Creating Reliable Systems with Imperfect Models
Autonomous Development and Shared Execution Contracts

Rick Lopez
MS Applied Artificial Intelligence, University of San Diego
ricklopez@alienship.io
https://dotprompt.ai

Working thought experiment, November 2026

Abstract

Natural-language interfaces allow people to request software without specifying how it should be constructed. The resulting development process must nevertheless establish requirements, resolve consequential assumptions, constrain effects, and determine whether a change is fit to operate. These responsibilities are often distributed across prompts, repository instructions, implementation artifacts, evaluators, and deployment systems. When each component reconstructs their meaning independently, an apparently successful change can satisfy a different contract at every stage.

This paper examines that fragmentation as a source of accidental complexity. It argues for an explicit, versioned representation of engineering obligations that persists across requirements, implementation, evaluation, release, and operation. We introduce DotPrompt as a thought experiment about a shared specification for that representation and its enforcement. The proposal distinguishes expressed intent from inferred assumptions, model judgment from authority, and evidence from acceptance. A small semantic core and scoped implementation profiles illustrate how parts of the proposal might be realized without requiring a universal development platform.

Substantial progress is being made across agent frameworks, execution harnesses, development environments, code intelligence, and verification. This paper asks how the obligations established within those systems can remain consistent as work moves between them. In particular, changes and discovered behavior would create explicit review and evidence obligations, while accepted constraints would govern which effects may proceed. The contribution is a diagnosis, an architectural hypothesis, and a candidate contract model. A synthetic example traces a bounded information-retrieval capability through development and execution; it is an illustration, not an evaluated implementation. The paper identifies failure modes, compares relevant approaches, and proposes ways to test whether shared contracts reduce inconsistency without imposing more complexity than they remove. Its aim is dependable autonomy: less routine supervision because engineering obligations are better represented and enforced, rather than because those obligations have disappeared.

1 Introduction

Large language models are probabilistic by construction. Uncertainty is part of their computational DNA: their judgments emerge from learned statistical relationships, not from a specification that establishes the correctness of each answer. This distinction matters as models take on more of the work of designing, implementing, reviewing, and operating software. A system can produce a convincing implementation without establishing that the implementation is dependable. The ability to generate software does not, by itself, establish the conditions under which that software should be trusted.

We expect software development to become increasingly autonomous. A person will describe a desired outcome, and software agents will translate that intent into an implementation, evaluate the change, and deploy it. For suitable classes of changes, this cycle may take minutes and require no routine human approval. This is the future this paper seeks to make practical. Its central question is what engineer-

ing foundations are needed to make that autonomy dependable.

Software engineering already has a substantial body of knowledge about reliable operation. Explicit interfaces, isolation, observability, controlled releases, and recovery mechanisms allow systems to perform routine work without a person approving every action. Such systems still fail and require maintenance. Their useful autonomy rests on bounded behavior and mechanisms for recognizing and managing failure. Autonomous development must inherit that discipline.

Twenty years ago, Moseley and Marks examined sources of complexity in *Out of the Tar Pit*. Their separation of essential difficulty from complexity introduced by implementation choices provides a useful method for the present problem. We adopt that diagnostic method, not their particular architectural solution. The question is which difficulties belong to the task and which arise because we repeatedly reconstruct its meaning as work moves through an AI-assisted development process. [1]

Shared conventions offer one precedent. Ruby on Rails makes convention over configuration a central principle; the Twelve-Factor methodology supplies a common vocabulary for developing and operating service applications. These approaches reduce repeated local decisions. Formal specifications offer another precedent: ECMAScript defines language semantics against which implementations can be assessed. It does not make all host environments identical or prove that programs are correct. Its relevance is the value of shared meaning across independent implementations. [2, 3, 4]

JavaScript developers can target shared language semantics across browsers such as Chrome and Firefox because their engines implement a common specification. Implementations can differ without requiring developers to redefine the language for each environment. That portability illustrates the opportunity for autonomous development: different tools could construct, review, and release software while preserving a common meaning for requirements, authority, and acceptance. The proposed consistency concerns engineering obligations, not identical model judgments or identical generated code. [4]

Natural language simplifies how we express intent; it does not eliminate the need to establish precisely what the resulting system is permitted and required to do. Removing notation is not the same as removing semantics. Type inference provides a limited analogy: a statically typed language can omit annotations while still deriving and checking types under defined rules. Dynamic typing is different, checking types during execution. Neither mechanism makes the interpretation of natural-language requirements unambiguous.

For a person without software-engineering experience, the distinction can be hidden. A demonstration works, yet an unfamiliar input reveals behavior the person reasonably expected the system to handle. The answer should not be that every user must learn to specify an entire production architecture in a prompt. The development system should identify consequential gaps, apply declared defaults where appropriate, and distinguish assumptions from settled requirements.

The thesis of this paper is that intent needs an explicit, shared representation that carries through requirements, implementation, evaluation, release, and operation. Each stage should work from identified obligations rather than reconstructing them independently. Testing is one place where the consequences of a missing contract become visible; it is not the subject of this proposal. The same ambiguity can shape the wrong feature, an unsuitable design, a misleading acceptance check, or an unsafe deployment decision.

This proposal builds on an active field of engineering work across these areas. The following sections examine selected approaches where they illuminate a particular mechanism or limitation. These examples are illustrative rather than exhaustive; their inclusion does not imply that other systems lack comparable capabilities.

We call the proposed specification DotPrompt. It is a community-oriented thought experiment, not an adopted standard or a completed language. Its intended contribution is a contract boundary across existing tools: what they must preserve, what they may infer, what evidence they produce, and what authority they need to advance work. A shared representation may reduce inconsistency, but it can also create bureaucracy or encode a shared mistake. The remainder of the paper develops the proposal far enough to make those possibilities examinable.

2 Where the complexity enters

2.1 Capability and dependable operation are different objects

A model's success at a task and a software system's reliability are measured at different boundaries. A model can correctly classify an input while the application applies that classification to the wrong account. A generated implementation can meet its visible examples while ignoring a requirement that never reached the agent. A deployment can pass its checks while depending on an environmental assumption those checks did not exercise.

These are not all model errors. Some are specification errors, some are integration errors, and some are failures to preserve relevant evidence. Treating every failure as a need for a stronger model obscures the engineering decisions involved. Treating every failure as a harness defect is equally incomplete: some judgments genuinely improve when the model improves.

A useful analysis must identify where a claim changes status. A user expresses an intention. An agent interprets it. A design commits to some consequences of that interpretation. An evaluator produces observations. A release mechanism decides that those observations justify exposure to users. At each boundary, the system can preserve the reasoning and authority for the transition, or it can silently substitute a new interpretation.

2.2 Essential difficulty is relative to the task

Some requirements are genuinely ambiguous. Evidence may be unavailable. Stakeholders may disagree. External services change while operations are in flight. Whether a design is appropriate can depend on expected usage that no participant knows yet. These difficulties do not disappear because a schema has been introduced.

The distinction between essential and accidental complexity is relative to the problem and its assumptions. Missing information may be unavoidable now but obtainable through a question. A human decision may be required by policy rather than by a permanent limit of automation. The paper therefore does not classify all model uncertainty as irreducible or all manual intervention as waste.

What is avoidable is concealing the distinction. A system that cannot decide should be able to represent an unresolved question. A system using a default should identify that default and its scope. A system that has only partial evidence should not convert it into an unqualified claim of completeness. These are representation and execution choices even when the underlying uncertainty remains.

2.3 Repeated interpretation is an integration cost

Consider the same requirement expressed in a ticket, a planning conversation, a generated test description, and an operational runbook. The copies may be sensible individually and inconsistent collectively. A summary can omit a restriction. An implementation can narrow the meaning of a feature to what was easy to build. An evaluator can reconstruct the intended behavior from the implementation and fail to notice the narrowing.

Agents make this problem particularly visible because their working context is selected, compressed, and replaced. The material available to an agent is not necessarily the authoritative state of the project. A statement in a conversation that a condition was satisfied is different from a retained result identifying the condition, inputs, implementation version, and method of evaluation.

The architectural mistake is allowing each downstream component to decide independently which interpretation is binding. More context may help the component reason, but it does not by itself establish which record has authority. A shared contract should make that distinction explicit without requiring every participant to carry the entire conversation.

2.4 Local success can accumulate into system failure

Software evolves through sequences of changes. Each change can appear reasonable while the sequence produces duplication, incompatible assumptions, or dependencies that make future changes expensive. Local checks are necessary evidence, but their scope must remain visible.

Dex Horthy's account of HumanLayer's experience with unattended development describes this gap: agent-produced changes can satisfy immediate checks while maintainability problems appear later. His discussion of rapidly changing repositories becoming difficult within months is a practitioner observation, not a measured universal threshold. We use it to motivate

longitudinal evaluation, not to establish that every autonomous workflow fails. [5]

The author's team has observed a related, narrower problem: substantial generated test suites created confidence while leaving important paths unexamined. This is an experience report without a controlled dataset. It illustrates how verification activity can be mistaken for adequate assurance. It does not establish that generated tests are generally inferior, nor does it prove that the proposal in this paper would prevent the omissions.

The common concern is a missing account of what the available evidence actually supports. If an interpretation is wrong, code and tests derived from that interpretation can agree. If a design degrades slowly, a check concerned only with immediate behavior may not detect it. Reliable autonomous development requires a shared definition of adequate evidence for particular obligations, not merely a growing collection of passing checks.

3 What established engineering contributes

3.1 Conventions reduce decisions; contracts define obligations

A convention establishes an expected choice when a project has not supplied a reason to diverge. A contract establishes an obligation whose satisfaction has consequences. Confusing them makes a system either rigid or unsafe. A naming convention should not become an authorization rule. A restriction on access to customer data should not become a default an agent can override for convenience.

The proposed representation must distinguish defaults, recommendations, and mandatory requirements. A default has an identified source and an override mechanism. A mandatory requirement has an authority and a defined failure consequence. An agent may recommend changing either, but permission to edit the project does not automatically include permission to change its governing policy.

