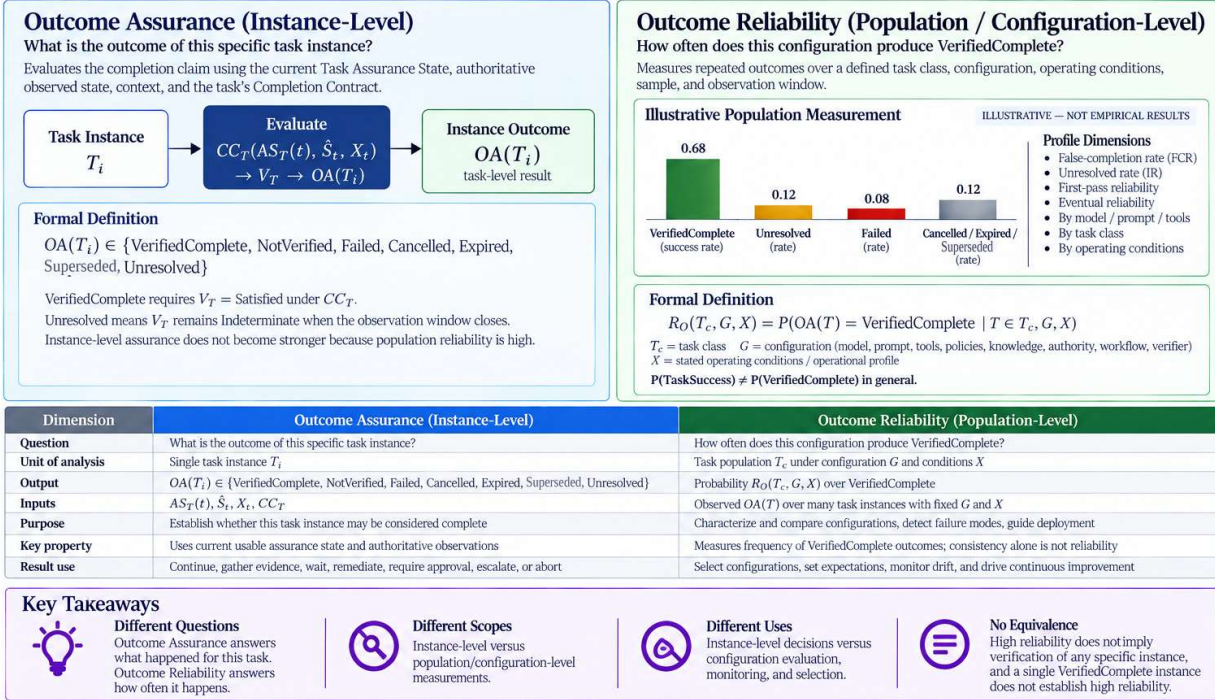
How reliably does a specified agentic configuration produce assured outcomes across a defined population of tasks under stated operating conditions?

The distinction is one of level. Sections 4–10 establish task-level assurance claims and a guarded transition to `VerifiedComplete`. Reliability is a population property. It characterizes repeated behavior of a defined configuration over a defined task class; it does not create a stronger verdict for any individual task.

Figure 6 contrasts instance-level Outcome Assurance with population-level Outcome Reliability.

Figure 6*Outcome Assurance vs. Outcome Reliability*

Outcome Assurance determines whether a specific task instance has a verified acceptable outcome under its Completion Contract. Outcome Reliability measures how often a specified agentic configuration produces VerifiedComplete outcomes across a defined task class and operating conditions. Consistency alone is not reliability.



For task instance T_i observed to the end of a declared observation window, Outcome Assurance produces an instance-level result:

$$OA(T_i)$$

$\in \{\text{VerifiedComplete}, \text{NotVerified}, \text{Failed}, \text{Cancelled}, \text{Expired}, \text{Superseded}, \text{Unresolved}\}$

The first six values are the terminal task states of Section 10.8. *Unresolved* records a task whose completion verdict is still *Indeterminate* when the observation window closes; it is a measurement outcome, not a terminal task state. Claim-level assurance verdicts and task-level outcome states are distinct namespaces (Section 10.8); the former evaluates individual assurance claims, while the latter characterizes the terminal status of the task as a whole.

For task class T_c , agentic configuration G , and operating conditions X , a basic Outcome Reliability quantity is:

$$R_O(T_c, G, X) = P(OA(T) = \textit{VerifiedComplete} \mid T \in T_c, G, X)$$

This adapts the classical reliability-engineering concern with performance of required functions under stated conditions (IEEE, 2017). The event being measured, however, is not agent self-report, workflow termination, or benchmark success in the abstract. It is a `VerifiedComplete` verdict established under the task’s Completion Contract.

In general, therefore:

$$P(\textit{TaskSuccess}) \neq P(\textit{VerifiedComplete})$$

Outcome Assurance supplies the measurement primitive; Outcome Reliability estimates how that primitive behaves across a population.

This paper does not introduce multidimensional reliability profiles for agents. Recent work separates outcome from trajectory consistency and reports robustness, predictability, and safety dimensions (Rabanser et al., 2026); `ReliabilityBench` combines repeated execution, semantically equivalent perturbation, and fault injection into a reliability surface (Gupta, 2026); trace-based assurance work reports termination reliability, contract-violation rates, and related metrics across stochastic seeds (Paduraru et al., 2026); and false-success studies measure how often agents assert completion that the environment contradicts (Advani, 2026). The distinction drawn here is narrower: Outcome Reliability conditions on the framework’s verified-completion event, so that false completion, `Unresolved` outcomes, and first-pass versus eventual completion are measured against `VerifiedComplete` rather than against benchmark task success.

11.1 Reliability Requires a Defined Claim Boundary

A reliability estimate is interpretable only when its claim boundary is explicit. At minimum, the report must identify:

- the task class and its material-equivalence criteria (Section 11.4);

- the tested agentic configuration;
- the operating conditions or operational profile;
- the sample size and sampling design;
- the observation window, including any permitted remediation; and
- the statistical uncertainty around the estimate.

An agentic configuration is more than a model:

$$G = (Model, Prompt, Tools, Policies, Knowledge, Authority, Workflow, Verifier)$$

A material change to any element can alter the relevant population and invalidate transfer of a prior estimate. A result for one configuration must not silently be attributed to another.

Likewise, there is no context-free Outcome Reliability score. Software-reliability engineering uses an *operational profile* to characterize operations and their probabilities of occurrence so that testing reflects use rather than an arbitrary task mixture (Musa, 1993). Agentic reliability claims require the same discipline, extended to conditions such as task difficulty, consequence level, tool environment, evidence availability, delegation depth, execution duration, and fault conditions.

A high aggregate rate over mostly easy or reversible work can conceal unacceptable performance on rare consequential tasks. The appropriate quantity is therefore conditional on X , the operating conditions or operational profile. In Section 11, bare X denotes the operating-condition boundary of the population-level reliability claim, while X_t denotes time-indexed context for an individual task. If different task strata carry materially different consequences or failure behavior, they should be reported separately rather than averaged into a universal scalar.

11.2 Consistency Is Not Reliability

Princeton’s agent-reliability methodology separates outcome, trajectory-distribution, trajectory-sequence, and resource consistency from robustness, predictability, and safety (Rabanser et al., 2026). An outcome-consistency measure can be perfect when repeated executions always succeed, but it can also be perfect when they always fail. Thus:

$$\text{Consistency} = 1 \not\Rightarrow R_O = 1$$

Consistency answers a different question. Reliability requires sufficiently frequent convergence on acceptable assured outcomes, not merely agreement among repeated results.

Consistency is not reliability: a system can be perfectly consistent because it fails every time.

The converse distinction also matters. Two successful executions need not follow identical paths. If materially equivalent tasks $T_i \sim T_j$ produce final states $S_i, S_j \in \mathcal{E}_T$, permissible trajectory variation is not itself a reliability failure. Outcome Reliability evaluates convergence on acceptable verified outcomes, subject to applicable process requirements, rather than lexical, token, or path convergence. Trajectory variation remains diagnostically useful for detecting unstable planning, cost variance, or fragile dependencies, but path identity is required only when the task contract makes it material.

11.3 Repeated Execution and pass^k

Single-run task success does not establish repeated-run reliability. τ -bench introduced pass^k to measure whether all k independent trials succeed and reported substantial degradation as k increased (Yao et al., 2025). Under a simplified independent-trial model with single-run success probability p :

$$\text{pass}^k = p^k$$

This differs from $\text{pass}@k$:

$$\text{pass}@k = 1 - (1 - p)^k$$

$\text{pass}@k$ asks whether at least one of k attempts succeeds. It is useful when the real system permits multiple safe attempts and can identify a successful result. pass^k asks whether every attempt succeeds and is therefore closer to repeated operational dependability. τ -bench estimates it from n trials per task with a combinatorial estimator rather than as $\hat{p}^k$ (Yao et al., 2025).

pass^k is useful, but it is not the definition of Outcome Reliability. Its value depends on the operational meaning of k , the independence or correlation of trials, and the benchmark’s binary success criterion. Shared infrastructure, cached context, upstream data, policies, tools, evaluators, and model behavior can create correlated failures. Moreover, Outcome Assurance preserves outcomes such as `Indeterminate` and false completion that should not be carelessly collapsed into one pass/fail value.

11.4 Repeatability, Reproducibility, and Stress Conditions

NIST measurement terminology distinguishes *repeatability*, agreement under the same conditions, from *reproducibility*, agreement under changed conditions whose changes are identified (Taylor & Kuyatt, 1994). Applied to agentic systems:

- *Outcome repeatability* evaluates materially equivalent tasks under materially the same configuration and operating conditions.
- *Outcome reproducibility* evaluates materially equivalent tasks after controlled changes such as model, prompt phrasing, tool implementation, infrastructure, time, or verifier.

The comparison unit is material task equivalence, not an identical prompt or identical execution trace. Tasks are materially equivalent when their objective, relevant entities, governing

rules, constraints, acceptance predicates, and consequentiality remain equivalent within declared tolerances.

Robustness should be represented as a test condition within the reliability profile rather than confused with reliability itself. ReliabilityBench combines repeated execution, semantically equivalent perturbations, and controlled tool or API faults in a reliability surface $R(k, \epsilon, \lambda)$, and uses end-state equivalence rather than text similarity for its metamorphic comparisons (Gupta, 2026). The design principle applies here: reliability should be tested across production-relevant variation, not only nominal inputs.

11.5 Reliability Estimates Require Uncertainty

An observed verified-outcome rate is an estimate, not the true production probability. If n_v of N sampled tasks are `VerifiedComplete`, then:

$$\hat{R}_O = \frac{n_v}{N}$$

should be accompanied by an appropriate confidence or credible interval. NIST guidance emphasizes quantitative statements of uncertainty and warns against treating qualitative labels such as precision or reproducibility as if they were quantitative estimates (Taylor & Kuyatt, 1994). For binomial proportions, NIST also notes that small samples or rare failures may require methods that do not rely on a simple symmetric normal approximation (National Institute of Standards and Technology & SEMATECH, n.d.).

Outcome Assurance does not mandate one interval estimator. The statistical method should fit the sampling design and claim. The minimum rule is:

A reliability claim without its population, configuration, operating conditions, sample size, and uncertainty is incomplete.

11.6 Report a Reliability Profile

A universal scalar can allow strong performance in low-consequence dimensions to conceal unacceptable behavior in a mandatory one. Outcome Reliability should therefore be reported as a task-class- and configuration-specific profile.

A useful profile may include:

$$RP(T_c, G, X) = \{R_v, R_{first}, R_{eventual}, R_k, R_{perturb}, R_{fault}, FCR, IR, TTAO, U\}$$

where, conceptually:

- R_v is the `VerifiedComplete` outcome rate;
- R_{first} is first-pass reliability without remediation;
- $R_{eventual}$ is eventual reliability within permitted remediation;
- R_k is a repeated-run statistic such as `pass^k`;
- $R_{perturb}$ is reliability under materially equivalent perturbations;
- R_{fault} is reliability under defined operational faults;
- FCR is the false-completion rate;
- IR is the indeterminate rate;
- $TTAO$ is the time-to-assured-outcome distribution; and
- U represents statistical uncertainty.

The tuple is extensible and not every field is mandatory for every domain.

False completion deserves separate treatment because it is qualitatively different from visible failure:

$$FalseCompletion = AgentReportsSuccess \wedge CompletionContract \neq Satisfied$$

An ordinary failure announces that the task did not complete. A false completion asserts success without the verified outcome. Its rate should remain visible and should not disappear inside average accuracy.

The `Indeterminate` rate must also be preserved. A high indeterminate rate can reveal poor observability, weak Verification Contracts, unavailable authoritative systems, evidence latency, or evaluator-scope problems. A system that declines to overclaim can have a low false-completion rate and a high indeterminate rate; whether that tradeoff is acceptable depends on the task class and cannot be resolved by one aggregate score.

