Semantic Eduction: From Latent Knowledge to Executable Semantics for Coding Agents

Ulvi Guliyev

Abstract

Coding agents can design, implement and evolve software while relying on an incomplete or incorrect account of its intended meaning. Consequential knowledge can be latent in documents, data, reports, worksheet formulas, operating practices and subject-matter expertise, as well as code and configuration where software exists. Some practices embody applicable obligations; others are accommodations or defects. A successful build or passing regression suite does not by itself establish semantic correctness.

This paper proposes semantic eduction as an enabler for mitigating design and implementation drift during software development and evolution with coding agents. It draws out latent domain and process knowledge and makes it explicit as a justified semantic foundation for assessing proposed designs and implementations. Eduction can begin before a target application exists. Two conditional results support the argument: an automated acceptance mechanism cannot distinguish acceptable and unacceptable candidates when its trusted decision basis omits the relevant distinction; intended semantics cannot be uniquely reconstructed from evidence compatible with conflicting requirements. These are information limitations.

Rigorous requirements and architectural specifications can communicate intent. Their approval and machine-readable organization do not, by themselves, establish implementation conformance. A correct requirement and a correct implementation are separate propositions. Documentation contributes to mechanical assurance when its relevant meaning has a defined checking interpretation and an established implementation connection. Controlled English and executable models demonstrate that this boundary concerns capabilities rather than file format.

The method establishes applicable behavioral and design obligations, grounds their concepts in domain phenomena, establishes supporting mechanisms and assumptions, and justifies a formal contract. This foundation guides an agent's initial design and implementation; actual correspondence evidence supports admission of the first implementation and subsequent changes. Human practitioners examine the relevant business and technical environment and construct a justified eduction map: the sources, consequential dependencies, interpretations and abstractions that guide machine extraction, inference, synthesis and verification. Machine-checkable contracts support covered properties; explanatory documents retain purpose and evidence. Lean, other proof assistants, verification languages, source contracts and behavioral models can supply the representations. Payment and Excel-based financial operations illustrate the reasoning; a practitioner account illustrates application. The integrated method's effectiveness at mitigating drift has not been independently evaluated.

1 The engineering claim

This paper concerns software development and evolution in which coding agents propose designs, implement an application or change it. Its objective is to establish the semantic foundation needed to mitigate design and implementation drift from the first design onward. Semantic eduction draws out latent domain and process knowledge, establishes applicable requirements,

design commitments, meanings and assumptions, and supplies the connections needed to assess the agent's work. An existing target application is one possible source of evidence, rather than a prerequisite.

Design drift is a departure of a proposed or maintained design from accepted responsibilities, interfaces, data ownership or structural and behavioral constraints outside an authorized change. **Implementation drift** is a divergence of code or operating behavior from the accepted design and behavioral contract outside that change. Either can occur in the first application: a prior codebase is unnecessary when accepted obligations already supply the reference. An implementation can exhibit both, for example by bypassing a required component boundary while altering an effect's meaning. Deliberate evolution identifies the intended differences and the obligations that remain applicable.

The supporting information arguments are general. This paper applies them to a specific engineering purpose: guiding coding agents with justified system semantics and assessing their design and implementation changes against that basis. Correctness depends on the accepted contract and the evidence connecting the proposed and actual system to it; confidence in the agent's interpretation does not supply that evidence.

The proposed method produces a domain- and system-specific foundation for development and evolution with coding agents. It reconstructs consequential semantics, expresses behavioral and design obligations precisely, establishes their relationship to a proposed and then actual implementation, and supplies a maintained basis for checking subsequent agent changes. The recurring obligations provide a basis for standardizing the eduction discipline for this purpose. The educed business rules, design constraints, abstractions, assumptions and implementation connections must be established for the particular application; independent reproducibility of the complete method remains unestablished.

1.1 Claim A concerns mechanical assurance

An automated acceptance process needs a justified, machine-checkable basis for every semantic property it claims to guarantee. Where objectives remain outside that basis, the process cannot independently establish that an admitted implementation or change satisfies them.