Existing conventions need not be encoded anew. An implementation can reference a versioned organizational profile containing approved build practices, data-handling rules, and release requirements. The important property is that adoption is explicit and that a profile change has observable consequences for existing definitions, packages, and releases.

3.2 Specification and implementation should remain separable

Once obligations are distinguished from conventions, the next question is where their meaning resides. If each tool defines that meaning privately, replacing a tool can change the contract. A shared specification must therefore describe observable responsibilities independently of the mechanism chosen to satisfy them.

A compiler or adapter may validate a representation and produce checks, configuration, or executable code. An execution environment may enforce the resulting obligations. These roles can coexist in one program, but their claims differ: validation concerns the representation, while enforcement concerns permitted transitions and effects. Neither role establishes that an unstated requirement was discovered.

This separation provides a basis for interchange. Two implementations can use different strategies while being assessed against the same required behavior.

3.3 Authority must be outside the proposal it evaluates

The thing proposing work shouldn't get to rewrite the acceptance rules it's being judged against. This does not require every evaluation to be performed by a different model or a human. It requires the authority for changing criteria to be separate from the authority to submit an implementation against them.

An evaluator may be implemented as deterministic code, a formal checker, an empirical assessment, or a bounded model judgment. Its independence depends on its inputs, privileges, and governing rules. Asking a second model whether the first model did well is not automatically independent verification; both may inherit the same missing assumption.

Likewise, an instruction in a prompt is not an access boundary. The implementation needs a mechanism outside the model's generated proposal to decide whether an effect is permitted. A contract describes that mechanism's obligations; the contract file does not enforce itself.

4 Relevant developments and their limits

The principles above require more than a common document format. Existing work supplies several mechanisms a contract could connect: structured judg-

ments, persistent execution context, independently checked claims, scoped capabilities, and dependency analysis. Each imposes a different requirement on the representation developed in section 5.

4.1 Decision interfaces intended for software

TypeSafe describes Jev as a System One model returning typed decisions and probabilities for use by software. Its documentation explicitly places those judgments inside workflows that also use deterministic checks. This is relevant because it narrows the interface through which a model participates in a program. It is not evidence that input structure alone improves correctness, and calibration across groups does not certify an individual decision. [6]

OpenAI's Decisions API similarly exposes bounded questions through decision-oriented interfaces. Its relevance is the availability of outputs that an application can consume under an explicit contract rather than interpreting arbitrary prose. The interface does not establish whether the application's question was appropriate or whether the chosen action is authorized. [7]

These developments support a division of responsibilities: a model supplies a judgment, while software defines how that judgment may affect the system. They do not remove control flow. They can reduce the need to hand-author some classification rules while leaving the permitted states and transitions explicit. DotPrompt should accommodate such providers without requiring a particular model family or probability scale.

4.2 Harness effectiveness and its limits

A harness supplies the operating environment around a model: tools, context, control flow, persistent state, feedback, and stopping conditions. Its design changes which actions are available and which failures become visible. Anthropic's long-running-agent report describes initialization, incremental work, and persistent records that let successive sessions resume a project. This is an engineering approach to continuity, not evidence that context management alone establishes dependable operation. [8]

LangChain provides a measured example. With GPT-5.2-Codex held fixed, its team reports improving Deep Agents from 52.8% to 66.5% on Terminal Bench 2.0 through harness changes. The changes included explicit verification before completion, environment context, and time-budget guidance, informed by execution traces. The 13.7-percentage-point gain demonstrates that the surrounding system materially affects measured task performance. It is a first-party benchmark result, not a controlled demonstration of long-term production reliability. The report also warns that task-specific improvements can overfit and cause regressions elsewhere. [25]

Cursor's experiments expose a different boundary: coordination. A shared-file locking design produced contention and brittle behavior; a flat group of agents avoided difficult work. Separating planning from implementation improved progress, while adding an integration role introduced another bottleneck. Cursor also reports persistent drift and the need for fresh starts. These observations argue against assuming that more agents or more orchestration necessarily provide stronger assurance. Its large migration example still required careful review despite passing early checks. The report is valuable engineering evidence, but code volume and sustained activity are not measures of correctness or maintainability. Attribution and the subsequent SpaceX affiliation are documented separately. [26, 27]

HumanLayer's *12-Factor Agents* makes ownership of context and control flow explicit engineering responsibilities. These practices provide a starting point for deciding what a harness must preserve when work resumes or moves between components. [9]

Taken together, these reports motivate persistent, inspectable state without prescribing a universal orchestration strategy. The contract must preserve which work was assessed and under what conditions; implementations remain free to optimize scheduling and context management. Cursor's coordination failures also make the cost of that structure a question for evaluation, addressed in sections 10 and 11.

4.3 Neuro-symbolic reasoning and machine-checked proofs

Independent checking has advanced beyond asking another model to assess an answer. Logic-LM translates natural-language problems into symbolic formulations and delegates inference to solvers. Its results concern the logical reasoning tasks evaluated; the solver does not establish that the translation faithfully captures an unstated intention. This separation

between proposing a representation and checking its consequences remains central. [10]

AlphaProof extends this pattern to formal mathematical proof search in Lean. Together with AlphaGeometry 2, it achieved silver-medal-equivalent performance on the 2024 International Mathematical Olympiad problems. The qualification matters: the problems were manually formalized, and some solutions required up to three days. AlphaGeometry 2 combines learned proposals with symbolic reasoning; AlphaProof searches for proofs accepted in a formal environment. These results demonstrate substantial progress in generating checkable reasoning, not general verification of natural-language software requests. [28]

Harmonic's Aristotle combines informal reasoning, Lean proof search, and a geometry solver; its research paper reports gold-medal-equivalent performance on the 2025 IMO problems. This strengthens the precedent for models producing results whose acceptance depends on an independently defined checker. The benchmark is mathematical theorem proving, so it should not be presented as a measured software reliability gain. [29]

There is also a direct software connection. Aristotle's documentation describes verification of C programs, returning proved properties, caller obligations, and assumptions about external libraries. Its default memory-safety claim is conditional on callers meeting stated preconditions. That is a concrete example of why evidence needs scope: a proof can be strong while depending on obligations outside the function being proved. These are documented product capabilities, not independently reproduced results in this paper. [30]

The resulting requirement is to preserve the proposition and assumptions with the checking result. A proof's caller preconditions become obligations that another component may need to discharge. Section 5.3 defines the evidence binding needed to retain this meaning; section 8.6 applies it to a changed caller. Proof generation and operational verification can then cooperate without treating their claims as interchangeable.

4.4 Incidents expose enforcement assumptions

OpenAI's account of the July 2026 Hugging Face incident describes internal evaluations under reduced safeguards. It directs attention to the conditions under which controls operate and the capabilities available to an agent. It cannot establish that the proposed specification would have prevented the incident. [11]

The design consequence is to identify the enforcement boundary and its assumptions as part of an implementation's claim. A declared restriction is useful only where a component actually mediates the relevant action. Section 6.3 makes that responsibility explicit.

4.5 Code graphs and verification are already advancing

Significant progress is already being made in the mechanisms this proposal would depend on. CodeGraph describes a local code-intelligence system that derives symbols and relationships through Tree-sitter parsing, exposes them to agents, and supports change-impact analysis. Graphify describes persistent relationships across code and supporting material, source-linked answers, review, and formal-verification features with coverage dependent on workflow and language. These are project descriptions, not independently reproduced evaluations. [16, 17]

These tools supply relationships that can help locate the consequences of a change. The remaining representation question is how a structural finding becomes a claim about an accepted obligation. That step needs its own supporting records and authority; a call edge alone cannot supply them. This is the final requirement for the core developed next. Section 9 compares the proposed handoff semantics with existing approaches.

5 The proposal: preserve meaning across the lifecycle

5.1 The smallest useful contract

The diagnosis identifies three ways meaning is lost: interpretations become binding without a visible decision, evidence is used beyond the conditions it examined, and obligations disappear at a tool boundary. The preceding engineering principles suggest a minimum representation that makes each failure observable.

Four relationships are required. An obligation has an identified authority. An implementation claims to satisfy an identified obligation. Evidence supports a stated property of a particular implementation under

specified conditions. A decision applies a governing rule to that evidence to permit or reject a transition.

Each relationship answers a question left open by the others. Without authority, a recorded requirement may be only a suggestion. Without an implementation binding, a result may concern different code. Without the property and conditions, a passing check has no defined scope. Without the decision rule, evidence does not explain why an action was permitted.

Three rules follow. A proposal cannot acquire authority merely by being generated or well formed. Acceptance cannot travel to a different implementation or context without an explicit applicability decision. A receiving component must expose an unsupported mandatory obligation rather than silently weaken it. Sections 6 and 7 turn these rules into admission and compatibility behavior.

The graph is the set of these relationships. It can reference records in existing tools without requiring a universal database or a fixed workflow. Work may branch, proceed concurrently, and revisit decisions, provided consequential decisions resolve to authoritative records. Sections 5.2–5.4 define those records; sections 5.5–5.8 explain how new observations cause their relationships to be reconsidered.

theight

Figure 1: Contract relationships. Acceptance supplies authority; evidence supplies a scoped claim. Neither a definition nor an implementation grants itself admission.

5.2 Intent, assumptions, and commitments

The authority relationship begins with interpretation. Expressed intent records the request; an interpretation proposes what satisfying it entails; acceptance makes an obligation a commitment. To make that transition inspectable, the representation needs both provenance and resolution status.

An interpretation records its origin separately from its resolution status. It may originate in an explicit request, an inference, or an adopted default; it may remain unresolved regardless of that origin. Neither provenance nor acceptance establishes truth. A user-supplied statement may be inconsistent, an inference appropriate, or a default unsuitable for the environment. Keeping these dimensions separate prevents "the user said it" or "the model inferred it" from becoming a substitute for resolving a consequential question.

Authority to accept a commitment may come from a project owner, an organizational policy, or a previously authorized automated rule. Requiring an explicit authority does not require a human click on every change. It requires the implementation to identify the rule that made the change authoritative and the scope within which that rule applies.

Consequential unresolved questions should block the transitions they affect. They need not block unrelated work. An unknown display preference can remain open while a data-isolation restriction is settled. An unknown authorization rule should not be filled in merely because code generation can proceed without an answer.