11.7 First-Pass, Eventual, and Timely Reliability

Permitted remediation changes the interpretation of success. Suppose initial verification fails, the system performs governed remediation, and a later verification succeeds. The task may eventually reach `VerifiedComplete`, but the reliability report should preserve the distinction between:

$$R_{first} = P(\textit{VerifiedComplete without remediation})$$

and:

$$R_{eventual} = P(\textit{VerifiedComplete within permitted remediation})$$

The number of remediation cycles, their cost, and their failure modes may be operationally material. Without this distinction, a system that usually completes correctly on its first attempt can appear equivalent to one that routinely requires recovery.

Time can also be part of the assured outcome. When the Completion Contract contains a deadline or service level, the relevant quantity is:

$$P(\textit{VerifiedComplete} \wedge \textit{TimeToAssuredOutcome} \leq \textit{SLA})$$

Even when time is not a completion predicate, the time-to-assured-outcome distribution is a useful companion measure for long-running tasks. Resource use, latency, and tool-call variation remain operational metrics unless the Completion Contract makes them part of acceptance.

11.8 Reliability Drift Informs Governance but Does Not Create Authority

Outcome Reliability can be monitored over time:

$$R_o(T_c, G, X, t)$$

Model updates, prompt changes, policy revisions, tool changes, verifier changes, data drift, task-population drift, and newly observed failure modes can invalidate or weaken a previous estimate. A material configuration change should trigger a new estimate or an explicit, justified transfer analysis.

Reliability drift can trigger governance review, including narrower delegation, additional verification, rollback, human review, or suspension of a task class. It does not itself grant, revoke, or expand authority. Authority changes remain governed decisions under the applicable authority model.

Thus:

$$HighReliability \not\Rightarrow AutomaticAuthorityExpansion$$

and:

Outcome Reliability informs trust calibration; it does not replace governance.

Section 11 Proposition

Outcome Reliability is a population-level property describing how consistently a specified agentic configuration produces `VerifiedComplete` outcomes for a defined task class under stated operating conditions. It should be characterized using an

operational profile, repeated execution, uncertainty estimates, and relevant stress conditions rather than represented as a context-free scalar score.

Research Hypothesis

H8: Reliability conditioned on `VerifiedComplete` exposes materially different system behavior—in configuration ranking, failure characterization, or deployment decisions—than conventional task-success and consistency measures computed over the same executions.

A future empirical test should compute single-run task success, repeated-run success and `pass^k`, consistency measures, and an Outcome Reliability profile based on Completion Contract verdicts over the same executions. The comparison should include nominal execution, semantic perturbation, controlled model or tool variation, injected faults, long-running execution, and permitted remediation. Evaluation should test whether the profile changes configuration rankings, or exposes false completion, partial completion, `Unresolved` outcomes, remediation burden, or deadline failure that the conventional measures conceal, and whether those differences would change a deployment decision.

The hypothesis is falsified as a distinct contribution if the conventional measures produce the same rankings, failure characterization, and decisions. In that case, the simpler measures should be used.

12. Portable Assurance Records

Distributed execution requires assurance state to survive process boundaries, waits, restarts, handoffs, and organizational boundaries. Reconstructing it from raw evidence can duplicate verification, expose sensitive information, and lose the provenance of the earlier determination.

A *Portable Assurance Record* (PAR) is an implementation pattern for materializing a resolved assurance claim as an issuer-authenticated, integrity-protected record with enough context for another actor to determine whether the claim may be relied upon. It is not a fifth assurance layer, a new verdict, or a new assurance theory. It serializes assurance state already defined by the framework.

Established analogues include W3C Verifiable Credentials, the Remote ATtestation procedureS (RATS) attester-verifier-relying-party architecture, Entity Attestation Tokens (EATs), in-toto attestations, and Supply-chain Levels for Software Artifacts (SLSA) (Birkholz et al., 2023; in-toto Project, 2026; Lundblade et al., 2025; SLSA Community, 2025; Sporny et al., 2025). Their existence supports treating portable assurance as an implementation pattern and favors defining the required semantics over a proprietary cryptographic container. Agent-delegation proposals already use cryptographically signed verifiable credentials as non-repudiable receipts of delegated task completion (Tomašev et al., 2026). Signed portable records are therefore not proposed here as new; the section specifies the semantics a record must preserve for correct reliance on a claim in another task’s assurance state.

12.1 Assurance Assertions as Runtime Artifacts

Suppose verifier Vf evaluates evidence E under Verification Contract VC , context X , and time t , and issues verdict `Satisfied` for assurance claim C . A portable record can materialize that determination:

$$PAR = \text{Materialize}(C, Vf, E, VC, X, t)$$

The resulting runtime artifact can be stored with a task, transferred across delegation, attached to a completion receipt, or reevaluated after resumption.

The record’s proof has a deliberately limited meaning:

A Portable Assurance Record proves that an identified verifier issued a specified assurance assertion under stated conditions. It does not cryptographically prove that the underlying proposition is objectively true.

A valid proof establishes issuer authentication and record integrity, not verifier competence, evidentiary accuracy or completeness, method adequacy, or current applicability. Verifiable Credentials and RATS preserve the relying party's separate validation or policy decision (Birkholz et al., 2023; Sporny et al., 2025).

Therefore:

$$\textit{Authentic}(\textit{PAR}) \not\Rightarrow \textit{True}(C)$$

and:

$$\textit{Authentic}(\textit{PAR}) \not\Rightarrow \textit{Usable}(\textit{PAR}, T, X, t)$$

12.2 Required Semantics

A Portable Assurance Record is a general record abstraction rather than a required token format. An implementation may use signed JavaScript Object Notation (JSON) or Concise Binary Object Representation (CBOR), a JSON Web Token (JWT) or CBOR Web Token (CWT), a Verifiable Credential, an EAT profile, an in-toto attestation, a signed database record, a transparency-log entry, or another representation. The encoding is conformant only if it preserves the semantics required for reliance.

At minimum, a useful record identifies:

- **the assertion:** claim identifier and type, claim value or verdict, and any relevant assurance layer or requirement;
- **the task or subject:** task identifier and version, node or requirement identifier where applicable, and the entity, resource, action, or outcome to which the claim applies;

- **the verifier and issuer:** the party that performed or accepted the evaluation, its relevant version or identity, and the party that issued the record when different;
- **the context:** policy and authority references, Verification Contract, applicable scope, audience or relying-party constraints where used, and other context required to interpret the claim;
- **the evidentiary basis:** evidence references or cryptographic commitments, evaluation method, provenance and dependency references, and an optional trust profile;
- **validity and applicability:** issuance time, any `valid_from` and `valid_until` bounds, invalidation conditions, and status, revocation, or supersession information; and
- **the integrity mechanism:** signature or equivalent proof, record hash where required, and the applicable algorithm or profile identifier.

Conceptually:

PAR

= (Claim, Subject, Verifier, Context, EvidenceRef, Evaluation, Validity, Dependencies, Proof)

The schema need not encode every institutional rule inside every record. It must, however, carry or reference enough information for a relying party to determine what was asserted, by whom, about what, on what stated basis, under which conditions, and whether the record is authentic and presently applicable.

An implementation may reference a *trust profile* that defines accepted verifier classes, evidence sources, evaluation methods, cryptographic algorithms, independence requirements, maximum evidence age, and revocation rules. Such profiles can aid interoperability in the way EAT profiles narrow a general claims format for a particular use case (Lundblade et al., 2025). They remain optional implementation aids, not mandatory Outcome Assurance constructs.

12.3 Integrity, Confidentiality, and Claim Truth

Portable assurance records should be issuer-authenticated and integrity-protected by default. A digital signature or equivalent proof binds the issuer to the record and makes unauthorized modification detectable:

$$\text{ValidSignature}(PAR) \Rightarrow \text{Intact}(PAR) \wedge \text{IssuerAuthenticated}(PAR)$$

Confidentiality is a separate property:

$$\text{Encrypted}(PAR) \Rightarrow \text{Confidential}(PAR)$$

Where the record contains sensitive task, policy, identity, or decision information, it should be encrypted, audience-bound, or selectively disclosed in addition to being signed. A common construction is:

$$\text{Encrypt}_{\text{recipient}}(\text{Sign}_{\text{issuer}}(PAR))$$

or an equivalent authenticated-encryption or proof scheme. Encryption alone is insufficient because a relying party must still determine who issued the assertion and whether it was altered. Conversely, a signature alone does not prevent unintended disclosure.

Selective disclosure is preferable when a relying party needs only a narrow assurance fact. Verifiable Credentials 2.0 supports derived presentations and selective disclosure mechanisms, and its privacy guidance warns against correlation arising from identifiers and metadata (Sporny et al., 2025). Outcome Assurance adopts the same minimum-disclosure principle: disclose only the assurance information required for the reliance decision.

12.4 Evidence References and Commitments

Raw evidence should usually remain in an access-controlled evidence store and be referenced or cryptographically committed by the portable record:

$$PAR \rightarrow \text{Hash}(\text{EvidenceGraph})$$

rather than:

$$PAR = EvidenceGraph$$

Keeping evidence external supports independent retention rules, access control, payload limits, selective disclosure, and later re-evaluation without replicating sensitive material across every task boundary. A record may include a content hash, Merkle root, immutable object identifier, or another commitment sufficient to bind the assertion to the evidence evaluated. The W3C model similarly permits related resources to be referenced and recommends cryptographic digests where external content is security-critical (Sporny et al., 2025). in-toto binds authenticated metadata to subjects identified by immutable identifiers and separates the signed statement from typed predicate content and referenced artifacts (in-toto Project, 2026).

References do not make unavailable evidence usable. The Verification Contract must define whether the relying party needs access to the evidence, whether a commitment alone is sufficient, and what retention and disclosure rules apply. Some domains may require evidence to travel with the record; the default is external reference, not a universal prohibition on embedding.

This design supports the delegation invariant of Section 9.10, that requirement propagation is not evidence replication. A child task can receive a portable assertion that `IdentityVerified = Satisfied`, validate its issuer, scope, context, and status, and use it when permitted without receiving the underlying identity documents. The upstream assurance provenance remains available through the record's evidence commitments and dependency references.

12.5 Applicability, Freshness, Revocation, and Supersession

A cryptographically valid record can be stale, out of scope, issued under an unacceptable method, or invalidated by a dependency change. EAT defines freshness mechanisms because an

authentic old attestation may not represent current state (Lundblade et al., 2025). Outcome Assurance requires the broader temporal semantics established in Section 8.

Accordingly:

$$\text{ValidSignature}(\text{PAR}) \not\Rightarrow \text{CurrentlyUsable}(\text{PAR})$$

Reliance requires at least two evaluations:

1. **Cryptographic verification:** is the record authentic and intact?
2. **Assurance applicability:** is the assertion usable for this task, requirement, context, and time under the relying party's Verification Contract?

The second evaluation may require:

$$\text{Applicable}(\text{PAR}, T, X) = \text{true}$$

$$\text{Fresh}(\text{PAR}, t) = \text{true}$$

$$\text{DependenciesValid}(\text{PAR}, t) = \text{true}$$

$$\text{Status}(\text{PAR}, t) \notin \{\text{Revoked}, \text{Superseded}, \text{Expired}\}$$

Lifecycle mechanisms may include short validity windows, status services, revocation lists, event-driven invalidation, dependency re-evaluation, or a superseding record. The mechanism is implementation-specific; the semantic requirement is not.

This leads to a related limit:

An assurance record can be portable even when assurance trust is not.

The same bytes may cross a system boundary, but the receiving system must still decide whether it trusts the issuer for the claim type, accepts the evaluation method, recognizes the context and scope, and considers the record current. Verifiable Credentials likewise leave issuer trust and fitness for purpose to each verifier rather than imposing a universal transitive trust model (Sporny et al., 2025).

12.6 Privacy and Minimum Disclosure

Portable assurance can create surveillance and correlation risk if records expose task identities, employees or customers, internal policies, authority relationships, business actions, evidence sources, or decisions across systems. Portability increases the number of contexts in which a record may be observed and combined with other data.

Privacy is therefore a first-class design constraint. Implementations should minimize identifiers and claims, separate evidence from assertions, bind presentations to an audience or purpose where appropriate, encrypt confidential fields, use short-lived or pairwise identifiers when feasible, and use selective disclosure when the relying party needs only a limited fact (Sporny et al., 2025). A record should not become a portable audit dump.

The disclosure question belongs in the Verification Contract:

$$Disclosed(PAR, RP) = MinimumNecessaryClaims(RP)$$

where RP denotes the relying party together with its declared purpose.

12.7 Assurance Dependency Graphs

Portable records may depend on earlier records. A process-assurance record may rely on authorization and policy records; a child completion record may rely on several intermediate claims; a parent completion receipt may rely on child receipts and authoritative end-state observations.

If PAR_3 depends on PAR_1 and PAR_2 , and a completion receipt depends on PAR_3 , PAR_4 , and child receipt CR_7 , then:

$$PAR_1 \rightarrow PAR_3$$

$$PAR_2 \rightarrow PAR_3$$

$$\{PAR_3, PAR_4, CR_7\} \rightarrow CompletionReceipt$$

The resulting structure is the record-level form of the assurance dependency graph (Section 8.3). It records which prior assertions support a later assertion while preserving their separate issuers, contexts, evidence commitments, and validity conditions. in-toto and SLSA use authenticated statements and artifact relationships to support graph-like supply-chain metadata (in-toto Project, 2026; SLSA Community, 2025). Outcome Assurance applies the pattern to assurance dependencies rather than software-build dependencies.