For development and evolution with coding agents, the engineering consequence is to construct an explicit formal contract, executable specification or equivalent checkable representation for the behavioral and design properties requiring mechanical assurance. These representations supply obligations for the agent's initial work and criteria for assessing its subsequent changes. This need not be one complete model of the enterprise or a standalone file in a particular language. A verified interface, a type-level constraint or a soundly enforced database invariant can supply part of the representation.

Reliable practice also includes human review and empirical evidence. This paper does not claim that useful software cannot be produced without formal methods, or that every empirically reliable development process requires a theorem prover. Its claim concerns a defensible semantic guarantee made by an automated acceptance process. A measured error rate on a stated population and a proof covering all modeled behaviors are different claims.

1.2 Claim B concerns reconstruction from incomplete evidence

When the evidence considered is compatible with conflicting intended behaviors, no procedure using only that evidence can guarantee reconstruction of the uniquely

correct intended contract. Additional evidence or an authorized decision is necessary.

For development and evolution with coding agents, the proposed method requires comprehensive expert-led examination of the relevant business and technical environment. Practitioners locate and examine the sources of latent knowledge, reconstruct consequential relationships across domains, distinguish applicable obligations from defects and incidental structure, and justify the abstractions through which machines will educe that knowledge. This investigation produces an eduction map specifying the semantic path from distributed evidence to the contract. It applies before a target application exists and across repository and team boundaries. Section 6.1 defines the map and derives its information limit from Proposition 2.

1.3 Scientific status and scope

The paper combines a targeted critical synthesis of primary research, industrial reports and current project documentation with a proposed method, explicit mathematical arguments and a practitioner experience report informed by repository inspection. Sources were consulted through 5 October 2026. Older foundational results are used for enduring distinctions; recent reports and maintained documentation establish current capabilities. This is not an exhaustive systematic review or a claim to have proved the absence of every possible automated approach.

Source selection follows the paper's claims: requirement adequacy, correctness oracles, design recovery, specification mining, formalization, implementation correspondence and assurance through change. Primary papers, author manuscripts and official project documentation provide the evidence; citation following connects these subjects to adjacent approaches. Inclusion depends on whether a source establishes a definition, mechanism, relevant application or counterexample. Positive automation results and controlled-language specifications are included because they constrain the central claims. Preprints and industrial reports retain their stated evidential status. Product documentation establishes described capabilities, while evaluated outcomes require the corresponding study. This account specifies the selection rationale without claiming a reproducible systematic search protocol.

Propositions 1 to 3, the documentation corollary in Section 3.7 and the eduction-map corollary in Section 6.1 are arguments stated and proved within this paper. They are not attributed to an external publication or presented as newly discovered mathematical results. Their deterministic counterparts are also checked in the companion Lean development described in Appendix A. Published studies support particular mechanisms and applications. They do not, collectively, constitute an empirical evaluation of the complete method. Sections 9.1 to 9.4 use a constructed payment example. Sections 9.5 to 9.7 present Signal repository evidence and the author's first-hand account of an agent comparison, with their evidential limits stated separately. Section 9.8 adds a constructed example of educing latent semantics from Excel-based financial operations before a dedicated application exists.

1.4 Is rigorous documentation enough?

Rigorous documentation is a source of intent. It is not, solely by being rigorous or machine-readable, evidence that an implementation preserves that intent. The argument applies to approved BRDs, architectural specifications and implementation specifications expressed in prose, diagrams and examples. It does not depend on those documents being casual, neglected or ambiguous.

For an automated conformance or preservation guarantee, the workflow must establish both what is permitted and why the admitted implementation obeys it. Document schemas and

retrieval systems can help supply the first; they do not automatically establish the second. A workflow that gives its documents executable semantics and establishes conformance has supplied the semantic engineering capability described here. Section 3.7 derives the distinction, considers the strongest documentation objection and identifies supporting industrial evidence and counterevidence.

2 System meaning and the semantic foundation

2.1 Objectives and semantics

A system objective describes an intended result or obligation. Some objectives are software properties, such as preserving an account balance during a transfer. Others concern outcomes in the wider world, such as reducing losses or improving customer experience. Formal software assurance addresses a justified translation of those objectives into observable obligations. It does not prove the business outcome merely by proving the translation's implementation.