5.3 Binding evidence to an implementation

Once an obligation is accepted, evaluation must identify what was examined. The implementation record names an immutable version or digest and the commitments it claims to realize. A check identifies the property, method, inputs, and limits. An observation records its result. The decision record then identifies the policy under which that result supports a particular transition.

Evidence applicability depends on explicit conditions: the implementation revision, relevant dependencies, environment, schema, or observation interval. A field-presence check supports an output-structure claim; it does not acquire authority over ordering or account selection by being attached to the same package. A later change requires reconsideration only of the conditions on which the evidence depends.

Formal proof evidence from section 4.3 makes the need for those conditions especially clear. Its record identifies the proposition, implementation or formal model, assumptions, checker and relevant versions, and checking result. A profile states which trusted axioms or components it permits and how it rejects incomplete proofs. A timeout leaves a proof attempt unresolved; it is not a counterexample.

The representation must retain conditional results without prematurely treating them as usable acceptance evidence. A checked proof may remain valid while a changed caller no longer satisfies its premises. Sections 5.5 and 6.4 develop how changes trigger that applicability decision.

5.4 A shared vocabulary

These records give separate identities to the relationships above. An accepted definition has binding obligations accepted under an identified authority; unresolved questions may remain, blocking only the transitions they affect. A package binds an implementation to that definition, while a release records its admission to a particular environment.

Why the specification avoids "artifact" as a core record name

"Artifact" can refer to a generated document, a build output, an executable bundle, or a retained evaluation result. The umbrella term is useful within a platform that defines its meaning, but insufficient at a boundary where different tools assign it different roles. Calling something an artifact does not tell a receiving component whether it describes intended behavior, implements that behavior, records evidence, or has been admitted for execution.

Martin Fowler describes *semantic diffusion* as the weakening of a term's definition as it spreads through a community. His response emphasizes restating precise meanings rather than assuming that a replacement label will solve the problem. That is a useful caution here: this paper does not establish that "artifact" once had a single meaning and subsequently lost it. The immediate problem is that its breadth leaves distinctions essential to this specification unstated. [24]

We therefore use **definition**, **implementation**, **execution package**, and **release** where those roles affect interpretation or authority. These names require explicit semantics and conformance rules; they are not immune to the same ambiguity. "Artifact" remains appropriate when discussing existing tooling. An artifact repository can store and distribute an execution package if it preserves the profile's identities, version bindings, and admission rules. No new binary format or renaming of existing platforms is required.

The vocabulary expands the core relationships in section 5.1 for different profiles. An impact assertion, introduced below, is a claim about a relationship between a report and an obligation; its acceptance is recorded as a decision. It does not require another universal top-level record kind. A requirements-only tool may produce candidate commitments without executing effects; a runtime-only tool may consume accepted commitments without interpreting natural language. Each profile declares the records and relationships it supports.

5.5 Execution discovery and obligation mapping

Rails instrumentation makes named events available to subscribers, separating observation from response. React Effects provide a complementary precedent: declared dependencies determine when synchronization must be reconsidered. These are established patterns for making activity visible and responding to changes. Neither discovers execution paths; analysis and instrumentation must supply that information. [18, 19]

The evidence bindings above describe an assessment at a point in time. Keeping them useful as software changes requires a way to identify what deserves reconsideration. The proposed execution-discovery interface supplies that trigger.

A discovery report identifies its subject, baseline and candidate revisions, method, and observation or analysis scope. Static analysis reports potentially affected relationships; runtime instrumentation reports behavior observed under particular conditions. An execution path may be a trace sequence or an affected dependency slice, as defined by the profile. A call edge is not a universal identifier for either.

Mapping a report to an obligation requires a separate impact assertion. That assertion names the report, the versioned obligation, the claimed relationship, its supporting records, its producer, and its resolution status. A profile must say who may accept the assertion and what follows while it remains unresolved. Attaching an obligation identifier to a report is insufficient.

An impact assertion can rest on several kinds of support. A profile may adopt an explicit binding between an operation and its obligations. A versioned dependency analysis may connect a changed helper to that operation. A runtime trace may show the helper executing within it. A model may infer a possible relationship that those mechanisms cannot resolve. These sources have different scopes. An accepted explicit binding establishes which obligation governs an operation; it does not prove that every dependency has been found.

For a minimal change-review profile, a report affecting a bound operation opens an assessment for its applicable obligations. An accepted dependency mapping can narrow the assessment. A suspected consequential mapping remains unresolved until the declared resolution rule is satisfied. An unsupported

Table 1: Shared contract vocabulary

Record	Meaning	Required distinction
Intent	The request and its source	A request is not a complete contract.
Definition	A structured, versioned interpretation of intended behavior and its conditions	Well-formed does not mean authoritative, complete, or correct.
Commitment	An accepted obligation with an authority and version	An inferred candidate is not automatically binding.
Assumption	A premise used to interpret or implement the task	Its origin and validity conditions remain visible.
Implementation	Versioned code or executable logic intended to satisfy a definition	Existence does not establish satisfaction.
Execution package	A versioned unit binding an implementation to its definition and execution requirements	Declared capabilities are requests, not grants.
Release	A package admitted to an identified environment and scope	Admission is a scoped decision, not indefinite correctness.
Evidence	An observation produced by an identified method	A reported result is not unlimited assurance.
Decision	A scoped acceptance, rejection, or deferral under a policy	Acceptance for testing is not acceptance for deployment.
Effect	An attempted action and its observed outcome	Requested, dispatched, and confirmed are different states.
Capability	An enforceable class of permitted actions	Describing a tool is not granting permission to use it.

analysis result cannot certify that nothing is affected. The profile must define the boundary within which it claims coverage and its fallback outside that boundary.

Assessment has four useful outcomes: retain evidence under a justified applicability rule; require replacement or additional evidence; propose a repair for a demonstrated violation; or raise a contract question where expected behavior is unsettled. Each outcome records the versions and rule used. The discovery producer can propose an outcome, but only the authorized evaluator can accept it. Resolving an impact assertion does not itself admit the package.

These outcomes establish the input to continuous contract evaluation: an identified obligation, a supported or unresolved impact claim, and an assessment that remains to be performed. Section 8.6 exercises both an explicit dependency and an unresolved dynamic dependency through this interface.

5.6 Potential application: continuous contract evaluation

Continuous contract evaluation assesses successive change reports and runtime observations against accepted obligations. "Continuous" describes responsiveness to new information, not exhaustive monitoring or instantaneous judgment.

A dedicated QA agent could consume an impact assertion, retrieve the governing obligation and applicable evidence, and propose a focused check. The observation supplies the scenario; the contract supplies the expected behavior. If a retrieval trace contains a report from another account, the accepted account-scope requirement explains why that outcome is wrong. The agent can build a controlled fixture that reproduces the condition without treating the observed output as the correct answer. Access and data-handling rules still govern use of operational data.

The evaluator first asks whether an existing check already covers the case and remains applicable. If it does, another generated test may add no value. If expected behavior is unresolved, the response is a contract question. If a new test is warranted, evaluation should establish its requirement link, reproducibility, and sensitivity to the relevant failure. A controlled faulty implementation can help test that sensitivity.

The QA agent returns a candidate check and its requirement link, or a recorded reason that another response is needed. That output can be consumed by a separate evaluator or release component under the authority rule in section 3.3. It is not a release decision.

The hypothesis is that this process directs verification toward consequential changes as implementation evolves. It complements unit, regression, integration, and end-to-end testing. Section 11 specifies how to

assess its effect on missed obligations, brittle or duplicate tests, rework, and cost.

theight

Figure 2: Continuous contract evaluation. Discovery initiates assessment. Evidence reuse, further checks, clarification, and repair are different outcomes; none bypasses admission.

5.7 Transporting reports without confusing notification with evaluation

A discovery interface need not introduce a new messaging protocol. In the November 2025 MCP specification, resource subscriptions are an optional capability: a compatible client can subscribe to changes to an identified resource, receive an update notification, and retrieve its current contents. Resource-list notifications are a separate capability. This is a concrete transport precedent for a discovery report, not a requirement that every MCP implementation support the pattern. An adapter must declare the protocol version and capabilities it actually supports. [20]

A DotPrompt implementation could expose a report as a resource whose contents identify the implementation revision, comparison baseline, discovery method, affected obligations, and unresolved mappings. Notification would prompt an evaluator to retrieve that report and its governing definition. The report's identity and version must survive retrieval: fetching a mutable "latest" resource must not silently substitute a newer revision for the one under review. A resource-change notification alone is neither a durable event history nor proof that every relevant change has been evaluated. Retention, recovery, access control, and reconciliation remain explicit implementation responsibilities.

CodeRabbit documents using MCP servers to supply context for code review. A future integration could let review agents such as CodeRabbit receive compatible notifications, retrieve a changed report together with its definition, and assess the affected obligations. The documented context integration is a precedent; notification-triggered contract evaluation is the prospective example. No claim is made here about current support for that complete workflow. [21]

For such an integration, the testable transport obligation is identity preservation: a notification for report R must resolve to R and its referenced versions, even if newer reports already exist. Scheduling its consumers introduces the concurrency questions below.

5.8 Concurrent, event-driven quality evaluation

Evaluation can proceed alongside implementation. Repository events identify changed code and dependencies; runtime observations identify behavior exercised under particular conditions. A local trace cannot reveal an unrelated commit, and a commit does not establish that an execution occurred. The profile preserves the source and scope of each signal.

Suppose a developer adds pagination against an account-scoped helper while another contributor changes that helper to return unfiltered results. An evaluator can flag the affected isolation obligation before a full end-to-end run. The unfinished candidate may already contain a compensating filter, so the finding is a reason to investigate rather than a verdict.

Each finding names a stable snapshot, definition and policy versions, the affected obligation, supporting evidence, and proposed response. Uncommitted work also needs a stable snapshot identity. Superseded findings remain historical until their applicability is rechecked. An evaluator may advise the development process; blocking requires a separately authorized enforcement boundary.