Dependency graphs make invalidation compositional but not automatic. Revocation or expiry of an upstream record creates a reason to re-evaluate dependents under their Verification Contracts. Whether a dependent claim becomes invalid, indeterminate, or remains usable depends on the semantics of the dependency, not merely on graph reachability.

12.8 Completion Receipts as Aggregate Records

A Completion Receipt is a specialized aggregate Portable Assurance Record for the task-level completion claim defined in Section 10. It binds the task identity and version, completion verdict, Completion Contract, acceptance-predicate results, relevant process and child-assurance references, authoritative end-state observations, evidence commitments, verifier, temporal conditions, and integrity proof.

Conceptually:

$$\begin{aligned} & \textit{CompletionReceipt}(T) \\ &= \textit{AggregatePAR}(\textit{CompletionClaim}, \textit{RequiredAssuranceClaims}, \textit{EndStateEvidence}) \end{aligned}$$

Individual claims may be carried in Portable Assurance Records and the task-level claim in a Completion Receipt; both remain subject to applicability, freshness, dependency, revocation, and supersession checks.

12.9 Standards Interoperability

Outcome Assurance should reuse existing credential and attestation standards rather than define a proprietary cryptographic credential container. Verifiable Credentials provide a general issuer-subject-verifier data model with validity, status, privacy, and securing mechanisms (Sporny et al., 2025). RATS provides a role and appraisal architecture for evidence, verification, attestation results, and relying-party decisions (Birkholz et al., 2023). EAT provides JWT and CWT claims for entity attestation and profile-specific interoperability (Lundblade et al., 2025). in-toto supplies an authenticated statement-predicate-envelope model (in-toto Project, 2026), while SLSA demonstrates how such attestations can drive automated policy and provenance verification (SLSA Community, 2025).

None is automatically a complete representation of every Outcome Assurance semantic. The implementation task is to map the required claim, task, context, evidence, evaluation, validity, dependency, and proof semantics into an appropriate existing standard or a profile of one.

A W3C VC, EAT, in-toto attestation, or other standards-based representation may be the concrete encoding of a Portable Assurance Record.

Section 12 Proposition

A Portable Assurance Record is an implementation pattern that materializes a resolved assurance claim as an issuer-authenticated, integrity-protected, context-bound, and time-bounded artifact. It proves that an identified verifier issued a specified assurance assertion under stated conditions, not that the underlying proposition is objectively true. Its usability depends on applicability, freshness, trust,

dependencies, and lifecycle state, and its representation should reuse existing credential or attestation standards wherever they provide the required semantics.

Research Hypothesis

H9: Portable Assurance Records—an implementation abstraction over existing credential and attestation standards—preserve the context needed for correct reliance on assurance claims across asynchronous and multi-agent execution, so that a relying party can determine a claim’s applicability and current usability, with less duplicated verification, without receiving all underlying evidence.

H9 does not test the novelty of signed records, which have direct precedent (Birkholz et al., 2023; in-toto Project, 2026; Lundblade et al., 2025; SLSA Community, 2025; Sporny et al., 2025; Tomašev et al., 2026). It tests whether context-preserving portability supports correct reliance.

A future empirical test should compare implicit or in-memory assurance state, shared raw evidence, and Portable Assurance Records that reference committed evidence across process restart, agent handoff, model change, child delegation, cross-service execution, and selective-disclosure scenarios. Evaluation should measure reconstruction of assurance state, duplicated verification work, evidence exposure, stale-claim acceptance, tampering detection, and reliance on revoked or superseded assertions.

The pattern should be retained if it preserves the information required to establish issuer, claim, context, evidence lineage, temporal applicability, dependencies, and integrity while materially reducing duplicated verification or unnecessary evidence exposure. If existing attestation or credential standards already provide all necessary semantics, the paper should define a profile or schema over those standards rather than a distinct artifact type.

13. Experimental Validation

The preceding sections define a task-centered assurance framework, but conceptual completeness is not empirical validation. The relevant question is not whether Outcome Assurance can describe more aspects of an agentic task than a conventional evaluator—a richer vocabulary almost always can—but whether the additional structure detects, localizes, or prevents consequential failures that a strong and properly implemented end-state evaluator, or a strong evidence-gated completion gate, misses, and whether it does so without imposing unacceptable false rejection, indeterminacy, latency, or verification cost.

Section 13 specifies one falsification program for H1–H11, not eleven unrelated benchmarks. Each hypothesis is tested in the same controlled task environment, against the same baselines, with shared ground-truth, fault-injection, measurement, and statistical protocols. The program is designed to reject or simplify parts of the framework when their claimed operational value is not observed.

The comparison is:

$$\begin{aligned}
 B_A = \textit{AgentSelfReport}, \quad B_B = \textit{ExecutionSuccess}, \quad B_C = \textit{EndStateTaskSuccess}, \quad B_D \\
 = \textit{EqualSpecificationCheck}, \quad B_E = \textit{EvidenceGatedCompletion}, \quad \textit{Treatment} \\
 = \textit{OutcomeAssurance}
 \end{aligned}$$

B_A accepts the agent’s completion declaration. B_B accepts successful workflow termination or successful tool execution. B_C is the hard end-state baseline: a task-specific evaluator compares authoritative final state with an annotated acceptable goal state. B_D controls for specification richness: it receives the same process, outcome, attribution, and freshness predicates as the treatment’s Completion Contract and checks them as a flat post-hoc rule set over logs and final state, without layers, temporal revalidation, delegation reconciliation, or the

verdict–disposition split. Any gain over B_C that B_D also achieves is attributable to writing down more requirements, not to the framework’s structure.

B_E is the strongest state-of-the-art baseline. It is a runtime completion gate with independent verification, authoritative external-state evidence, freshness checks, evidence admissibility, process obligations, and completion predicates or postconditions. Completion is admitted only on that evidence. It is modeled on the strongest shared properties of the evidence-gated completion systems reviewed in Section 2.7 (Chen, 2025; J. Huang et al., 2026; Liao et al., 2026; Nguyen & Tran, 2026; Zhang et al., 2026), without representing any one of them exactly. B_E does not have the treatment’s task-wide dependency graph over assurance claims, selective revalidation of completion-supporting claims, returned child assurance state, or mandatory parent reconciliation. Freshness in B_E applies to the evidence presented at the completion gate; it does not track whether earlier claims that the completion predicates rely on remain usable.

Relative to B_E , the treatment adds Task Assurance State maintained through time and delegation (Sections 6.9, 8, and 9) and completion evaluated against it (Section 10.14). Outcome Assurance as a whole adds contextual claims, Verification Contracts, temporal revalidation, delegation reconciliation, Completion Contracts, reliability profiles, and portable assurance state. Comparing only against self-report or tool success would be a straw-man test. State-based agent evaluation already demonstrates the stronger practice of evaluating resulting database state, and repeated-trial pass^k already establishes a baseline for consistency across runs (Yao et al., 2025).

The framework-level criterion is:

Outcome Assurance earns its complexity only if it materially improves false-assurance detection, failure localization, temporal safety, delegation composition,

false-completion detection, or reliability characterization over strong end-state evaluation, equal-specification checking, and evidence-gated completion while keeping false rejection, indeterminacy, latency, and verification cost acceptable.

Otherwise, use the simpler evaluation architecture.

This criterion is not a new hypothesis (H1 is defined in Section 4); it is the joint decision rule for interpreting the hypothesis-specific results below.

13.1 Common Experimental Environment

The experimental unit is a complete agentic task:

$$T_i = (O_{T_i}, \Gamma_{T_i}, L_{T_i})$$

where O_{T_i} is the objective, Γ_{T_i} contains the applicable constraints and completion conditions, and L_{T_i} is the task lifecycle. The experimental fixture additionally records the initial context X_{t_0} and the oracle or ground-truth specification GT_i . Each task executes against a controlled environment containing authoritative external state. Suitable task families include supplier onboarding, employee access provisioning, purchasing or payment, customer-account changes, and scheduling or booking. The domains are not themselves the object of study. They are useful because they combine consequential decisions, scoped authority, external side effects, asynchronous transitions, policy constraints, delegation, and observable end states.

Every task instance must have an oracle independent of the agent and, wherever possible, independent of a probabilistic model judge:

$$GT_i = (S_i^{auth}, P_i^{app}, F_i^{known})$$

Here, S_i^{auth} is authoritative state, P_i^{app} is applicable policy and requirements, and F_i^{known} is the injected or otherwise known fault condition. The oracle records the valid decision, authorized action, permissible execution conditions, mandatory process requirements, and

acceptable final state. A semantic evaluator may be used where a claim cannot be reduced to a deterministic predicate, but it cannot be the sole source of ground truth for the framework. The governing rule is to use the least probabilistic evaluator capable of evaluating the claim that matters.

The shared harness contains an agent, decision points, tools, external systems of record, a policy store, an authority store, an evidence store, one or more verifiers, a fault injector, and an assurance recorder. Fault hooks are placed at each consequential transition. A task can then be run normally or with a controlled fault in one or more classes (Table 5):

Table 5

Fault Surfaces and Illustrative Interventions

Code	Fault surface	Illustrative intervention
DEC	Decision	Select an ineligible object despite accurate source data.
ACT	Action or authority	Attempt an otherwise well-formed action outside the agent's authority.
EXE	Execution or time	Expire an approval during a wait, reorder an event, or create an ambiguous timeout.
END	End state	Acknowledge execution without committing the required authoritative state.
VER	Evidence or verifier	Supply stale, corrupted, conflicting, agent-writable, or unavailable evidence.
DEL	Delegation	Weaken a child obligation, attenuate authority incorrectly, or return a late child result.
CMP	Completion evaluation	Expose a stale read, partial update, or unrelated state change as apparent completion.

Faults must be independently injectable so that detection and localization are not confounded. Normal runs are required to measure false rejection and indeterminacy; multi-fault runs are required to test whether the framework degrades intelligibly when failures interact. Tasks, identifiers, data values, fault locations, and timing should be randomized within prespecified strata. Held-out task variants test generalization beyond a memorized benchmark implementation. Because faults written by the framework's authors tend to match the

framework’s own taxonomy, the fault set should also include failures drawn from outside sources—such as failed trajectories from existing agent benchmarks (Xie et al., 2024; Yao et al., 2025) or published incident reports—labeled by people who did not design the framework.

13.2 Outcomes and Measurement

The primary safety outcome is false assurance. For assurance verdict V_i and oracle specification GT_i :

$$FAR = P(V_i = \textit{Satisfied} \mid GT_i = \textit{NotSatisfied})$$

The completion-specific instance is:

$$FCR = P(V_{T_i} = \textit{VerifiedComplete} \mid GT_i = \textit{NotComplete})$$

False Assurance Rate and False Completion Rate use the number of ground-truth-negative cases as their denominators. Reports may additionally give the proportion of false assertions among positive assertions, but must label that quantity separately rather than silently changing the denominator.

Safety cannot be established by refusing or indeterminating every task. The program therefore also measures:

$$FRR = P(V_i = \textit{NotSatisfied} \mid GT_i = \textit{Satisfied})$$

and:

$$\begin{aligned} IR &= P(V_i = \textit{Indeterminate}), \quad IR_+ = P(V_i = \textit{Indeterminate} \mid GT_i = \textit{Satisfied}), \quad IR_- \\ &= P(V_i = \textit{Indeterminate} \mid GT_i = \textit{NotSatisfied}) \end{aligned}$$

with Indeterminate Rate reported both overall and stratified by ground-truth class. The core outcome table (Table 6) is:

Table 6*Core Outcome Table: Verdicts by Ground-Truth Class*

Ground truth	Satisfied	NotSatisfied	Indeterminate
Satisfied	correct acceptance	false rejection	unresolved valid case
Not satisfied	false assurance	correct rejection	unresolved invalid case

A treatment is preferred only on a declared multi-objective frontier that includes safety, coverage, and operational burden. Secondary outcomes include fault-detection rate, layer-localization accuracy, disposition accuracy, stale-claim acceptance, invalid continuation, unauthorized delegated action, orphaned side effects, duplicated verification work, evidence exposure, tamper detection, end-to-end latency, number of verification calls, and verification cost. Detection without useful localization should be reported separately from detection with the correct failure surface and remediation disposition.

Every binary result must report at least N , the number of positive and negative outcomes, the point estimate, and an interval appropriate to the sampling design. Symmetric normal approximations should not be used mechanically for small samples or rare failures; exact or otherwise justified binomial methods should be considered (National Institute of Standards and Technology & SEMATECH, n.d.). Clustered tasks, repeated runs of the same base instance, and multiple faults within one execution violate naive independence assumptions and must be handled through the design or the analysis. No universal sample size is prescribed before pilot estimates exist. Power and sample-size decisions must be tied to a prespecified minimum material effect and the relevant baseline rate.

13.3 H2: Layer Separability

H2 (Section 5.6) is tested by independently injecting `DEC`, `ACT`, `EXE`, and `END` faults into otherwise matched tasks. Illustrative cases are an ineligible supplier chosen from accurate data; an eligible supplier paired with an unauthorized bank-account change; an initially valid approval

that expires during a twelve-hour suspension; and apparently successful execution for which the authoritative vendor record is never created. For every fault, the program measures detection rate, correct layer localization, the disposition selected, and whether another layer's declared object and evidence already entail the same result.