Concurrent describes overlapping work, asynchronous describes independent notification and completion, and event-driven describes what initiates evaluation. Subagents are one implementation option. IDE diagnostics, watch-mode tests, and parallel CI already provide earlier feedback. DotPrompt's proposed addition is the common obligation and evidence binding across them.

The timing hypothesis is narrower than the verification hypothesis in section 5.6: starting assessment earlier may reduce late repairs and reruns without changing required release checks or the execution time of an individual test.

Noise and workload need explicit control. Evaluators can group reports by obligation and revision, coalesce duplicates, and reuse applicable evidence. Optional telemetry may be sampled, but a processing budget cannot erase a mandatory assessment. Unresolved consequential mappings follow the profile's fallback rather than being discarded as irrelevant.

Queue limits, deadlines, cancellation of superseded work, and visible backlog bound concurrent evaluation. A saturated evaluator reports pending work. Section 6 explains why pending work cannot silently become admission.

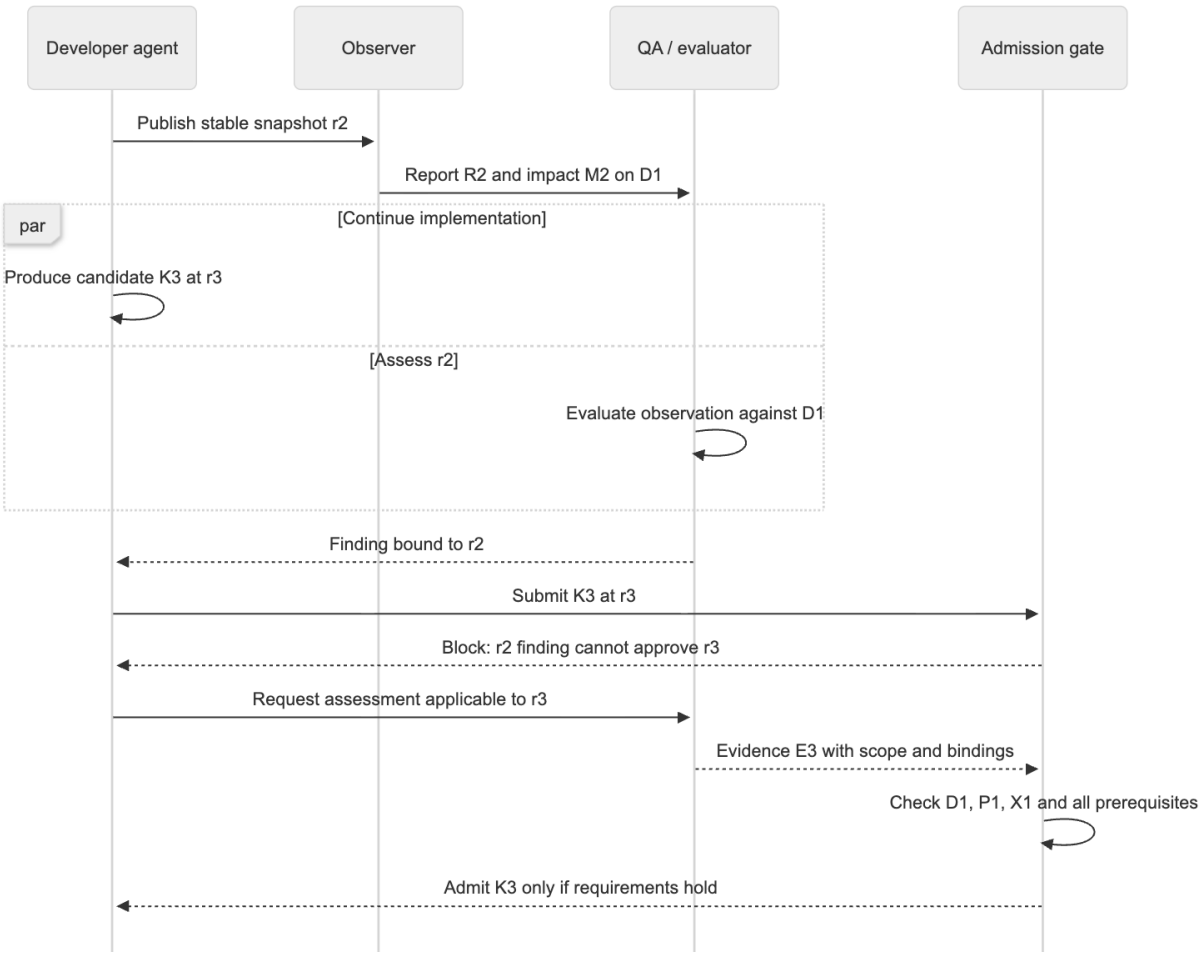


Figure 3: Concurrent evaluation. A finding for r2 remains about r2 after development advances. The gate requires evidence applicable to r3 before admitting K3.

6 From representation to enforceable behavior

The representation can now identify obligations, findings, and applicable evidence. Execution requires one further step: a component must use those records to permit or reject a transition. This section defines that decision boundary and what happens when conditions change or an action's outcome becomes uncertain.

6.1 Candidate lifecycle rules

An implementation would interpret the contract through explicit transitions. A candidate requirement can be proposed, accepted, superseded, or left unresolved. An implementation can be constructed and evaluated. An execution package binds that implementation to a definition and declared execution requirements; admission creates a release scoped to an environment. That release can later be superseded or withdrawn without changing the historical package or its evidence. A run can be ready, executing, awaiting a permitted response, reconciling an uncertain effect, complete, failed, or cancelled.

These are distinct lifecycles. Finishing a development task does not imply that the deployed operation is healthy indefinitely. Cancelling a run does not erase an effect already dispatched. Superseding a requirement does not retroactively change the meaning under which an old release was evaluated.

Every transition that changes authority or declares success must identify its prerequisites and policy. A model may propose the transition, but the enforcement component determines whether it is permitted. An unsupported prerequisite must yield a declared incompatibility or a blocked transition, never an implicit assumption that the requirement was satisfied.

6.2 A scoped admission rule

Let `C` be a contract version, `K` an execution-package version, `E` the relevant evidence, `P` the governing policy, and `X` the current execution context. An illustrative admission rule is:

```
admit(K, C, E, P, X) =
    required_capabilities_supported(C
     ↪ ,
     ↪ X)
    and authority_valid(P, X)
    and blocking_questions_resolved(C
     ↪ ,
     ↪ P)
    and package_bound_to(K, C)
    and evidence_bound_to(E, K, C, X)
    and required_obligations_discharg
     ↪ ed(C, E,
     ↪ P)
```

This is a decomposition of responsibilities, not a definition of universal readiness. The last predicate is only meaningful if a profile defines its obligations and admissible evidence. A profile might require a checked invariant, an empirical threshold, a signed approval, or a combination. The predicate must not reduce to asking the proposing model whether it feels confident.

Admission is also scoped. Admission to a sandbox is not admission to production. Admission for one caller, destination, or data class does not grant broader authority. The context must include the attributes on which the policy actually depends, rather than an underspecified label such as "trusted environment."

For example, a package evaluated with read-only credentials cannot reuse that admission after its execution identity gains write access unless the governing policy establishes that the changed context preserves the required boundary. Conversely, changing a documentation comment need not invalidate an account-isolation check if the profile can establish that the relevant implementation and environment are unchanged. The rule is explicit applicability, not universal invalidation.

The transition itself must bind the decision to the candidate actually admitted. If the package, policy, or relevant environment changes between evaluation and admission, the executor must reject the stale decision or reevaluate the changed conditions. Version comparison or an equivalent atomic admission mechanism prevents a valid assessment of one candidate from being applied to another. Long-running operations may also require authority checks at effect dispatch; admission at the start cannot authorize a capability that has since been revoked.

6.3 What an executor can promise

Admission establishes eligibility to run; effect dispatch determines whether a particular action may proceed. A capability therefore names an action and its scope, including destination, identity, resource class, and relevant limits. Permission to read one account does not permit reading another, and permission to propose a repair does not permit publishing it.

An executor claiming preventive enforcement must mediate the action before dispatch. It must protect its governing state and prevent the implementation from bypassing that boundary with broader credentials. A monitor that detects a forbidden action afterward can support containment, but cannot claim to have prevented it.

Within this boundary, conformance can require rejection of out-of-scope attempts, retention of the run's governing version, and refusal to declare completion while mandatory checks remain unresolved. These observable behaviors provide the enforcement component of the admission claim. They do not establish the adequacy of the original requirement or the truth of an external source.

6.4 Change and evidence invalidation

A change creates a candidate with its own bindings. Evidence from a previous candidate may be reused only where its validity conditions still hold or an authorized applicability decision establishes that they do. The impact assertions in section 5.5 supply the reasons to reconsider particular obligations; they do not decide applicability by graph reachability alone.

A previously admitted release may continue operating while its replacement is evaluated if policy permits. Revoked access or a known isolation failure may instead require withdrawal. The policy identifies the response, so creation of a candidate does not automatically imply either shutdown or permission to continue.

Provider changes follow the same rule. An unchanged output schema establishes structural compatibility, while changed model judgments may require renewed task-specific evaluation. The adapter must

identify which properties its compatibility claim covers.

6.5 Admission does not depend on notification delivery

Notifications schedule work; admission reads the required evidence state. A delayed, duplicated, or lost event must not turn an unassessed candidate into an accepted release. The admission component checks the candidate's bindings and the status of its required assessments directly.

The profile must define how it establishes that required discovery and evaluation have occurred for the candidate. This might involve a revision-bound analysis manifest and recorded outcomes for each mandatory obligation. An unavailable analyzer or unresolved consequential mapping follows the declared fallback: broader evaluation, deferral, or rejection. A subscriber's silence cannot discharge the obligation.

This separates transport reliability from acceptance semantics. Retrying a notification may restore processing, but only applicable evidence and an authorized decision can restore eligibility for admission.

6.6 Repair preserves both constraints and required outcomes

The repair outcome from section 5.5 creates a new candidate. Its acceptance must consider affirmative requirements as well as prohibitions: eliminating an operation can eliminate an unauthorized effect while also eliminating required functionality.