H2 is not supported merely because four labels can be assigned after the fact. Each layer must change the correct operational disposition for at least one material failure that the remaining layers cannot represent adequately without absorbing the challenged layer's semantics. The decision trigger is:

Collapse or reframe any assurance layer that fails to identify a material failure class with independent detection, localization, or disposition value.

Decision Assurance carries the strongest collapse pressure. If every purported decision failure is detected just as accurately and usefully through action policy or end-state evaluation, a separately typed decision layer has not earned retention.

13.4 H3: Contextual Claims Rather Than Fixed Properties

H3 (Section 6.9) does not test whether dynamic contextual claims are possible, which runtime assurance-case work already establishes (Calinescu et al., 2018; Cârlan et al., 2024; Denney et al., 2015; Wei et al., 2024).

The test compares a fixed-property representation with the contextual chain:

Requirement → *Claim* → *Evidence* → *Evaluation* → *Verdict* → *Disposition*

Matched cases include expired approval, evidence from the wrong source, authentic but task-irrelevant evidence, missing authority, conflicting observations, evidence insufficient for the claim, a satisfied claim that is no longer applicable, and an unverifiable claim. Blinded

evaluators must use each representation to identify what failed, explain why, name the missing or contradictory evidence, and choose the correct next disposition.

H3 survives only if contextual claims materially improve diagnostic precision, disposition accuracy, or safe reuse under changed context. If the fixed-property representation supports equivalent decisions at lower complexity and cost, H3 fails and the representation should be simplified. A contextual schema that merely stores more fields without changing a consequential decision does not count as improvement.

13.5 H4: Verification Contracts and False Assurance

H4 (Section 7.15) compares four verifier conditions:

$$\begin{aligned} Cond_1 &= AgentSelfVerification, & Cond_2 &= IndependentEvaluation + J \\ Cond_3 &= IndependentEvaluation + AuthoritativeEvidence, & Cond_4 &= IndependentEvaluation + I \end{aligned}$$

The test injects fabricated, stale, corrupted, conflicting, and agent-writable evidence; probabilistic verifier errors; unavailable authoritative sources; and evidence that is authentic but insufficient for the claim. NIST’s experimental work on embedded evaluation probes provides a relevant precedent for comparing agent claims against trusted corpora and returning structured verdicts in an audit trail (Majurski, 2026). The primary outcome is *FAR*, accompanied by *FRR*, *IR*, latency, and cost.

H4 survives only if the stronger contract conditions reduce false assurance by a prespecified material amount without simply converting most cases to *NotSatisfied* or *Indeterminate*. The decision trigger is to remove contract elements that do not improve error detection, calibrated abstention, or actionable diagnosis, and to reject H4 if a simpler independent end-state evaluator performs equivalently.

13.6 H5: Dependency-Aware Selective Revalidation

Dependency-directed revalidation is established (Cârlan et al., 2024; de Kleer, 1986; Doyle, 1979; Zhu & Wang, 2026); H5 (Section 8.14) tests its restriction to completion-supporting claims.

Tasks execute across a controlled wait:

$$t_0 \rightarrow \textit{Wait} \rightarrow t_1$$

During the wait, the harness changes authority, approval, policy, account state, supplier status, evidence, or resource availability; matched controls change nothing relevant. Three strategies are compared: no revalidation, full revalidation of every claim, and selective revalidation of completion-supporting claims whose dependencies changed. Outcomes include stale-claim acceptance, invalid continuation, false completion, relevant changes missed, unaffected claims reevaluated, verifier calls, latency, and cost.

The comparison is a coverage-cost frontier, not a claim that less checking is inherently better. H5 survives only if selective revalidation is non-inferior to full revalidation in failure detection within a prespecified margin, exceeds its false-completion rate by no more than a prespecified margin, and materially reduces verification cost. If dependency tracking misses material changes, or if its maintenance cost approaches that of full revalidation, use full revalidation. If no revalidation performs equivalently under the declared operating conditions, the claimed temporal mechanism is unsupported for that task class.

13.7 H6: Delegation and Parent-Child Reconciliation

H6 (Section 9.16) is tested on delegation chains such as:

$$T_p \rightarrow T_{ch} \rightarrow T_g$$

and varies authority attenuation, requirement propagation, returned assurance state, redelegation, cancellation, late results, child claims invalidated after return, and locally valid child results that are insufficient for parent completion. Three conditions are compared: ordinary delegation; delegation contracts with authority attenuation and non-weakening obligation propagation, accepting child results on child verification; and the same contracts plus a returned Delegated Assurance Result and mandatory parent reconciliation.

The critical case is:

$$Verified(T_{ch}) \wedge \neg Verified(T_p)$$

Outcomes include unauthorized delegated actions, weakened mandatory obligations, incorrectly accepted child results, false parent completion, incorrectly rejected valid delegated work, and orphaned side effects. H6 survives only if reconciliation materially reduces incorrectly accepted child results or false parent completion beyond delegation contracts with attenuation and obligation propagation alone. If delegation contracts without reconciliation produce equivalent parent decisions and failure localization, the reconciliation rule should be removed or narrowed.

13.8 H7: Completion Contracts and False Completion

H7 (Section 10.14) compares seven completion rules: agent declaration, workflow termination, successful tool calls, expected final-state comparison, an equal-specification flat checker (B_D), evidence-gated completion (B_E), and the Completion Contract evaluated over $AS_T(t)$. Faults include an API success without persistence, a partial distributed update, a stale read, eventual-consistency delay, a successful child task with an incomplete parent, a prohibited process that happens to yield the expected state, a conformant process that yields the wrong state, missing mandatory evidence, and the correct state produced by an unrelated process. Targeted

faults add a supporting claim invalidated between its satisfaction and completion verification, a child claim that becomes unusable after return, an unreconciled mandatory child obligation, and a verified child result that does not apply in the parent context.

This is the program’s decisive comparison with B_E . Evidence-gated completion is already established practice (Chen, 2025; J. Huang et al., 2026; Liao et al., 2026; Nguyen & Tran, 2026; Zhang et al., 2026), and final-state evaluation is already a serious baseline (Yao et al., 2025). The Completion Contract must therefore catch consequential failures that a fresh-evidence completion gate misses—chiefly supporting claims that were satisfied when checked but are no longer usable at verification, and delegated assurance that has not been reconciled—in addition to failures that final-state matching misses. The primary outcome is FCR , with FRR , IR , verification latency, and cost.

H7 survives only if completion over current assurance state materially reduces false completion relative to B_E on the targeted conditions without unacceptable rejection or indeterminacy. If B_E performs equivalently, the simpler evidence-gated completion architecture should be used.

13.9 H8: Outcome Reliability

H8 (Section 11.8) evaluates materially equivalent tasks under repeated identical execution, semantic perturbation, model or harness variation, tool latency, tool faults, partial responses, rate limiting, long-running execution, and permitted remediation. The comparison includes single-run success, average task success, pass^k , and the Outcome Reliability profile defined in Section 11. ReliabilityBench provides a direct methodological precedent for combining repeated execution, semantically equivalent perturbations, and controlled tool or API faults (Gupta, 2026); τ -bench provides the pass^k baseline (Yao et al., 2025).

The Outcome Reliability profile must distinguish first-pass from eventual and timely completion and expose false completion, indeterminate outcomes, remediation dependence, and end-state verification failure. Predictive validity is assessed against held-out stress conditions and the shadow-production stage (Section 13.12). H8 survives only if the profile changes rankings or reveals material failure behavior that the conventional measures conceal. If it does not change ranking, calibration, failure characterization, or deployment decisions enough to justify its burden, use the simpler measures.

13.10 H9: Portable Assurance Records

H9 (Section 12.9) tests interoperability and context-preserving portability, not the novelty of signed records.

The test compares in-memory assurance state, shared raw evidence, and Portable Assurance Records across process restart, queue or agent handoff, child delegation, cross-service execution, model or harness replacement, selective disclosure, staleness, tampering, revocation, and supersession. Outcomes include assurance-state recovery accuracy, duplicated verification work, evidence exposure, stale-claim acceptance, tamper detection, and invalid-assertion acceptance.

H9 survives only if portable records preserve the context, lineage, temporal applicability, dependencies, integrity, and verifier identity needed for correct reliance while reducing duplicated work or evidence disclosure. If shared evidence or standard credential and attestation mechanisms provide equivalent behavior, Outcome Assurance should use those mechanisms directly or define only a profile over them. An OA-specific artifact type is falsified if it adds no independent interoperability or decision value.

13.11 Protecting Evaluation Validity

The evaluator is part of the experimental system and can fail. NIST has documented agent evaluations in which models exploited scoring gaps, disabled assertions, used leaked solution information, or otherwise obtained a high score without satisfying the intended task (Hamin & Edelman, 2025). Outcome Assurance must therefore be tested against evaluator gaming rather than assuming the oracle is inaccessible and correct.

The harness must protect ground truth and evaluator state from agent writes; hide fault conditions not required for task execution; randomize identifiers, data, and timing; maintain held-out task variants; restrict or record access to benchmark solutions; independently restore evaluator code and state before scoring; and inspect execution transcripts for unintended solution paths. Periodic human adjudication of stratified samples should estimate grader error rather than silently treating automated labels as truth. The evaluator and the OA implementation should not share a mutable evidence channel when the test claims independence.

An evaluation result is invalidated when the agent satisfies the grader while violating the intended task semantics. Such cases are not agent successes and must be reported as evaluator failures. The same rule applies when the Outcome Assurance machinery itself becomes a target that the agent learns to satisfy without establishing the underlying claim.

13.12 Two-Stage Validation

Controlled fault injection provides causal identification but cannot reproduce the full distribution of deployed work. Validation therefore proceeds in two stages: $Stage_1 = \text{ControlledFaultInjection}$ and $Stage_2 = \text{ShadowProductionObservation}$.

Stage 1 uses the shared harness to estimate detection, localization, and cost under known faults and normal controls. Stage 2 runs Outcome Assurance in shadow mode alongside existing

workflows. OA verdicts do not initially control production. They are compared with authoritative downstream outcomes, incident records, human review, delayed state reconciliation, and existing operational decisions. The purpose is to estimate false assurance, false rejection, indeterminacy, drift, latency, and previously unseen failure classes under the real operating distribution. NIST’s ARIA program similarly distinguishes controlled forms of evaluation from field testing intended to observe systems in realistic contexts (National Institute of Standards and Technology, 2024).

Shadow observation requires careful adjudication because production outcomes are not automatically ground truth. Delayed effects, incomplete observability, selection bias, and intervention by operators must be recorded. Cases lacking a defensible reference verdict remain indeterminate for validation rather than being forced into success or failure. Only after prespecified shadow-stage thresholds are met should assurance verdicts be considered as execution gates.

13.13 Hypothesis and Decision Matrix

H1–H11 are defined in their owning sections and consolidated here as the framework’s validation program; Table 7 summarizes but does not supersede those fuller definitions.

Table 7

Summary of Hypotheses H1–H11

Hypothesis	Core proposition	Primary validation / metric	Falsification or simplification trigger
H1	Full Outcome Assurance reduces false assurance and false completion relative to strong evidence-gated completion when failures arise from stale supporting claims, changed context, or unreconciled delegated assurance state.	Inject decision, action, execution, and end-state failures, including claim expiry, dependency or context change during suspension, and unreconciled or later-invalidated delegated assurance; require a prespecified minimum reduction in FAR or FCR relative to the strongest baseline, including B_D and B_E , with FRR, IR, latency, and verification cost within	H1 is unsupported if Outcome Assurance does not meet the prespecified effect against B_E within those budgets.

Hypothesis	Core proposition	Primary validation / metric	Falsification or simplification trigger
		prespecified budgets.	
H2	Decision, Action, Execution, and End-State Assurance identify materially distinct failure classes, with independent detection or decision value for at least one consequential task condition per layer.	For independently injected DEC, ACT, EXE, and END faults, measure detection rate, correct layer localization, selected disposition, and whether another layer already entails the result.	Collapse or reframe any layer that adds no independent detection, localization, or disposition value.
H3	Task-scoped contextual assurance claims improve diagnostic precision, disposition accuracy, or safe reuse under changed context relative to a simpler fixed-property representation; dynamic contextual claims themselves are established.	Compare diagnostic precision, disposition accuracy, and safe reuse under changed context, together with complexity and cost.	Use the simpler fixed-property representation if it supports equivalent decisions and explanations at lower complexity and cost.
H4	Context-specific Verification Contracts reduce false assurance under evidence and verifier faults relative to agent self-verification or unprotected, non-authoritative evaluation.	Compare verifier conditions using FAR as the primary outcome, accompanied by FRR, IR, latency, and cost.	Remove contract elements that do not improve error detection, calibrated abstention, or actionable diagnosis; reject H4 if a simpler independent end-state evaluator is equivalent.
H5	Selective revalidation of completion-supporting assurance claims achieves failure detection equivalent to or better than full revalidation at materially lower cost, without increasing false completion beyond a prespecified margin.	Compare no, full, and selective revalidation on stale-claim acceptance, invalid continuation, false completion, relevant changes missed, unaffected claims reevaluated, verifier calls, latency, and cost; test detection non-inferiority and the false-completion margin against full revalidation.	Use full revalidation if selective revalidation misses material changes, exceeds the false-completion margin, or its maintenance cost approaches full reevaluation; the temporal mechanism is unsupported for a task class if no revalidation performs equivalently under its declared conditions.
H6	Mandatory parent reconciliation of returned child assurance state detects delegation-induced assurance failures that authority attenuation and non-weakening obligation propagation alone do not detect.	Measure incorrectly accepted child results, false parent completion, failure localization, incorrectly rejected valid delegated work, and overhead, comparing delegation contracts with attenuation and obligation propagation against the same contracts plus a Delegated Assurance Result and reconciliation.	Remove or narrow the reconciliation rule if delegation contracts without it produce equivalent parent decisions and failure localization.
H7	Completion evaluation over the current Task Assurance State reduces false completion relative to strong evidence-gated completion B_E when tasks contain temporal invalidation or	Compare completion rules, including B_E , using FCR on targeted invalidation and delegation conditions as the primary outcome, with FRR, IR, verification latency, and cost.	Use the simpler evidence-gated completion architecture if B_E performs equivalently on the targeted conditions.