If the repair changes only the implementation, it follows the existing admission policy. If it changes expected behavior, it also requires a separately authorized contract revision and reconsideration of affected evidence. This is the operational consequence of the authority rule in section 3.3.

When discovery occurs after an external action, a code repair alone cannot settle the action's outcome. The run must follow the recovery rules below.

6.7 Recovery is part of meaning

For an external action, loss of the response does not establish failure of the action. An executor that retries blindly can duplicate an effect. Where the destination supports an idempotency key or conditional operation, the executor can use it under declared semantics. Where it does not, the run may need reconciliation before another attempt.

A proposed rule is that an unresolved consequential effect prevents a success declaration and blocks unsafe repetition of that effect. Cancellation stops new dispatch when possible but cannot retroactively undo a completed external action. Recovery may require observation, compensation, or escalation; those are different mechanisms and should be named separately.

Human participation is one possible recovery path. A response must be bound to the relevant run, contract version, question, and validity conditions. A late response should not authorize a cancelled run, and a stale response should not settle a materially changed question. The same rules apply when an authorized automated service supplies the response.

7 One specification, multiple implementation profiles

The enforcement rules require capabilities that no single tool need supply. A requirements interpreter may accept commitments without executing effects; a browser executor may enforce an operation without assessing architectural maintainability. A profile specifies the supported records, transitions, enforcement responsibilities, environmental prerequisites, and emitted evidence for one such boundary.

Composition then becomes an obligation-assignment problem. For every mandatory obligation, the integration must identify a supporting component and the evidence on which the receiving decision relies. If no component can discharge an obligation, parsing the exchanged document cannot make the integration compatible.

The same reasoning determines extension behavior. Unknown mandatory capabilities require rejection or explicit deferral; optional annotations may be ignored only when that cannot weaken mandatory semantics. Version negotiation and migration must preserve those distinctions, so a compatibility claim identifies a profile version and observable rejection behavior.

Consider two tools exchanging a report-retrieval package. The producer declares a prohibition on cross-account access; the receiver can validate output structure but cannot enforce isolation. A third component may supply an enforceable scoped session, but composition remains incomplete until the profile as-

Table 2: Candidate implementation profiles

Candidate profile	What it could establish	What it would leave unresolved
Requirements interpretation	Recorded intent, proposed assumptions, accepted commitments	Whether all unstated needs were discovered
Implementation and package construction	Versioned implementations bound to definitions and declared commitments	Whether those commitments are adequate
Release admission	Required evidence and authority for a specific environment	Indefinite correctness after deployment
Bounded operation execution	Capability checks, tracked effects, required result checks	General correctness of arbitrary model judgments
Evidence interchange	Provenance, scope, version bindings, supported evidence types	The truth of every external observation
Change and path discovery	Versioned analysis reports, observed paths, affected relationships, declared analysis limits	Exhaustive path discovery or authority to approve changed behavior

signs that component the boundary and identifies the evidence for it. This tests whether responsibility survives the handoff.

Such assignments leave implementation strategy open. A framework, compiler, runtime, or policy service can participate if it preserves the declared behavior. The next example examines whether an existing instruction-based package can satisfy those responsibilities.

7.1 Skills, scripts, and compiled modules

A skill bundle provides a useful packaging test. Its SKILL.md instructions could remain in their existing format while a manifest or adapter binds the bundle's version to a definition, dependencies, requested capabilities, and evidence required by the profile.

The executor must additionally identify the model and harness assumptions on which the instruction-driven implementation depends. A script or compiled module may have different dependencies, but the receiver still needs the bindings defined in section 5.4. Changing the packaging format cannot remove an obligation.

The package declares requirements; the executor enforces them. A conforming integration therefore needs both interpretable declarations and a component responsible for their enforcement. This shows how existing packaging could participate without treating a file extension or manifest as proof of conformance. No existing skill format is claimed to implement the proposed profile.

Sample: an instruction-based execution package

The following SKILL.md and package manifest illustrate the packaging option above using K3, r3, and D1 from the worked example. Identifiers are readable stand-ins for immutable references; a real profile must define how each resolves and is verified. These declarations request capabilities. They do not grant them.

```
---
name: finalized-report-reader
description: Retrieve finalized
 ↪  reports within one account.
---
# Finalized report reader

Resolve definition D1 and the run's
 ↪  granted capabilities before
 ↪  acting.
Confirm the designated account before
 ↪  reading each page.
Collect finalized reports in
 ↪  publication order, preserving
 ↪  source links.
If account identity or finalization
 ↪  is uncertain, report the
 ↪  unresolved
condition. Do not infer permission to
 ↪  edit provider records.
Return observations and provenance;
 ↪  do not declare release acceptance.
```

```
package_id: K3
implementation: r3
definition: D1
entrypoint: SKILL.md
profile:
 ↪  bounded-read-operation/proposed-1
requires:
```

```
  - scoped_browser_session
  - account_identity_check
  - result_provenance
requested_effects:
 ↪  [read_report_listing]
checks: [C-account, C-selection]
execution_assumptions:
  model: reader-model@1
  harness: reader-harness@1
admission_policy: P1
```

The executor must bind the recorded observations to the actual model, harness, definition, and implementation used. Changing those assumptions requires the applicability decision described in section 6.4.

8 A bounded worked example

8.1 The request and the missing semantics

Consider a synthetic request: "Give my assistant the latest service reports from our provider's website." The example is deliberately modest. It illustrates a reusable information capability, not a complete autonomous software-development platform. There is no live provider integration or measured result behind it.

The natural-language request leaves several questions open. Which account is in scope? Does "latest" mean the newest report or every report since the previous run? Must drafts be excluded? What evidence should accompany the response? Does the requester authorize downloading documents, or only reading the listing? These questions concern the meaning of the capability, not the particular browser library used to implement it.

For the example, suppose the requester accepts a contract that returns up to ten finalized reports ordered by the provider's publication timestamp, from a single designated account. Each result contains its source identifier, title, publication timestamp, source link, and retrieval time. Drafts are excluded. The capability must not publish, edit, or delete provider records. Authentication may require an account-owner handoff.

The accepted interpretation is now narrower and inspectable. Another user could choose a different interpretation. The shared specification supplies the form and transition rules; it does not decide that ten reports is universally correct.

8.2 An illustrative operation declaration

Sample file: finalized-reports.prompt. The following YAML is illustrative candidate syntax, not an executable format or a claim of Genkit Dotprompt compatibility. D1 identifies the accepted definition used throughout the samples; its accepted status must be established by an authority record outside this file. Names identify obligations whose checks and enforcement must be supplied by a profile.

```
definition_id: D1
operation: finalized_service_reports
contract_version: 1
profile:
 ↪  bounded-read-operation/proposed-1

intent:
  source: requester_statement
  text: "Get the latest service
   ↪  reports from our provider."

commitments:
  account_scope:
    obligation_id: O-account
    value: designated_account
    origin: requester_confirmed
  selection:
    obligation_id: O-selection
    status: finalized
    order: publication_timestamp_desc
     ↪  ending
    maximum_count: 10
    origin: accepted_interpretation
  excluded_effects:
    - publish_record
    - edit_record
    - delete_record

output:
  required_fields:
    - source_id
    - title
    - publication_timestamp
    - source_link
    - retrieved_at
  completeness:
    modes: [bounded_latest_set,
     ↪  partial]
    partial_requires:
     ↪  missing_scope_and_reason

execution:
  required_capabilities:
    - scoped_browser_session
```

```
    - durable_run_state
    - account_identity_check
    - result_provenance
  authentication:
    challenge:
     ↪ await_authorized_account_owner
    reuse:
     ↪ permitted_while_session_valid
  unexpected_page: stop_and_report
  repair: create_new_candidate

acceptance:
  requires:
    - account_matches_contract
    - records_are_finalized
    - required_fields_present
    - source_links_match_collected_re
     ↪ cords
    - ordering_matches_contract
    - completeness_mode_supported_by_
     ↪ evidence
```

A profile would have to define which destination controls enforce the excluded effects. Some portals combine read and write actions behind the same origin and request methods. A network allowlist alone may not establish read-only behavior. If the adapter cannot enforce the required restriction using available identity scopes, endpoint controls, or a sufficiently constrained action interface, it must report that limitation rather than label the operation conforming.

8.3 From definition to admitted release

For this retrieval capability, the builder produces an implementation and packages it with the accepted definition, dependency identities, required capabilities, and profile. The construction strategy may use recorded interaction, conventional code, or model-assisted exploration.

Evaluation must distinguish the operation's claims. Successful authentication does not identify the intended account. Ten extracted rows do not establish that they are the latest finalized reports. If the available source behavior cannot establish ordering or the bounded set, the candidate must support the declared partial outcome rather than claim completeness.

The admission rule in section 6.2 binds this package and its evidence to the designated account, environment, and policy. An unresolved required claim leaves the package a candidate. An accepted package becomes a scoped release whose result concerns faithful retrieval under the stated conditions, not the factual truth of the provider's records.

8.4 Run-time behavior

When an assistant requests the operation, the executor checks the caller's authority and starts a run bound to the accepted version. It opens the scoped session and checks the account. If authentication has expired, it enters an awaiting-response state and presents a controlled handoff. After authentication, the executor verifies the expected account rather than assuming that successful login proves account identity.

An authenticated session may be reused while valid and permitted by policy; expiration returns the run to its declared authentication transition.

The executor collects records, applies the checks, and returns the result with its evidence references and completeness mode. It must not report the full bounded set if pagination or ordering could not be established. A caller may choose to accept a partial result for its own purpose, but that choice must not alter what the operation claims to have established.

8.5 External change and interrupted observation

Suppose the provider adds a draft row or changes its listing structure. A selector may still locate elements even though the source assumptions behind selection evidence have changed. The applicability rule in section 6.4 therefore matters even when the package itself is unchanged.

Interruption creates a different problem. Retrieval is read-only, but source records may change between attempts. A resumed run must expose its observation window or re-collect consistently under the profile's rules; otherwise it may combine records that never formed the claimed bounded set.

These cases concern changes outside the implementation. The next case follows a deliberate code change through discovery, mapping, repair, and admission.

8.6 Following one change from discovery to admission

Give the accepted account-scope commitment the identifier O-account, in definition D1. The admitted package K1 uses implementation r1. Evidence E1 supports its account-isolation claim under execution con-

text X1. These identifiers are explanatory labels, not a proposed serialization format.

The profile includes an accepted binding from the retrieval operation to O-account. It requires each result to belong to the designated account and requires applicable isolation evidence before admission. Evidence E1 records the r1 dependency snapshot; its validity rule does not permit reuse after retrieval dependencies change without reevaluation.

A worker adds pagination in revision r2 and creates candidate K2. Analyzer A reports a new dependency from retrieval to helper paginate. Report R2 names r1 and r2, the analyzer version, and the source locations supporting the edge. Instrumentation separately records a second-page request under r2. The static report identifies a potential dependency; the trace shows one observed execution.

The mapping component proposes impact assertion M2: R2 may affect O-account. Its basis is the analyzed dependency together with the accepted operation-to-obligation binding. The profile authorizes an evaluator to accept this mapping for scheduling an isolation assessment. Acceptance of M2 establishes the need to reconsider E1; it does not establish that r2 violates isolation.

Now suppose paginate selects a provider adapter through a runtime registry that A cannot fully resolve. A trace using adapter A1 does not settle behavior under adapter A2. The mapping must retain this unresolved boundary. For this example, the profile requires checks for every declared production adapter, and prohibits admitting an undeclared adapter. It permits operation-level evaluation when a complete static dependency map is unavailable. If the candidate's adapter set cannot be identified, admission remains blocked. This is an explicit coverage choice with a visible limit, not a claim to enumerate every possible path.

The QA agent constructs a fixture in which a second-page response contains a report from another account. O-account supplies the expected result: the implementation must exclude or reject the mismatched record under the accepted operation behavior. Existing checks cover only first-page retrieval, so the evaluator requires additional evidence. K2 returns the foreign record, demonstrating a violation under the fixture. A repair adds account validation to every collected page, producing revision r3 and candidate K3.

The repair must satisfy both the isolation constraint and the affirmative retrieval obligations. Its evaluation checks that mismatched records cannot escape and that permitted finalized reports remain retrievable under the declared ordering and completeness rules. A proposed test is also checked against the faulty r2 implementation so that a passing r3 result is accompanied by evidence that the test detects this failure. These observations have a stated test scope; they do not prove isolation for arbitrary portals.

Admission now considers K3, D1, the applicable policy, X1, and the evidence bound to r3. A late success result from r2 cannot approve K3. Required evidence for the other obligations remains applicable or is renewed according to its own validity rules. If the bindings and required assessments hold, the authorized gate admits K3. At no stage does completion of the QA agent's task substitute for the admission decision.

A proof can participate in the same chain. Suppose a checker establishes that the collection helper preserves the account tag when every input record has the designated account tag. That conditional result belongs to its helper revision and assumptions. A changed adapter that supplies unvalidated records breaks the justification for applying it, even if the helper's proof still checks. The caller obligation must be discharged before the proof can support this release. This is how stronger verification enters the contract without acquiring an unlimited meaning.

The completed chain now identifies why reassessment began, how uncertainty was resolved, which candidate failed, what the repair preserved, and which evidence justified admission. The comparison in section 9 asks what preserving this chain across independent tools adds to existing practice.

8.7 A prohibited effect and an unmentioned path

Now suppose an exploratory candidate includes a branch that edits a report's status to make it match the finalized-report filter. That behavior conflicts with the explicit prohibition on editing provider records. In a controlled environment, a suitably constrained action interface can reject the attempted edit before dispatch. A graph analysis can help locate the branch; it does not supply the enforcement by itself.

A repair agent proposes removing the edit and retaining the required filtering behavior. The candidate must still demonstrate that it retrieves the appropriate

reports. Deleting the entire retrieval operation would eliminate the prohibited effect but would fail the affirmative requirement. Acceptance therefore considers both permitted effects and required outcomes. Weakening the no-edit rule to make the candidate pass would require separate authority and a new contract version.

By contrast, a new helper that normalizes whitespace in displayed titles is not forbidden merely because the contract does not enumerate it. If normalization could alter a consequential source value, the system raises an interpretation question or applies an already accepted preservation rule. Otherwise it can remain an implementation detail. The mechanism governs supported obligations rather than treating every newly discovered branch as a defect.

The contrasting outcomes follow from the accepted commitments: the write conflicts with an explicit constraint, while the formatting helper requires a relevance decision. Discovery alone decides neither case.

8.8 Samples: decision-model checks and admission records

These are illustrative samples of one possible implementation, not executable DotPrompt syntax or a completed standard. The .prompt extension in the definition sample does not imply compatibility with Genkit Dotprompt. The excerpts share the identifiers from section 8.6 and show the records exchanged rather than a complete application configuration.

Account equality is suited to a deterministic check, C-account. A decision-model check can assess source language about finalization against O-selection, returning a scoped verdict with its evidence. It cannot establish the account identity or the completeness of the retrieved set. Missing or conflicting source evidence remains unresolved.

```
check_id: C-selection
definition: D1
obligation: O-selection
method: decision_model
evaluator: review-model@1
rubric_version: finalization-rubric@1
inputs: [record_snapshot,
 ↪  source_status, source_link]
question: >
  Does the supplied source evidence
   ↪  establish that this
  record is finalized under D1's
   ↪  accepted selection rule?
result:
  verdict: [satisfies, violates,
   ↪  unresolved]
  required: [verdict, reason,
   ↪  evidence_refs]
scope: supplied_record_snapshot_only
on_missing_evidence: unresolved
```

The check itself needs evaluation. These fixture expectations come from an authorized evaluation policy; they are not recorded copies of the model's answers. They demonstrate two cases, not an adequate validation suite or a measured result.

```
suite: selection-check-fixtures@1
check: C-selection
oracle_authority:
 ↪  evaluation-policy-owner
cases:
  - id: explicit-draft
    source_status: "Draft - pending
     ↪  review"
    expected: violates
  - id: absent-status
    source_status: null
    expected: unresolved
# Expected labels are approved
 ↪  independently of model outputs.
```

The discovery report and eventual evidence remain separate records. R2 and M2 describe the r2 concern; E3 refers to the repaired r3 candidate and identifies its individual check results. Naming a check in E3 is not evidence that it ran: the referenced results must exist, expose their scopes, and satisfy the governing policy.

```
report: R2
baseline: r1
candidate: r2
method: dependency_analysis
impact:
  id: M2
  obligation: O-account
  basis: [retrieval-to-paginate,
   ↪  retrieval-to-O-account]
  unresolved: [runtime-adapter-target]

evidence:
  id: E3
  package: K3
  implementation: r3
  definition: D1
  context: X1
```

```
  checks: [C-account, C-selection]
  result_refs: [account-check-run-3,
   ↪  selection-check-run-3]
```

P1 consumes those records and the outcomes for every required obligation, including ordering, completeness, and excluded effects from D1. A model assessment is an input to that policy. A required unresolved judgment blocks admission unless an authorized, declared alternative establishes the obligation; changing the candidate does not change the policy.

```
policy: P1
candidate: K3
definition: D1
context: X1
requires:
  - bindings_match_candidate
  - required_capabilities_granted
  - deterministic_account_check_satis
   ↪  fied
  - required_obligation_assessments_a
   ↪  ccepted
  - consequential_mappings_resolved_o
   ↪  r_fallback_completed
on_unresolved: block_admission
model_verdict_alone_authorizes_releas
 ↪  e:
 ↪  false
```

Together, these samples make the proposed division inspectable: a definition establishes the target, a skill proposes execution, checks produce evidence, and an authorized policy governs admission. The fixture suite and field names illustrate the design; they do not assert that any existing package or model implements it.

9 The contribution boundary

The worked example requires several mechanisms, many of which already exist. The contribution question is whether their handoffs preserve the same decisions without bespoke reinterpretation. The comparison below separates reusable mechanisms from the semantic obligations a shared profile would need to specify.

Lifecycle relationships. OSLC Core supports loosely coupled integration among heterogeneous tools through linked resources and domain specifications. It is a candidate foundation for identifying obligations and connecting their lifecycle records. DotPrompt would need to specify how a supported profile distinguishes proposed interpretations from accepted commitments and how those distinctions govern transitions. This paper does not establish that OSLC needs extension to express the rules; demonstrating a mapping is part of the proposed implementation work. [22]

Build history. SLSA provenance describes how software was produced. An execution package should reference that evidence where available. The additional question in this proposal is whether evidence supports an accepted behavioral obligation for the candidate and execution context under review. Build provenance can inform that decision without answering it alone. [23]

Agent construction and execution. LMQL, DSPy, and SGLang provide established approaches to programming, optimizing, and executing model-driven components. Harnesses organize tools, context, and feedback. These mechanisms can construct or evaluate packages. The proposed interoperability obligation is that their outputs retain the same requirement, authority, and evidence bindings when another component consumes them. [8, 9, 12, 13, 14, 25, 26]

Discovery and verification. CodeGraph, Graphify, and the formal-verification approaches discussed in section 4 supply relationships, analyses, or checked claims. DotPrompt does not introduce those capabilities. Its impact assertions make explicit which obligation a finding bears on, why the mapping is justified, and what uncertainty remains. Its evidence records preserve the conditions under which a checked claim can support acceptance. [10, 16, 17, 28, 29, 30]

Existing integrated pipelines. Versioned requirements, CI checks, policy engines, and release gates can already preserve these distinctions within a well-engineered system. They are the baseline. The proposed benefit arises only where exchanging work between independent systems otherwise requires repeated, bespoke interpretation or loses a consequential rule. An integrated platform with equivalent semantics is not deficient because it lacks the proposed name.

The narrow contribution is therefore a candidate interoperability contract with observable rejection behavior. A receiver must not accept evidence for the wrong revision, treat an inferred obligation as authoritative without the required decision, or drop a mandatory constraint during exchange. These are proposed conformance rules. Whether standardizing them re-

duces integration effort and failures remains an empirical question.