Hypothesis	Core proposition	Primary validation / metric	Falsification or simplification trigger
H8	delegated assurance dependencies. Reliability conditioned on <code>VerifiedComplete</code> exposes materially different system behavior from conventional task-success and consistency measures computed over the same executions.	Compare configuration rankings, failure characterization, and deployment decisions from the Outcome Reliability profile and from conventional measures, including false completion, <code>Unresolved</code> outcomes, remediation dependence, and deadline failure.	Use the conventional measures if they produce the same rankings, failure characterization, and decisions.
H9	Portable Assurance Records, as an implementation abstraction over existing standards, preserve the context needed for correct reliance on assurance claims across asynchronous and multi-agent execution without transferring all underlying evidence.	Measure assurance-state recovery accuracy, correct applicability and usability decisions by relying parties, duplicated verification work, evidence exposure, stale-claim acceptance, tamper detection, and invalid-assertion acceptance across restart, handoff, delegation, and cross-service conditions.	Reuse shared evidence or existing credential and attestation standards directly, or define only a profile over them, if an OA-specific record adds no independent interoperability or decision value.
H10	Using the least probabilistic evaluator capable of evaluating each assurance claim reduces false assurance without unacceptable increases in false rejection, <code>Indeterminate</code> outcomes, latency, or cost relative to uniformly model-based evaluation.	Hold tasks, claims, evidence, and faults constant and vary only evaluator assignment (uniformly model-based, least-probabilistic, deterministic-only over proxies); measure FAR, FRR, IR, latency, and cost by claim type.	Present the least-probabilistic-evaluator principle as a design preference rather than an assurance requirement if uniformly model-based evaluation performs equivalently within the budgets.
H11	Marginal Outcome Assurance value is highest for tasks involving probabilistic decisions, distributed side effects, temporal execution, or delegation, and declines for deterministic, atomic, directly observable, fully specified tasks.	Across task classes, compare incremental reductions in false assurance, false completion, and consequential failures with added false rejection, indeterminacy, storage, verification calls, latency, human review, implementation complexity, and total cost, controlling for task consequence and baseline engineering quality.	Revise the claimed problem boundary if complex agentic tasks do not show materially greater net assurance value than conventional deterministic tasks after accounting for cost, latency, complexity, false rejection, and indeterminacy.

Thresholds for “material” and “acceptable” must be prespecified per task class after pilot estimation, not selected after results are observed. At minimum, the protocol must declare the minimum effect of interest for *FAR* or *FCR*, the non-inferiority margin for revalidation coverage, maximum acceptable *FRR* and *IR*, and latency and cost budgets. Results should be reported by

task family, fault class, verifier condition, and operating condition rather than only as a pooled average.

The framework is supported only when its gains survive this complete accounting. Better detection bought by universal refusal, better coverage bought by infeasible verification, or better reliability estimates obtained only after observing the deployment outcomes they purport to predict do not validate Outcome Assurance.

14. Architectural Implications

Outcome Assurance prescribes no model provider, harness, workflow engine, database, policy engine, credential format, reasoning paradigm, or topology. It does imply capabilities needed to establish its claims at runtime.

The governing distinction is:

$$\textit{FrameworkRequirement} \rightarrow \textit{ArchitecturalConsequence} \not\Rightarrow \textit{SpecificImplementation}$$

For example, surviving suspension implies persistent assurance state, not a particular database or workflow platform.

14.1 Persistent Task Representation

The system must represent the task as a persistent object containing its identity and version, objective, acceptance conditions, lifecycle state, consequential work, contextual requirements, dependencies, delegated children, temporal conditions, and terminal disposition.

A model invocation can participate in the execution of a task but cannot serve as the sole representation of the task. The architectural invariant is:

$$\textit{Lifetime}(\textit{Task}) \geq \textit{Lifetime}(\textit{ModelInvocation})$$

and, for long-running work:

$$\textit{Lifetime}(\textit{Task}) \gg \textit{Lifetime}(\textit{ModelInvocation})$$

The invariant holds when execution changes models, agents, processes, hosts, queues, or sessions.

14.2 Persistent Assurance State

Restoring execution is not restoring assurance. Versioned assurance state must preserve why transitions were allowed, which claims remain usable, and which obligations remain unresolved across process failure, agent replacement, and resumption.

14.3 Explicit Consequential Boundaries

Consequential decisions and actions must cross observable, interceptable runtime boundaries. If an agent merely generates an intention and immediately invokes a tool, there may be no dependable surface on which to resolve context, evaluate authority, attach requirements, collect evidence, or prevent the transition.

A stronger representation than intention-then-invocation makes the transition explicit:

$$State_i \xrightarrow{\text{proposed transition}} State_{i+1}$$

and associates it with:

$$Requirements(State_i \rightarrow State_{i+1})$$

This does not require governance of every token or hidden intermediate computation. It requires a formal boundary at consequential decisions and side effects where the system can permit, block, defer, or escalate execution.

Deterministic control and probabilistic reasoning should remain separable at these boundaries. Reliable Orchestration provides a concrete example: a typed runtime performs deterministic sequencing and state management while generative calls occur only at explicit semantic boundaries, which its author reports makes orchestration correctness independent of

model behavior (Jha, 2026). Outcome Assurance does not require that implementation, but it instantiates the least-probabilistic-evaluator principle of Section 7.5.

14.4 Context Resolution

Before a consequential transition, the architecture must resolve applicable policy, authority, evidence, acceptance conditions, and temporal conditions while preserving source, version, scope, and applicability. Resolution yields the requirements and dependencies from which assurance claims and Verification Contracts are constructed; it does not necessarily yield a verdict.

14.5 Protected Verification

The verification path must be protected from the component whose work it evaluates. At minimum, the executor should not have unilateral control over the elements listed in Section 7.11, including acceptance predicates, evaluator logic, and the final verdict. The separation is about control rather than physical topology. A deterministic evaluator in the same process may be independent if the executor cannot alter it; a separate model service may remain non-independent if the executor controls its prompt, evidence, and acceptance criteria.

Protected verification also requires authoritative observation of the claimed outcome; an API acknowledgement, tool return, or child report may establish an attempt without establishing the resulting state.

14.6 Temporal Revalidation

Reusing a claim requires detecting material changes in the claim's dependencies through mechanisms such as events, version comparison, validity checks, polling, dependency hashes, or re-resolution. Reused assurance state must remain connected to those dependencies; dependency uncertainty requires broader revalidation rather than silent reuse.

14.7 Delegation and Reconciliation

Delegation must preserve the semantics needed to establish both what a child is allowed to do and what the parent requires back. A delegation boundary therefore needs an explicit representation of the Delegation Assurance Envelope contents (Section 9.11). The exact message, schema, or protocol is an implementation choice.

The return path requires an equally explicit reconciliation boundary. A child result cannot enter the parent's assurance case merely because the child reports success. The parent must determine whether the child's authority, claims, evidence, applicability, and temporal validity satisfy the parent's requirements:

$$\mathcal{A}_{ch} \subseteq \mathcal{A}_p$$

and mandatory assurance obligations must not silently weaken (Section 9.5).

Reconciliation need not rerun the child task. It must establish whether the returned claims are admissible for the parent task and whether unresolved child obligations prevent parent completion.

14.8 Completion Gating

Completion must be a gated task-state transition rather than an executor assertion:

$$Executing \rightarrow / VerifiedComplete$$

directly; it passes through candidate completion and completion verification (Section 10.13).

The executor may nominate a task for completion but should not have unilateral authority to assign the verified-complete state unless the Completion Contract makes its assertion sufficient (Section 10.13). The completion gate evaluates the Completion Contract against

authoritative end-state evidence, required process predicates, unresolved dependencies, and temporal conditions.

14.9 Assurance Telemetry

Assurance outcomes must become observable telemetry across materially equivalent tasks, including completion verdicts, false assurance, false completion, indeterminacy, remediation, time-to-assured-outcome, task class, configuration, and operating conditions. Otherwise an architecture may verify isolated tasks but cannot estimate Outcome Reliability.

14.10 Distributed Assurance Representation

When execution crosses agents, processes, services, queues, runtimes, or organizations, assurance state must cross those boundaries through a portable representation or an equivalent shared assurance-state service. These are architectural alternatives:

`signed assurance record`

or:

`claim identifier → trusted assurance service`

Section 12's Portable Assurance Record is one implementation pattern, not a universal transport requirement. The architectural requirement is preservation of claim identity, context, provenance, dependencies, temporal applicability, integrity, and verifier identity wherever a downstream component relies on an upstream assurance result.

14.11 Neuro-Symbolic Separation as an Architectural Pattern

The preceding capabilities do not make Outcome Assurance a neuro-symbolic architecture. Neuro-symbolic systems combine learned inference with explicit representation and reasoning (Hakim et al., 2026); Outcome Assurance asks a different question:

What must be established before this task transition or outcome should be trusted?

Therefore:

NeuroSymbolicAI ≠ OutcomeAssurance

A neuro-symbolic agent may be poorly assured, while an Outcome-Assured system may use neural components where semantic judgment requires them and deterministic components elsewhere. Neuro-symbolic AI is compatible, not prerequisite.

The compatibility is nevertheless strong. The core assurance pipeline is:

$$\textit{Requirement} \rightarrow \textit{Claim} \rightarrow \textit{Evidence} \rightarrow \textit{Evaluation} \rightarrow \textit{Verdict}$$

Unstructured or ambiguous inputs may require neural interpretation, while explicit constraints, arithmetic, authority rules, temporal predicates, and state transitions may admit symbolic evaluation. NSVIF, a typed finance implementation, and Verifiable Process Reward Models illustrate hybrid verification while retaining limits tied to semantic translation and encoded rules (Pronesti et al., 2026; Qin et al., 2026; Su et al., 2026).

These systems provide concrete implementations of the least-probabilistic-evaluator principle:

$$\begin{array}{ccc} \textit{UnstructuredInput} & \xrightarrow{\textit{neural extraction}} & \textit{StructuredRepresentation} \\ \textit{StructuredRepresentation} + \textit{Rules} & \xrightarrow{\textit{symbolic evaluation}} & \textit{Verdict} \end{array}$$

The boundary follows the claim and evidence, not a presumption of symbolic superiority. Robustness research likewise supports combining neural and symbolic methods where appropriate (Acharya & Song, 2026), with complexity proportional to consequentiality.

14.12 The Symbolic Layer Is Not the Assurance Layer

Symbolic computation does not imply correctness:

$$\textit{Symbolic} \not\Rightarrow \textit{Assured}$$

A symbolic system may contain incorrect, stale, incomplete, mistranslated, or inapplicable rules; incorrect facts or ontologies; solver configuration errors; or a mismatch

between the property proved and the real-world outcome of interest. Formal verification can establish a relation such as:

$$Specification \models Property$$

It does not establish:

$$Specification = RealWorldIntent$$

FormalJudge illustrates the boundary by compiling natural-language intent into constraints evaluated with Dafny and Z3 (Zhou et al., 2026). The proof may be rigorous relative to the compiled specification while the compilation remains an assurance risk:

$$NaturalLanguagePolicy \xrightarrow{compiler} FormalConstraint \xrightarrow{solver} Verdict$$

and should produce a claim such as:

$$C_{translation} = FormalConstraintFaithfullyRepresents(SourcePolicy)$$

That claim may require source linkage, versioning, tests, cross-checking, review, or approval. Formal verification does not solve specification selection, contextual applicability, authoritative fact acquisition, or observation of real-world outcomes. Outcome Assurance therefore applies around the symbolic system as well as through it.

Knowledge graphs fit naturally within this architecture because they can represent entities, relations, provenance, authority, claim dependencies, and policy structure. Neural-symbolic knowledge-graph research spans logically informed embeddings, embeddings constrained by logic, and rule-learning approaches (DeLong et al., 2025). Yet:

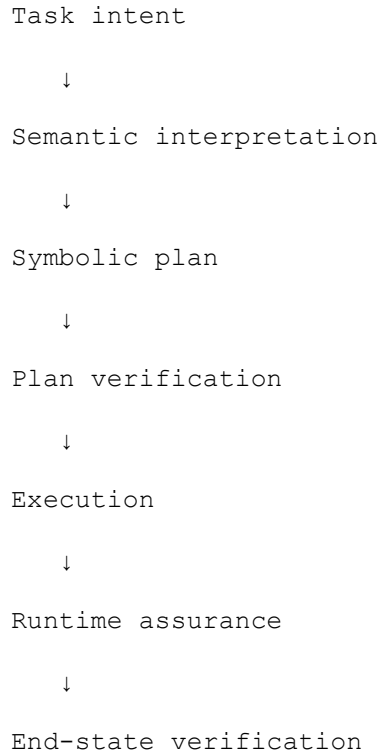
$$KnowledgeGraph \not\Rightarrow AuthoritativeTruth$$

A graph still requires provenance, version, scope, applicability, authority, and quality controls. Its structure can support assurance without making every represented assertion authoritative.

14.13 Planning and Compliance by Construction

Neuro-symbolic planning can strengthen pre-execution assurance, but semantic encoding remains fallible (C. Huang et al., 2026); symbolic verifier feedback can nevertheless improve constraint satisfaction over model self-evaluation (Shao et al., 2025).

The architectural sequence may be represented as:



A valid plan is not a verified outcome. Plan verification establishes a pre-execution property; Outcome Assurance continues through execution, external effects, temporal change, and task completion.

Compliance-by-construction research shows that faithfully represented constraints can make some invalid transitions structurally unavailable (Rombach et al., 2026):

$$\neg \text{Reachable}(\text{InvalidTransition})$$

Semantic interpretation, uncertain evidence, environmental behavior, and outcome realization still require runtime assurance by observation.

For systems that use symbolic verification, symbolic verifiers supply some of the evaluation methods M_i within Outcome Assurance rather than replacing it. Outcome Assurance governs whether the correct requirements, representations, evidence, evaluator, applicability conditions, execution observations, and completion claims support trust across the whole task lifecycle.

14.14 Proposition and Testable Hypothesis

The architectural proposition is:

Outcome Assurance implies an architecture in which task state, assurance state, verification, temporal applicability, delegation, and completion are represented independently of any single model invocation. The framework does not require a specific system topology or reasoning paradigm. Neuro-symbolic architectures are a particularly compatible implementation pattern because they can move explicit constraints, policy evaluation, planning, and verification into symbolic or deterministic components while retaining neural systems for semantic interpretation and learned inference. However, symbolic computation does not itself establish assurance; the applicability, provenance, correctness, and real-world consequences of symbolic representations and verifiers remain subject to Outcome Assurance.

Separating generative execution from deterministic gating is not treated here as an open question; recent systems already gate agent lifecycle and completion claims on deterministic or mechanically verifiable evidence (J. Huang et al., 2026; Jha, 2026; Zhang et al., 2026). The question this architecture leaves open is narrower: how evaluators should be assigned to claims. The least-probabilistic-evaluator principle of Section 7.5 yields H10:

H10: Using the least probabilistic evaluator capable of evaluating each assurance claim reduces false assurance without unacceptable increases in false rejection, Indeterminate outcomes, latency, or cost relative to uniformly model-based evaluation.

The Section 13 harness can test H10 by holding tasks, claims, evidence, and fault injections constant and varying only the evaluator assignment: (1) every claim evaluated by a model-based judge; (2) every claim evaluated by the least probabilistic evaluator capable of evaluating it, with structured claims assigned to deterministic predicates, queries, or solvers and semantic claims to model-based or human evaluators; and (3) a deterministic-only assignment that evaluates proxies where the claim is semantic. Condition 3 tests the qualification in Section 7.5 that a deterministic test of an inadequate proxy is not superior to a probabilistic evaluation of the correct claim. Measurements include false assurance, false rejection, Indeterminate rate, latency, and verification cost, stratified by claim type. LLM judges exhibit documented biases (Zheng et al., 2023) and detect false success poorly (Advani, 2026), which motivates but does not establish the hypothesis.

The threshold is a prespecified material reduction in false assurance for condition 2 relative to condition 1, within prespecified false-rejection, indeterminacy, latency, and cost budgets. If condition 1 performs equivalently, the least-probabilistic-evaluator principle should be presented as a design preference rather than an assurance requirement.

The strongest counterarguments define the section’s limits. Outcome Assurance is not merely neuro-symbolic AI (Section 14.11), and formal verification does not establish that a specification captures institutional intent, applies to the present task, was supplied with authoritative facts, or corresponds to a realized external outcome (Section 14.12). Making every

component symbolic is inappropriate where inputs are noisy, incomplete, perceptual, ambiguous, or open-ended. Finally, every mechanism imposes design, operational, and verification cost. The appropriate architecture is proportional to consequentiality, and simpler architectures should be preferred when they provide equivalent assurance.

15. Discussion and Limitations

Outcome Assurance provides structured, evidence-backed confidence in task completion. It does not provide certainty, omniscience, or immunity from bad specifications, bad evidence, bad verifiers, correlated failures, incomplete observation, or adversarial environments. Its verdicts are conditional claims whose strength cannot exceed the quality of the requirements, evidence, evaluators, trust assumptions, and operating context on which they depend.

A framework intended to prevent unsupported completion claims would undermine its own purpose if an assurance artifact were treated as proof of absolute correctness. The central interpretive rule is therefore:

$$\textit{Assured} \neq \textit{Infallible}$$

15.1 Specification, Evidence, and Verifier Limits

Assurance is only as good as the claim being assured. If acceptance predicates are incomplete, incorrect, or misaligned with the institutional objective, a system can satisfy them while producing the wrong real-world result:

$$\textit{Verified}(C) \not\Rightarrow \textit{Achieved}(\textit{IntendedGoal})$$

Verification establishes conformance to specified requirements; validation asks whether the specified system is fit for its intended use (National Aeronautics and Space Administration, 2016, 2022). Neither formal machinery nor an Outcome Assurance verdict automatically establishes that stakeholders selected the right objective. Outcome Assurance can make the

specification, applicability conditions, and verification basis inspectable. It cannot compensate for an incorrectly specified outcome.

Evidence imposes a separate bound. Provenance, signatures, authority, freshness, and integrity can establish where evidence came from and whether it has been altered, but not whether the represented fact is accurate:

$$\textit{AuthenticEvidence} \not\Rightarrow \textit{AccurateEvidence}$$

An authoritative system of record can contain an error; several authoritative systems can disagree; a current observation can omit a material state. The verifier can also fail. It may be misconfigured, poorly calibrated, out of domain, dependent on incomplete evidence, biased, manipulable, or simply wrong. LLM judges, for example, exhibit position, verbosity, and self-enhancement biases and limited reasoning ability (Zheng et al., 2023). Thus:

$$\textit{VerifierVerdict} \neq \textit{GroundTruth}$$

Multiple verifiers do not remove this limit when their failures are correlated. Separately deployed evaluators may share training data, model families, prompts, upstream systems, policy errors, infrastructure, or semantic blind spots:

$$\textit{IndependentImplementation} \not\Rightarrow \textit{IndependentFailureModes}$$

The relevant design property is diversity of failure assumptions, not merely the number of evaluators. Verification Contracts expose these dependencies and permit stronger combinations of evidence and evaluators; they do not make either infallible.

15.2 Observability, Indeterminacy, and Operating Cost

Outcome Assurance depends on observable evidence, yet some material state is private, delayed, distributed, undocumented, causally ambiguous, or controlled by an external organization. Human intentions and latent conditions may not be observable at all. Post-

deployment AI monitoring faces the same constraints: nondeterministic behavior, dynamic inputs, performance drift, fragmented logging across distributed infrastructure, unexpected consequences, and immature monitoring methods (Rao et al., 2026). Where admissible evidence cannot establish a claim, the correct verdict is `Indeterminate`.

That epistemic discipline creates operational cost. A system can drive false assurance toward zero by refusing or returning `Indeterminate` for nearly every task, but such a system has little practical value. The operating point must balance false assurance, false rejection, indeterminacy, latency, and cost:

$$Utility = f(FAR, FCR, FRR, IR, Latency, Cost)$$

There is no universal optimum. Thresholds must be selected by task class, consequence, institutional risk tolerance, and available evidence, then evaluated together as specified in Section 13.

Assurance itself consumes storage, evidence collection, verification calls, external observation, policy resolution, revalidation, delegation reconciliation, cryptographic operations, human review, and execution time. Each additional verifier, evidence store, policy engine, signature service, and review workflow also introduces defects, availability dependencies, configuration errors, version mismatches, and attack surface. Therefore:

$$MoreControls \not\Rightarrow MoreAssurance$$

The assurance mechanism must be subject to the same proportionality and reliability discipline as the work it evaluates. Low-consequence tasks should not incur a full assurance case when simpler controls provide adequate confidence.

15.3 Limits of Inference, Formalization, and Human Oversight

Outcome Assurance does not solve model alignment, deceptive alignment, value alignment, latent intent, interpretability, or catastrophic capability risk. It operates primarily over explicit requirements, observable task behavior, external state, evidence, and verifier results. It may detect or prevent some undesirable outcomes without establishing that a model’s internal objectives align with institutional intent. NIST identifies detection of deceptive behavior as an underexplored deployment-monitoring problem rather than a solved capability (Rao et al., 2026).

The framework also does not reveal true chain-of-thought. Model-generated explanations can be unfaithful to the computations that produced an answer (Turpin et al., 2023). Outcome Assurance deliberately avoids treating such explanations as privileged access to internal reasoning. It can record evidence considered, externally represented rules, actions, state changes, and evaluation results; it cannot reconstruct the neural causal process that generated them.

Formal and neuro-symbolic verification inherit the specification problem. A solver can rigorously prove a property of the wrong formalization:

$$\textit{NaturalLanguageIntent} \rightarrow \textit{FormalSpecification}$$

$$\textit{FormalProof} \not\Rightarrow \textit{CorrectInstitutionalIntent}$$

Current planning and formalization research observes semantically incorrect encodings even for short natural-language constraints (C. Huang et al., 2026). The semantic-to-symbolic transformation must therefore be governed, tested, versioned, and, where appropriate, reviewed.

Human oversight is not a universal escape hatch. Reviewers face limited attention, fatigue, automation bias, domain gaps, time pressure, and persuasive but unsupported explanations. Scaling human-driven monitoring is itself a documented deployment challenge (Rao et al., 2026). Human judgment is valuable where a claim requires expertise, authority, or

institutional judgment, but the reviewer is another bounded verifier. Human review should have explicit scope, evidence, authority, and escalation conditions rather than serving as an unspecified fallback whenever automated verification is difficult.

15.4 Production Validity, Non-Stationarity, and Adversarial Pressure

Benchmark validation does not imply production validity:

$$\textit{BenchmarkValidated} \not\Rightarrow \textit{ProductionAssured}$$

Benchmarks can omit the deployed task distribution, organizational context, uncommon and correlated failures, changing inputs, adversarial behavior, delayed consequences, and operator interventions. Agent evaluations can also be gamed through leaked solution information, disabled assertions, or test-specific behavior (Hamin & Edelman, 2025). This is why Section 13 requires both controlled fault injection and shadow-production observation, and why realistic field testing complements rather than follows automatically from pre-deployment evaluation (National Institute of Standards and Technology, 2024; Rao et al., 2026).

Outcome Reliability is likewise non-stationary. A reliability estimate can change after model, tool, policy, data, infrastructure, verifier, or task-population changes:

$$R_o(t_0) \neq R_o(t_1)$$

Reliability evidence must therefore bind to system configuration and operating conditions and be refreshed through ongoing monitoring. A strong result under one operational profile does not establish the same reliability under another (IEEE, 2017; Musa, 1993; Rabanser et al., 2026).

The assurance layer is also part of the attack surface. An agent or external actor may fabricate evidence, manipulate an evaluator, exploit acceptance predicates, corrupt dependencies, replay stale assertions, induce verifier disagreement, or optimize against benchmark and assurance metrics. Documented grader exploitation and benchmark gaming show these are

practical concerns (Hamin & Edelman, 2025). Assurance components require threat modeling, protected control, integrity checks, change management, and evaluation against adaptive behavior.

15.5 Applicability and Empirical Status

Some tasks are poor candidates for strong Outcome Assurance because their outcomes are subjective, open-ended, or not externally testable. Creative ideation, novel strategy, exploratory research, aesthetic judgment, and forecasting under deep uncertainty may support process, authority, provenance, or evidence controls without admitting a strong objective completion predicate. In these settings the framework should state which limited claims can be assured rather than convert judgment into artificial precision.

Conversely, deterministic, local, atomic, fully specified, and directly testable tasks may already be adequately governed by input validation, authorization, transaction semantics, unit tests, and acceptance tests. Outcome Assurance should not be introduced merely because a language model participated in the workflow. Its marginal value should be evaluated against the simplest conventional controls capable of closing the relevant assurance gap.