This leaves two legitimate outcomes for implementation work: an existing representation may carry the required semantics through a profile, or a specific missing rule may justify an extension. Neither outcome requires a new universal format. Section 11 defines the comparison needed to establish whether the resulting agreement is useful.

The working name also remains unsettled. Genkit already uses Dotprompt for a separate prompt-management format. No affiliation or compatibility is implied. Naming and any new syntax should follow the interoperability investigation, not precede it. [15]

10 Counterarguments and limits

10.1 The specification may move complexity rather than remove it

A contract graph adds records, links, migrations, and authorities that must themselves be understood. If an ordinary application already has one coherent source of requirements and a dependable delivery system, another representation may make the work harder. The relevant comparison is with well-engineered alternatives, not a deliberately weak prompt-only workflow.

The proposal should therefore be evaluated on the amount of interpretation and coordination it removes, including the costs of maintaining the contract. Counting recorded obligations or generated evidence is not sufficient. A small task should not require an elaborate ceremony merely to satisfy the architecture.

10.2 Explicitness cannot guarantee completeness

A representation may be precise and wrong. It may faithfully record a request that omits a critical stakeholder, deployment condition, or data path. It may contain an acceptance rule too weak to establish the intended property. Formalizing such a mistake can make it more durable rather than more correct.

DotPrompt must preserve the distinction between contract conformance and suitability for the real task. Conformance establishes behavior relative to declared obligations. Suitability requires additional inquiry and evidence. A system should permit contracts to be challenged by operational observations and stakeholder feedback, with changes reviewed under the appropriate authority.

10.3 A model can generate the contract too

Separating proposal from acceptance does not guarantee independent understanding. If the contract, implementation, and evaluator all inherit the same mistaken interpretation, every declared rule may be followed while the resulting feature is wrong.

The relevant failure is correlated omission. A requester asks for reports, the generated definition omits archived accounts, and neither the implementation nor its checks consider them. Authority separation can prevent unauthorized criteria changes without detecting this shared mistake.

A useful evaluation must therefore include requirements or counterexamples supplied independently of the construction process. The representation should make their introduction and the resulting contract revision inspectable. Its contribution is traceability and controlled correction; the discovery of missing requirements still depends on inquiry beyond the existing contract.

10.4 Long-term architecture resists simple predicates

Maintainability depends on future changes, team practices, and requirements that are not yet known. There is no general predicate in this proposal that certifies good architecture. Dependency checks, compatibility rules, and change-locality measures can expose some problems; they do not exhaust the concept.

This is an important limit of the response to Horthy's observations. Making obligations explicit may help preserve design decisions and reveal violations. It does not follow that the representation captures every architectural quality that matters. Longitudinal evidence and better evaluators may be needed in addition to better interfaces and enforcement.

10.5 Strict controls may reduce useful autonomy

The unresolved-impact rule in section 5.5 can make a system refuse work. Broad propagation may send every change for expensive assessment; narrow propagation may miss a consequential dependency. Both errors can persist even when the representation and admission mechanism behave exactly as specified.

The evaluation must therefore count completion, delay, intervention, and error severity together. A system that prevents all unsafe releases by admitting

nothing has not achieved useful autonomy. Policies can permit bounded uncertainty where appropriate, but that choice must remain visible in the acceptance claim.

A reliability measure needs a defined event and denominator. Schema-valid responses, correct task outcomes, and successful authorized effects are different measures. Counting only admitted candidates can hide excessive rejection; counting only completed tasks can hide consequential failures.

10.6 Continuous evaluation may cost more than it saves

A complete implementation could be economically impractical even if each individual mechanism works. Instrumentation consumes resources; graph maintenance and evidence retention consume storage and engineering effort; model-based assessment consumes inference capacity; new findings generate further analysis and tests. Broad impact propagation can turn one small change into many evaluations. The proposal cannot assume that falling model prices will make this coordination free.

A useful cost model separates discovery, mapping, assessment, verification, and maintenance. For a chosen interval, let the evaluation demand be the number of distinct changes requiring assessment multiplied by their average assessment cost, plus the cost of required checks, instrumentation, storage, and contract upkeep. Deduplication may reduce repeated work; it does not reduce the need to assess distinct consequential changes. If incoming work persistently exceeds processing capacity, a system promising timely evaluation accumulates a backlog regardless of how quickly it generates code.

The workload controls in section 5.8 bound resource use, but bounds do not establish a favorable tradeoff. Deferral can move cost into release delay, and reuse saves work only where evidence remains applicable.

For example, a shared helper may appear on twenty observed paths while only two accepted obligations depend on the changed behavior. Grouping assessment by those obligations could avoid twenty independent reviews. That saving depends on a defensible mapping; an incomplete graph cannot justify ignoring the other paths. If the mapping is unresolved, broader analysis may erase the expected efficiency benefit. This is a limitation to measure, not an exception to conceal.

Feasibility today is therefore a question about a specific workload and implementation. This paper establishes neither a price per accepted change nor a break-even point. A prototype should report evaluation lag, resource use, intervention time, and repeated validation alongside failures detected. If equivalent assurance is cheaper with a conventional pipeline, the added machinery is not justified for that workload.

10.7 Enforcement has its own trust boundary

A compromised executor can lie about a check. An adapter can misclassify an external action. An administrator can grant permissions that defeat intended isolation. Cryptographic signatures establish the origin and integrity of a statement under their assumptions; they do not establish the statement's truth.

A deployment must identify which components are trusted to enforce capabilities, record evidence, and authorize transitions. The more authority these components hold, the more carefully they need to be engineered and reviewed. The specification cannot make its own implementations trustworthy merely by naming the required properties.

11 How the hypothesis could be evaluated

The paper reports no experimental outcomes. A useful evaluation program would separate conformance of an implementation from empirical benefit of the approach.

11.1 Conformance: does the implementation preserve the declared rules?

Conformance testing should control the information given to a component and inspect its externally visible decision. Fix model responses and source observations where possible, then vary a binding, permission, capability, or delivery condition. This isolates an enforcement failure from variability in model judgment.

Appendix A specifies candidate mismatches and expected outcomes. For each case, the test fixture should name the supported profile, starting state, mutation, and permitted final states. A test passes only when the required rejection, invalidation, pause, or reconciliation is observable; a parser accepting the record is insufficient.

Run the same fixtures across component boundaries. Export an obligation and evidence from one implementation, consume them in another, and compare the resulting decisions. A failure introduced only by exchange identifies an interoperability defect. These tests establish adherence to the rules; the separate comparison below measures whether adhering to them improves the workflow.

11.2 Empirical value: does the representation help?

The baseline must include competent engineering: versioned requirements, reviewable plans, ordinary tests, access controls, and release practices. Hold tasks, models, budgets, and operational conditions comparable. Otherwise, an improvement may measure basic engineering or additional compute rather than the value of shared contracts.

The primary comparison should hold discovery and verification mechanisms fixed while changing how independent tools exchange obligations and evidence. Use the same code intelligence, checks, and admission policy with ordinary integration in one condition and a shared profile in the other. Include an integrated platform that already preserves equivalent semantics. Measure lost or silently changed obligations, invalid evidence reuse, integration and migration effort, and total cost per accepted change. Contract authoring and maintenance count toward that cost.

Follow sequences of changes rather than only fresh tasks. Include a new observed path, a suspected but unreachable path, an omitted dependency, an explicitly prohibited effect, and a legitimate unmentioned detail. Report missed obligations, false impact mappings, unnecessary reevaluation, repair-induced regressions, and effort to resolve ambiguity. Introduce delayed or duplicated notifications, wrong-revision reports, and unavailable analyzers to distinguish interface conformance from discovery quality.

A separate concurrency comparison should hold the contract, analyzer, models, and required test coverage constant while changing when evaluation starts. Compare evaluation after a completed change with evaluation during development. Exercise overlapping edits, superseded snapshots, slow evaluators, and questions that cannot be resolved automatically. Measure time to the first actionable finding, false interruptions, stale findings acted upon, downstream repair effort, reruns, and end-to-end admission time. Required work left pending is not a speedup.

Include inference, instrumentation, storage, contract upkeep, and investigation in the economic comparison. Test ordinary workloads and bursts that exceed processing capacity. Earlier warnings are useful only if their benefits justify the resulting interruptions, assessment work, and release delay.

11.3 Interpreting the results

The comparisons above can succeed or fail independently. Correct rejection of stale evidence supports a conformance claim even if maintaining the contract costs more than it saves. Reduced rework supports an operational benefit only if the comparison preserves required verification and accounts for deferred work.

The broader hypothesis would be weakened if shared profiles still require extensive bespoke reinterpretation, if maintenance effort exceeds the inconsistencies removed, or if blocking and interruption cause users to bypass the system. Equivalent results from a simpler integrated pipeline would limit the case for adoption to other boundaries rather than establish a universal benefit.

Success on one profile supports that profile under the evaluated conditions. It does not establish complete requirements discovery, long-term architectural quality, or dependable autonomy across unrelated workloads. Claims should expand only as the evaluations do.

12 A community path to a specification

The first implementation should make the proposed comparison possible. Use one repository, one accepted definition, a discovery producer, and two consumers: an evaluator and an admission component. Implement the change sequence in section 8 and the relevant fixtures in Appendix A. A local process boundary is sufficient; distributed infrastructure is not a prerequisite.

The initial deliverables are a scoped profile, exchange examples with immutable identities, an executable conformance suite, and a mapping to existing lifecycle and provenance representations. A failed mapping should identify the exact missing rule. A successful mapping should be retained rather than replaced for the sake of a new format.

A second independently built implementation should consume the same records and fixtures. Disagreement becomes input to the specification: an ambiguous rule, an unsupported capability, or an implementation defect must be identified rather than hidden by an adapter written only for the demonstration.

Only rules needed to preserve shared behavior belong in the normative core. Scheduling advice, prompting techniques, and model-specific optimizations belong in guidance unless evidence and interoperability requirements justify making them mandatory. Revision and migration procedures must allow participants to challenge overbroad defaults and unimplementable obligations.