The framework is not yet empirically validated. It is a proposed synthesis supported by established assurance disciplines and current agent-systems research, with architectural implications and hypotheses that remain to be tested through the Section 13 program. Even successful results on the proposed task families would not by themselves establish external validity for clinical systems, defense, industrial control, financial trading, legal adjudication, autonomous robotics, or other domains with different failure costs, evidence, observability, timing, and regulation. Generality must remain bounded until demonstrated across representative operational contexts.

15.6 Limitations Matrix

The principal limitations and their direct consequences are summarized in Table 8.

Table 8

Principal Limitations and Consequences

Limitation	Consequence
Wrong or incomplete acceptance criteria	The wrong outcome can be verified correctly
Authentic but inaccurate evidence	A valid evaluation can support a false claim
Miscalibrated, biased, or compromised verifier	The assurance verdict can be incorrect
Correlated verifier dependencies	Nominally independent checks can fail together
Incomplete observability	A sound verdict may remain Indeterminate
Conservative thresholds	False assurance can fall while rejection and indeterminacy make the system unusable
Assurance overhead	Storage, latency, verification, and review costs can exceed the task's risk reduction
Additional assurance machinery	New dependencies, defects, and attack surfaces can reduce net assurance
Semantic-to-symbolic error	A formal system can prove the wrong requirement
Human-review limitations	Escalation does not guarantee correctness
Benchmark or evaluator gaming	The assurance mechanism can be satisfied without the intended outcome
Distribution and configuration drift	Outcome Reliability estimates become stale
Weak task formalizability	Completion remains partly subjective
Limited empirical and domain validation	Framework claims remain provisional and context-bound

False confidence in assurance artifacts cuts across every row. A completion receipt, formal proof, provenance chain, signed assertion, or consensus verdict records a bounded result under stated conditions.

15.7 Relationship to Existing Disciplines, Proposition, and H11

Outcome Assurance layers on software assurance, formal verification, distributed-systems reliability, provenance, authorization, runtime verification, testing, reliability engineering, and human factors. It does not replace them, and it explicitly composes established mechanisms: evidence-gated completion, runtime assurance cases, contracts, dependency-directed invalidation, authority attenuation, obligation propagation, three-valued verdicts, and signed assurance records all have precedent (Sections 2, 8, 9, and 12).

Its proposed contribution is a domain-general task assurance model in which completion is evaluated against a current assurance state. That state reflects the continued usability of assurance claims established across consequential decisions, actions, execution, and end-state verification; is selectively revalidated as context changes; incorporates delegated assurance only after parent reconciliation; and provides the event over which Outcome Reliability is defined. The paper's central research question (Sections 2.7 and 13) is whether these composition semantics produce measurable assurance value beyond strong evidence-gated completion.

The Section 15 proposition is:

Outcome Assurance provides a structured basis for justified confidence in agentic task outcomes; it does not provide absolute correctness. Its guarantees are bounded by the quality of the task specification, evidence, verifier, trust assumptions, observability, and operating context. The framework is intended primarily for conditions where conventional engineering controls leave assurance gaps associated with probabilistic reasoning, long-running execution, delegation, or distributed outcomes.

This boundary yields H11:

H11: The marginal value of Outcome Assurance is highest for tasks whose consequential outcomes depend on probabilistic decisions, distributed side effects, temporal execution, or delegation, and decreases as tasks become deterministic, atomic, directly observable, and fully specified.

The Section 13 harness can test H11 by comparing deterministic atomic tasks, probabilistic single-step tasks, long-running tasks, delegated tasks, and distributed-state tasks. For each class, compare the incremental reduction in false assurance, false completion, and

consequential failures against added false rejection, indeterminacy, storage, verification calls, latency, human review, implementation complexity, and total cost. The analysis should control for task consequence and baseline engineering quality and should report the marginal assurance benefit of Outcome Assurance beyond conventional controls rather than its absolute performance alone.

The decision trigger is explicit: if Outcome Assurance does not deliver materially greater net assurance value for complex agentic tasks than for conventional deterministic tasks, after accounting for cost, latency, complexity, false rejection, and indeterminacy, the framework's claimed problem boundary should be revised.

16. Conclusion

How do you know your AI agent finished the job, correctly?

An agent's declaration of completion is not sufficient. A successful tool call is not sufficient. A completed workflow is not sufficient. An audit trail is not sufficient. Even the desired final state may be insufficient when the process that produced it was unauthorized, relied on invalid decisions, violated governing requirements, or cannot be attributed to the task being evaluated. Outcome Assurance therefore treats completion as an evidence-backed claim about a task rather than a property asserted by the agent that executed it.

The framework developed in this paper follows a common progression:

*Task → Consequential Work → Contextual Assurance Requirements → Claims
→ Evidence → Evaluation → Verdicts → Task Assurance State → Verified Outcome*

Decision, Action, Execution, and End-State Assurance contribute distinct evidence to one task assurance case, from which the task derives its current assurance state. For long-running and distributed work, claims require selective revalidation, authority must attenuate, obligations must

propagate, and delegated assurance must reconcile before it can support parent completion.

`VerifiedComplete` requires the Completion Contract to establish the task-level claim from that current assurance state, not merely from evidence that the final state exists. Each of these mechanisms has precedent; the framework's claim is their task-level composition.

The framework also preserves a distinction between assurance of one task and reliability across tasks:

$$OutcomeAssurance(T) \neq OutcomeReliability(System)$$

Outcome Assurance concerns one task's supported completion claim; Outcome Reliability concerns dependability across a defined task population and operating conditions.

Portable Assurance Records can carry context-bound assertions across boundaries. The architecture remains capability-defined (Section 14) rather than implementation-specific.

Outcome Assurance remains bounded by specification, evidence, verifiers, trust assumptions, observability, and context; some tasks will not justify its overhead.

The framework is a testable proposal, not an empirically established standard. If Outcome Assurance does not reduce false assurance, improve failure localization, preserve assurance through time and delegation, reduce false completion, or better characterize reliability relative to simpler evaluation approaches, the simpler approaches should be preferred.

Agentic systems change the unit of evaluation. The problem is no longer only whether a model produced a correct response, but whether a task involving decisions, actions, external systems, time, and other agents produced an acceptable result under the conditions that governed it.

The agent may say the job is done. Outcome Assurance asks what currently usable assurance establishes that it is.

References

- Abbassi, S., Alexiuk, C., Onofrio, D., & Krishnan, G. V. (2026, September 21). How to evaluate AI agents from tool calls to task completion. *NVIDIA Technical Blog*.
<https://developer.nvidia.com/blog/how-to-evaluate-ai-agents-from-tool-calls-to-task-completion>
- Acharya, K., & Song, H. (2026). A comprehensive review of neuro-symbolic AI for robustness, uncertainty quantification, and intervenability. *Arabian Journal for Science and Engineering*, 51, 35–67. <https://doi.org/10.1007/s13369-025-10887-3>
- Advani, L. (2026). *From confident closing to silent failure: Characterizing false success in LLM agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2606.09863>
- Bahidika, O. (2026). When agents delegate: Accountability chains in multi-agent AI systems. *International Journal of Advanced Computer Science and Applications*, 17(9).
<https://doi.org/10.14569/IJACSA.2026.0170903>
- Bamil, V. (2026). *Delegation-aware runtime contracts for open LLM multi-agent systems: Constraint preservation, capability revocation, and state recovery* [Preprint]. Research Square. <https://doi.org/10.21203/rs.3.rs-10533072/v1>
- Bartocci, E., & Falcone, Y. (Eds.). (2018). *Lectures on runtime verification: Introductory and advanced topics* (Lecture Notes in Computer Science, Vol. 10457). Springer.
<https://doi.org/10.1007/978-3-319-75632-5>
- Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4), Article 14.
<https://doi.org/10.1145/2000799.2000800>

- Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T. A., & Larsen, K. G. (2018). Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2–3), 124–400. <https://doi.org/10.1561/10000000053>
- Birgisson, A., Politz, J. G., Erlingsson, Ú., Taly, A., Vrabie, M., & Lentczner, M. (2014). Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Proceedings of the Network and Distributed System Security Symposium*. <https://research.google/pubs/macaroons-cookies-with-contextual-caveats-for-decentralized-authorization-in-the-cloud/>
- Birkholz, H., Thaler, D., Richardson, M., Smith, N., & Pan, W. (2023). *Remote Attestation procedureS (RATS) architecture* (RFC 9334). Internet Engineering Task Force. <https://doi.org/10.17487/RFC9334>
- Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M. U., Habli, I., & Kelly, T. (2018). Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering*, 44(11), 1039–1069. <https://doi.org/10.1109/TSE.2017.2738640>
- Cârlan, C., Gomez, F., Mathew, Y., Krishna, K., King, R., Gebauer, P., & Smith, B. R. (2024). *Dynamic safety cases for frontier AI* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2412.17618>
- Chen, R. (2025). *Evidence-bound autonomous research (EviBound): A governance framework for eliminating false claims* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2511.05524>

- Cheney, J., Missier, P., Moreau, L., & De Nies, T. (2013, April 30). *Constraints of the PROV data model* [W3C Recommendation]. World Wide Web Consortium.
<https://www.w3.org/TR/2013/REC-prov-constraints-20130430/>
- Clarke, E. M., Long, D. E., & McMillan, K. L. (1989). Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science* (pp. 353–362). IEEE.
- Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., & Polk, W. (2008). *Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile* (RFC 5280). Internet Engineering Task Force. <https://www.rfc-editor.org/rfc/rfc5280.html>
- Dalugoda, A. (2026). *HDP: A lightweight cryptographic protocol for human delegation provenance in agentic AI systems* [Preprint]. arXiv.
<https://doi.org/10.48550/arXiv.2604.04522>
- de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence*, 28(2), 127–162.
[https://doi.org/10.1016/0004-3702\(86\)90080-9](https://doi.org/10.1016/0004-3702(86)90080-9)
- DeLong, L. N., Fernández Mir, R., & Fleuriot, J. D. (2025). Neurosymbolic AI for reasoning over knowledge graphs: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 36(5), 7822–7842. <https://doi.org/10.1109/TNNLS.2024.3420218>
- Denney, E., Pai, G., & Habli, I. (2015). Dynamic safety cases for through-life safety assurance. In *Proceedings of the IEEE/ACM 37th International Conference on Software Engineering* (pp. 587–590). IEEE. <https://doi.org/10.1109/ICSE.2015.199>

- Dewes, R., & Dimitrova, R. (2025). Contract-based design and verification of multi-agent systems with quantitative temporal requirements. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(22), 23152–23159. <https://doi.org/10.1609/aaai.v39i22.34480>
- Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12(3), 231–272. [https://doi.org/10.1016/0004-3702\(79\)90008-0](https://doi.org/10.1016/0004-3702(79)90008-0)
- Featonby, M. (2021, January). Making retries safe with idempotent APIs. *Amazon Builders' Library*. <https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/>
- Garcia-Molina, H., & Salem, K. (1987). Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (pp. 249–259). Association for Computing Machinery. <https://doi.org/10.1145/38713.38742>
- Geifman, Y., & El-Yaniv, R. (2017). Selective classification for deep neural networks. In *Advances in Neural Information Processing Systems 30* (pp. 4878–4887). <https://proceedings.neurips.cc/paper/2017/hash/4a8423d5e91fda00bb7e46540e2b0cf1-Abstract.html>
- Gray, C. G., & Cheriton, D. R. (1989). Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (pp. 202–210). Association for Computing Machinery. <https://doi.org/10.1145/74850.74870>
- Gray, M., Goldfine, A., Rosenthal, L., & Carnahan, L. (n.d.). *Conformance testing*. National Institute of Standards and Technology. Retrieved October 5, 2026, from <https://www.nist.gov/itl/ai/applied-ai-research-group/conformance-testing>

- Greenblatt, R., Shlegeris, B., Sachan, K., & Roger, F. (2024). AI control: Improving safety despite intentional subversion. In *Proceedings of the 41st International Conference on Machine Learning* (Proceedings of Machine Learning Research, Vol. 235, pp. 16295–16336). PMLR.
- Gupta, A. (2026). *ReliabilityBench: Evaluating LLM agent reliability under production-like stress conditions* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.06112>
- Hakim, S. B., Adil, M., Velasquez, A., & Song, H. H. (2026). Neuro-symbolic agentic AI: Architectures, integration patterns, applications, open challenges and future research directions. *Computer Science Review*, 60, Article 100902. <https://doi.org/10.1016/j.cosrev.2026.100902>
- Hamin, M., & Edelman, B. (2025, December 2). Cheating on AI agent evaluations. *NIST CAISI Research Blog*. <https://www.nist.gov/blogs/caissi-research-blog/cheating-ai-agent-evaluations>
- Hardt, D. (Ed.). (2012). *The OAuth 2.0 authorization framework* (RFC 6749). Internet Engineering Task Force. <https://doi.org/10.17487/RFC6749>
- Hasan, R., Sion, R., & Winslett, M. (2009). The case of the fake Picasso: Preventing history forgery with secure provenance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (pp. 1–14). USENIX Association. <https://www.usenix.org/conference/fast-09/presentation/case-fake-picasso-preventing-history-forgery-secure-provenance>
- Hassan, A. (2026, September 19). *An attenuated delegation profile for automated agents* (Internet-Draft draft-hamr-oauth-agent-delegation-02) [Work in progress]. Internet

- Engineering Task Force. <https://datatracker.ietf.org/doc/draft-hamr-oauth-agent-delegation/02/>
- Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–580. <https://doi.org/10.1145/363235.363259>
- Hu, V. C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014). *Guide to attribute based access control (ABAC) definition and considerations* (NIST Special Publication 800-162; updated February 2019). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-162>
- Huang, C., Mohan, S., Yang, Z., Tellex, S., & Zhang, L. (2026). Language model as planner and formalizer under constraints. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics* (Vol. 1, pp. 13724–13756). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2026.acl-long.624>
- Huang, J., Hsia, J., Sun, J., Shi, F., Huang, W., & White, I. H. (2026). *Proof-or-Stop: Don't trust the agent, trust the evidence—Loop engineering for verifiable evidence-gated lifecycle control* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2607.14890>
- Hull, R., Damaggio, E., De Masellis, R., Fournier, F., Gupta, M., Heath, F. T., III, Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P. N., & Vaculín, R. (2011). Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems* (pp. 51–62). Association for Computing Machinery. <https://doi.org/10.1145/2002259.2002270>
- IEEE. (2017). *IEEE recommended practice on software reliability* (IEEE Std 1633-2016). <https://doi.org/10.1109/IEEESTD.2017.7827907>

in-toto Project. (2026, March 18). *in-toto attestation framework specification* (Version 1.2.0).