Non-engineers should be able to use the resulting system without reading its specification. That requires a further interface test: can a requester understand a consequential unresolved question and the scope of the decision they authorize? Explicit machine records are useful only if the people defining intent can still recognize what the system is committing to do.

This sequence turns the paper's proposal into a bounded community task: implement one exchange, expose disagreement, measure its cost, and revise the semantics before expanding their scope.

13 Conclusion

The argument began with a coordination problem: each stage can succeed locally while reconstructing a different meaning of the request. The proposed contract addresses that problem by preserving the authority behind obligations, the implementation and conditions behind evidence, and the rule behind each consequential decision.

Execution discovery makes this representation responsive to change. An observation becomes an inspectable impact assertion; assessment determines what evidence remains usable or what work is required; admission acts on those results. The worked example shows how that chain can preserve a requirement through uncertainty and repair without making the proposing component its own authority.

The case for a shared specification now rests on a specific test. Independently implemented tools must exchange those records while preserving rejection and admission behavior, at a cost justified against competent existing practice. A profile over existing standards may satisfy that test. If the common representation merely relocates interpretation effort, the hypothesis fails.

Dependable autonomy remains the aim. Natural language can make software creation accessible, while explicit obligations make its consequences reviewable and enforceable. The next step is a narrow implementation that allows this relationship to be tested.

A Candidate conformance cases

These are proposed cases, not executed tests or results. The expected behavior is conditional on a profile supporting the stated capability.

Table 3: Candidate conformance cases

Case	Deliberate mismatch	Expected observable behavior
Unaccepted inference	An agent treats a proposed interpretation as a commitment	Affected advancement is blocked until an authorized rule accepts it.
Changed acceptance rule	An implementation agent weakens a required check	A contract change is proposed; old evidence does not authorize the new rule.
Stale package evidence	A check refers to an earlier implementation or definition revision	Evidence is not used for the new package without a defined validity rule.
Missing capability	A required isolation or provenance mechanism is absent	The implementation reports incompatibility rather than silently degrading.
Scope expansion	A request targets a different account or destination	The effect is rejected under the existing capability.
Authentication expiry	A previously valid session no longer authenticates	The run enters the declared handoff or failure state.
Late response	Approval arrives after cancellation	The response does not restart or authorize the cancelled run.
Uncertain effect	A destination may have acted before a timeout	The executor reconciles or reports unresolved state before unsafe repetition.
Partial source	Retrieval cannot establish the full requested set	The result is explicitly partial or fails under the acceptance policy.
Provider change	The schema matches but the judgment provider changes	The declared compatibility policy determines required reevaluation.
Cross-tool export	An adapter drops a mandatory obligation	Import or admission fails visibly.
Concurrent candidate	Two agents revise the same contract base	A version conflict is exposed rather than resolved by silent overwrite.
Changed path	New retrieval behavior depends on a helper absent from prior evaluation	Affected obligations require applicable evidence before admission.
Unmentioned behavior	A new internal branch has no explicit prohibition	The system applies declared rules or raises a consequential question; absence alone does not justify deletion.
Prohibited effect	A candidate attempts an explicitly excluded write	A capable boundary rejects dispatch; a proposed repair remains subject to affirmative requirements.
Lost notification	A subscriber never receives a change event	Admission still requires the profile's revision-bound analysis and evidence.
Incomplete analysis	An analyzer cannot resolve a relevant dependency	The limitation remains explicit and the profile's fallback policy governs advancement.

Additional conformance cases follow from concurrent evaluation:

Table 4: Concurrency conformance cases

Case	Deliberate mismatch	Expected observable behavior
Superseded finding	Evaluation returns for a snapshot that has changed	The result cannot approve the new candidate without a valid applicability rule.
Repeated report	The same observation arrives twice	Processing preserves identity and avoids duplicate effects or conflicting decisions.
Saturated evaluator	Required assessments exceed available capacity	Backlog is visible; required evidence remains pending and affected admission stays blocked.
Admission race	A policy or package changes after evaluation	Admission detects the changed binding and rejects or reevaluates the decision.
Conflicting dependency	Another branch changes a dependency used by the candidate	The conflict is revision-bound; an unexamined merge does not inherit acceptance.

Mapping and proof cases

A profile supporting change review should also expose the basis of impact mappings. Given a report with an unresolved dynamic dependency, it must retain that uncertainty and apply its declared fallback rather than certify that no obligation is affected. A mapping that references a superseded commitment cannot settle assessment of its replacement without an applicability decision.

A profile accepting formal proofs should reject an attempt to use a conditional proof when its caller preconditions remain undischarged. It should also distinguish a counterexample, a checked proof, and an unfinished or timed-out proof attempt. Changing the implementation or a trusted assumption requires reevaluation of applicability even when the stored proof object remains unchanged.

B Contribution and evidence ledger

Table 5: Contribution and evidence ledger

Statement	Status in this paper	Evidence needed to go further
Repeated interpretation can create inconsistent commitments	Architectural diagnosis with illustrative mechanisms	Comparative studies of actual handoffs and requirement changes
Shared contracts reduce lifecycle inconsistency	Hypothesis	Controlled comparisons against competent engineering baselines
Explicit records can support observable transition rules	Proposed mechanism	Independent implementations and a conformance suite
A narrow executor can implement part of the proposal	Implementation strategy	A working profile with demonstrated enforcement boundaries
The approach improves long-term maintainability	Not established	Longitudinal studies and defensible quality measures
Typed decisions establish semantic correctness	Not claimed	Task-specific evaluation remains necessary
More test volume establishes reliability	Rejected as a sufficient inference	Requirement coverage, failure sensitivity, and operational evidence
Full autonomous development is inevitable on a fixed schedule	Not claimed	The paper presents a direction and engineering ambition, not a forecast
Concurrent evaluation reduces downstream rework	Hypothesis	Equal-coverage comparisons measuring actionable-feedback latency, repair effort, reruns, and total cost
Continuous evaluation is affordable for arbitrary workloads	Not established	Workload-specific cost and backlog measurements under explicit evaluation budgets
A shared specification can eliminate unknown requirements	Not claimed	Requirements discovery remains an open, context-dependent problem

References

[1] Ben Moseley and Peter Marks. **Out of the Tar Pit.** February 6, 2006. Architectural position paper. Paper.

[2] David Heinemeier Hansson. **The Rails Doctrine.** Particularly "Convention over Configuration." Primary source.

[3] Adam Wiggins. **The Twelve-Factor App.** Software-as-a-service methodology. Primary source.

[4] Ecma International and TC39. **ECMAScript Language Specification.** Living specification; introduction and conformance provisions. Specification.

[5] Dex Horthy. **Why Software Factories Fail.** AI Engineer World's Fair, 2026. Practitioner talk; the accompanying organizer article discusses the maintenance experience and its evidential limits. Talk and organizer article. Recording.

[6] TypeSafe. **System One.** Provider documentation describing Jev, typed decisions, probabilities, and integration with deterministic checks. Documentation.

[7] OpenAI. **Build more responsive apps with the Decisions API.** October 22, 2026. Provider announcement. Primary source.

[8] Anthropic. **Effective harnesses for long-running agents.** November 26, 2025. Engineering report. Primary source.

[9] HumanLayer. **12-Factor Agents.** Engineering principles for LLM-powered applications. Project.

[10] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. **Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning.** Findings of EMNLP, 2023. Paper.

[11] OpenAI. **The Hugging Face incident and the road ahead.** August 26, 2026. First-party incident account; used with its reduced-safeguards qualification. Report.

[12] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. **Prompting Is Programming: A Query Language for Large Language Models.** PLDI, 2023. Paper.

[13] Omar Khattab et al. **DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines.** 2023 preprint. Paper.

[14] Lianmin Zheng et al. **SGLang: Efficient Execution of Structured Language Model Programs.** 2024 revision. Paper.

[15] Genkit. **Managing prompts with Dotprompt.** Documentation for a separate project with a similar name. Documentation.

[16] Colby McHenry and contributors. **CodeGraph.** Project description and open-source repository; parsing, code relationships, MCP access, and impact analysis. Project. Repository. Reviewed November 1, 2026.

[17] Graphify. **Graphify: code knowledge graphs and verification.** First-party product description; feature claims are not independently evaluated here. Project. Reviewed November 1, 2026.

[18] Ruby on Rails. **Active Support Instrumentation.** Official guide to instrumentation hooks and subscribers. Documentation. Reviewed November 1, 2026.

[19] React. **useEffect.** Official reference for synchronization and declared dependencies. Documentation.

[20] Model Context Protocol. **Resources, protocol revision 2025-11-25.** Optional resource subscriptions and resource-change notifications. Specification.

[21] Ankur Tyagi, CodeRabbit. **How to deploy and integrate MCP servers with CodeRabbit.** November 20, 2025. Documented context integration, not evidence of the prospective subscription-triggered workflow. Primary source.

[22] OASIS Open Projects. **OSLC Core Version 3.0, Part 1: Overview.** Lifecycle-tool integration through linked resources and shared capabilities. Standard.

[23] SLSA. **Provenance, version 1.2.** Software supply-chain provenance and its scope. Specification.

[24] Martin Fowler. **Semantic Diffusion.** December 14, 2006. Terminology drift and the importance of preserving explicit meanings. Article.

[25] Vivek Trivedy, LangChain. **Improving Deep Agents with harness engineering.** February 17, 2026. First-party benchmark and engineering report. Report.

[26] Wilson Lin, Cursor. **Scaling long-running autonomous coding.** January 14, 2026. Engineering experiments and coordination limitations. Report.

[27] Cursor. **Cursor is now a part of SpaceX.** August 14, 2026. Corporate announcement, not independent harness evaluation. Announcement.

[28] Google DeepMind, AlphaProof and AlphaGeometry teams. **AI achieves silver-medal standard solving International Mathematical Olympiad problems.** July 25, 2024; updated with the 2025 methodology publication. Technical account.

[29] Tudor Achim et al. **Aristotle: IMO-level Automated Theorem Proving.** October 2025. Research preprint. Paper.

[30] Harmonic. **Aristotle software verification.** Provider documentation describing proved properties and caller obligations. Reviewed November 1, 2026. Documentation.