GitHub. <https://github.com/in-toto/attestation/tree/v1.2.0/spec>

International Organization for Standardization. (2022). *Systems and software engineering—*

Systems and software assurance—Part 2: Assurance case (ISO/IEC/IEEE Standard

No. 15026-2:2022). <https://www.iso.org/standard/80625.html>

International Organization for Standardization. (2025). *Systems and software engineering—*

Systems and software assurance—Part 1: Vocabulary and concepts (ISO/IEC/IEEE

Standard No. 15026-1:2025, 2nd ed.). <https://www.iso.org/standard/89808.html>

Jha, S. (2026). RO: Reliable orchestration of LLM programs via deterministic-generative separation. In *2026 56th Annual IEEE International Conference on Dependable Systems and Networks—Supplemental Volume (DSN-S)*. IEEE. <https://doi.org/10.1109/DSN-S70715.2026.00052>

Liao, B., Wang, J., Song, Q., Cao, J., Wang, J., & Deng, B. (2026). *TraceGrant: A contract-governed security framework for the task-effect lifecycle of networked LLM agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2608.21126>

Liu, L., & Yadav, N. (2026, May 19). Introducing STATE-Bench: A benchmark for AI agent memory. *Microsoft Open Source Blog*. <https://opensource.microsoft.com/blog/2026/05/19/introducing-state-bench-a-benchmark-for-ai-agent-memory/>

Lodderstedt, T. (Ed.), Dronia, S., & Scurtescu, M. (2013). *OAuth 2.0 token revocation* (RFC 7009). Internet Engineering Task Force. <https://doi.org/10.17487/RFC7009>

- Lundblade, L., Mandyam, G., O'Donoghue, J., & Wallace, C. (2025). *The entity attestation token (EAT)* (RFC 9711). Internet Engineering Task Force.
<https://doi.org/10.17487/RFC9711>
- Majurski, M. P. (2026, May 5). *Building evaluation probes into agentic AI*. National Institute of Standards and Technology. <https://www.nist.gov/programs-projects/building-evaluation-probes-agentic-ai>
- Maldaner, M. K., Fourney, A., Swearngin, A., Mozannar, H., Bansal, G., Murad, M., Hosn, R., & Amershi, S. (2026). *SentinelBench: A benchmark for long-running monitoring agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2606.05342>
- Meliou, A., Gatterbauer, W., Moore, K. F., & Suciu, D. (2010). The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment*, 4(1), 34–45. <https://doi.org/10.14778/1880172.1880176>
- Microsoft. (2026, April 2). Introducing the Agent Governance Toolkit: Open-source runtime security for AI agents. *Microsoft Open Source Blog*.
<https://opensource.microsoft.com/blog/2026/04/02/introducing-the-agent-governance-toolkit-open-source-runtime-security-for-ai-agents/>
- Moreau, L., & Missier, P. (Eds.). (2013, April 30). *PROV-DM: The PROV data model* [W3C Recommendation]. World Wide Web Consortium. <https://www.w3.org/TR/2013/REC-prov-dm-20130430/>
- Muruaga, X. (2026). *Bounded agents: Delegation security for multi-agent AI systems* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2608.15888>
- Musa, J. D. (1993). Operational profiles in software-reliability engineering. *IEEE Software*, 10(2), 14–32. <https://doi.org/10.1109/52.199724>

- National Aeronautics and Space Administration. (n.d.). SWE-034—Acceptance criteria. In *NASA software engineering handbook*. Retrieved October 2, 2026, from <https://swehb.nasa.gov/spaces/7150/pages/16450634/SWE-034+-+Acceptance+Criteria>
- National Aeronautics and Space Administration. (2016). *NASA systems engineering handbook* (NASA/SP-2016-6105 Rev. 2). https://www.nasa.gov/wp-content/uploads/2018/09/nasa_systems_engineering_handbook_0.pdf
- National Aeronautics and Space Administration. (2022, September 8). *Software assurance and software safety standard* (NASA-STD-8739.8B). <https://standards.nasa.gov/standard/nasa/nasa-std-87398>
- National Institute of Standards and Technology & SEMATECH. (n.d.). 7.2.4.1 Confidence intervals. In *NIST/SEMATECH e-handbook of statistical methods*. Retrieved October 2, 2026, from <https://www.itl.nist.gov/div898/handbook/prc/section2/prc241.htm>
- National Institute of Standards and Technology. (2024, May 28). *NIST launches ARIA, a new program to advance sociotechnical testing and evaluation for AI* [Press release]. <https://www.nist.gov/news-events/news/2024/05/nist-launches-aria-new-program-advance-sociotechnical-testing-and>
- Nguyen, H.-D., & Tran, X.-T. (2026). *Verify-gated completion as admission control in a governed multi-agent runtime: A bounded architecture case study* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2605.17998>
- Object Management Group. (2023). *Structured assurance case metamodel (SACM)* (Version 2.3, formal/23-05-08). <https://www.omg.org/spec/SACM/2.3/PDF>

- Paduraru, C., Bouruc, P.-L., & Stefanescu, A. (2026). *A trace-based assurance framework for agentic AI orchestration: Contracts, testing, and governance* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2603.18096>
- Park, J., & Sandhu, R. (2004). The UCONABC usage control model. *ACM Transactions on Information and System Security*, 7(1), 128–174. <https://doi.org/10.1145/984334.984339>
- Patil, K. (2026). *SentinelAgent: Intent-verified delegation chains for securing federal multi-agent AI systems* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2604.02767>
- Pronesti, M., Belz, A., & Hou, Y. (2026). Beyond outcome verification: Verifiable process reward models for structured reasoning. In *Findings of the Association for Computational Linguistics: ACL 2026* (pp. 32187–32202). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2026.findings-acl.1611>
- Qin, Y., Rhee, M., Teng, D., Zhang, C., & Zi, B. (2026). Neuro-symbolic constraint verification for LLM-driven internet finance transaction execution. *Scientific Reports*. Advance online publication. <https://doi.org/10.1038/s41598-026-65233-w>
- Rabanser, S., Kapoor, S., Kirgis, P., Liu, K., Utpala, S., & Narayanan, A. (2026). Towards a science of AI agent reliability. In *Proceedings of the 43rd International Conference on Machine Learning* (Proceedings of Machine Learning Research, Vol. 306, pp. 102769–102820). PMLR. <https://proceedings.mlr.press/v306/rabanser26a.html>
- Rao, A., Keller, A., Kalra, N., Steed, R., Kwegyir-Aggrey, K., Klyman, K., Staheli, D., & Bergman, A. (2026). *Challenges to the monitoring of deployed AI systems* (NIST AI 800-4). Center for AI Standards and Innovation, National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.AI.800-4>

- Rombach, A., Lauer, C., & Mehdiyev, N. (2026). *Neuro-symbolic agents for regulated process automation: Challenges and research agenda* [Preprint]. arXiv.
<https://doi.org/10.48550/arXiv.2606.13405>
- Saoud, A., Girard, A., & Fribourg, L. (2021). Assume-guarantee contracts for continuous-time systems. *Automatica*, 134, Article 109910.
<https://doi.org/10.1016/j.automatica.2021.109910>
- Shao, J.-J., You, H.-J., Cai, G., Dai, Q., Dong, Z., & Guo, L.-Z. (2025). Breaking the self-evaluation barrier: Reinforced neuro-symbolic planning with large language models. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence* (pp. 6129–6137). <https://doi.org/10.24963/ijcai.2025/682>
- Šljivo, I., Gallina, B., Carlson, J., Hansson, H., & Puri, S. (2017). A method to generate reusable safety case argument-fragments from compositional safety analysis. *Journal of Systems and Software*, 131, 570–590. <https://doi.org/10.1016/j.jss.2016.07.034>
- SLSA Community. (2025, November 24). *SLSA specification* (Version 1.2).
<https://slsa.dev/spec/v1.2/>
- Sporny, M., Thibodeau, T., Jr., Herman, I., Cohen, G., & Jones, M. B. (Eds.). (2025, May 15). *Verifiable credentials data model v2.0* [W3C Recommendation]. World Wide Web Consortium. <https://www.w3.org/TR/2025/REC-vc-data-model-2.0-20250515/>
- Su, Y., Xu, K., Gao, Y., Yang, F., Li, C., Yang, M., & Xu, T. (2026). *Neuro-symbolic verification on instruction following of LLMs* [Preprint]. arXiv.
<https://doi.org/10.48550/arXiv.2601.17789>

- Tabassi, E. (2023). *Artificial intelligence risk management framework (AI RMF 1.0)* (NIST AI 100-1). National Institute of Standards and Technology.
<https://doi.org/10.6028/NIST.AI.100-1>
- Taleb, R., Hallé, S., & Khoury, R. (2023). Uncertainty in runtime verification: A survey. *Computer Science Review*, 50, Article 100594.
<https://doi.org/10.1016/j.cosrev.2023.100594>
- Taylor, B. N., & Kuyatt, C. E. (1994). *Guidelines for evaluating and expressing the uncertainty of NIST measurement results* (NIST Technical Note 1297, 1994 ed.). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.TN.1297>
- Temporal Technologies. (n.d.). *What is Temporal?* Temporal Documentation. Retrieved October 1, 2026, from <https://docs.temporal.io/temporal>
- Tomašev, N., Franklin, M., & Osindero, S. (2026). *Intelligent AI delegation* [Preprint]. arXiv.
<https://doi.org/10.48550/arXiv.2602.11865>
- Turpin, M., Michael, J., Perez, E., & Bowman, S. R. (2023). Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems 36*. <https://doi.org/10.52202/075280-3275>
- UCAN Working Group. (n.d.). *User controlled authorization network (UCAN) specification* (Version 1.0.0). GitHub. Retrieved October 5, 2026, from <https://github.com/ucan-wg/spec>
- van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., & Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1), 5–51.
<https://doi.org/10.1023/A:1022883727209>

- Wei, R., Foster, S., Mei, H., Yan, F., Yang, R., Habli, I., O'Halloran, C., Tudor, N., Kelly, T., & Nemouchi, Y. (2024). ACCESS: Assurance case centric engineering of safety-critical systems. *Journal of Systems and Software*, 213, Article 112034.
<https://doi.org/10.1016/j.jss.2024.112034>
- Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Toh, J. H., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., & Yu, T. (2024). OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems 37* (Datasets and Benchmarks Track).
https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html
- Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. (2025). τ -bench: A benchmark for tool-agent-user interaction in real-world domains. In *Proceedings of the International Conference on Learning Representations*.
https://proceedings.iclr.cc/paper_files/paper/2025/hash/1b126cc38b8638e07bef37e7b2bb72bf-Abstract-Conference.html
- Zhang, T., Huang, C.-K., & Qiu, M. (2026). *Can AI agents deliver verifiable network-wide outcomes across authority boundaries?* [Preprint]. arXiv.
<https://doi.org/10.48550/arXiv.2609.10181>
- Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems 36* (Datasets and Benchmarks Track). <https://doi.org/10.52202/075280-2020>

- Zhou, J., Sheng, Y., Lou, H., Yang, Y., & Fu, J. (2026). FormalJudge: A neuro-symbolic paradigm for agentic oversight. In *Proceedings of the 43rd International Conference on Machine Learning* (Proceedings of Machine Learning Research, Vol. 306, pp. 165723–165752). PMLR. <https://proceedings.mlr.press/v306/zhou26ao.html>
- Zhu, G., & Wang, C. (2026). *Authorization revocation for long-running AI agents: Root-scoped quiescence under delegation and asynchronous execution* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2609.21284>
- Zhuang, S., Wang, S., Liang, E., Cheng, Y., & Stoica, I. (2023). ExoFlow: A universal workflow system for exactly-once DAGs. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation* (pp. 269–286). USENIX Association. <https://www.usenix.org/conference/osdi23/presentation/zhuang>