System semantics is the meaning assigned to states, operations, data and interactions. It includes distinctions such as an instruction versus a retry, reserved stock versus available stock, an accepted payment versus a settled payment, and a timeout versus a confirmed failure. Correctness depends on retaining the distinctions relevant to the covered claim.

Latent knowledge includes meanings, rules and dependencies embodied in practice without being assembled into an explicit, justified contract. A report can show a closing balance, a worksheet formula can calculate part of it, and an accountant or controller can know when an adjustment must be introduced. Educing the meaning requires reconstructing how those sources jointly determine the result, including exceptions and manual decisions. The same work applies when a project starts from documents, data, worksheets or subject-matter expertise and has no target application yet. Stated requirements contribute evidence; eduction additionally surfaces the implicit knowledge on which their correct interpretation and implementation depend.

Semantics is also latent relative to a consumer and a purpose. A rule can be explicit in an architectural document yet unavailable to an automated checker as a precise decision criterion. Conversely, an executable constraint can be explicit to a machine while its business purpose is poorly understood by maintainers. Human-readable documentation remains evidence and explanation. Reading that documentation through a language model does not, by itself, establish that the resulting interpretation is faithful or complete.

Zave and Jackson distinguish requirements R, implementable specifications S and relevant domain knowledge K. Their account requires the specification together with domain knowledge to entail the requirement. In logical notation:

$$K \wedge S \Rightarrow R.$$

A specification achieves its intended result because particular facts about the environment hold. Establishing those facts and the meaning of the terms is therefore part of establishing an adequate specification. A proof of the implication establishes what follows from its premises; the applicability of those premises remains a separate evidential obligation. [1]

For the payment example developed in Section 9, a rule about repeated submissions achieves an obligation about financial effects only through justified knowledge of instruction identity, persistence and provider behavior. This is the technical reason eduction requires domain and system understanding. In the assurance account below, the relevant domain facts and operating premises are included in A, while α expresses the interpretation of implementation observations.

2.2 Complex systems and semantic uncertainty

This paper uses **high semantic entropy** as a descriptive term for difficulty establishing a coherent account of required and implemented meaning. It is not a thermodynamic quantity, a Shannon entropy estimate or a validated universal software metric. Relevant characteristics include:

- Rules dispersed across reports, worksheet formulas, data, operating practices and subject-matter expertise, as well as services, databases, jobs, configuration and external providers where present.
- Conflicting or incomplete accounts of what behavior is required.
- Exceptions or manual interventions whose trigger, authority or original justification is unstated or missing.
- Important behavior controlled by timing, concurrency, retries and partial failure.
- Interfaces whose data shapes are documented but whose meaning is ambiguous.
- Implementation features or dependencies outside the available analysis tools' supported models.

These characteristics can obstruct particular eduction or verification activities, including before a target application exists. They do not establish that all automation will fail. A large system can expose a small, well-defined protocol suitable for learning; a small workbook or process can contain a single business ambiguity that its formulas cannot resolve. A scientific account therefore identifies the concrete obstacle, the affected obligation and the evidence needed to resolve it.

For a covered obligation, **essential complexity** consists of distinctions and relationships whose removal would invalidate that obligation, a required capability or a supported environmental premise. **Incidental complexity** consists of representational or historical structure that can be changed without such a loss. Brooks distinguishes the conceptual structure of software from difficulties of its representation and warns that simplifying a model can discard the substance it must express. The definitions here apply that distinction to a particular eduction claim. [51]

These classifications require a justified account of the domain and mechanisms. Different settlement dates can express necessary business obligations; contradictory definitions of settlement can express semantic uncertainty. An old compatibility path may still satisfy a binding external interface, so age or apparent redundancy does not establish removability. High semantic entropy describes uncertainty about meaning; it does not classify every complex structure as unnecessary.

Change can reopen these obligations without increasing system complexity. Removing a component, replacing an integration or simplifying a recovery path can alter the assumptions or implementation relationship on which an earlier assurance depended. Section 8.6 develops this consequence; the argument does not require a law of increasing complexity or inevitably deteriorating documentation.

2.3 A formal account of assurance

Let I be an implementation together with the configuration and dependencies relevant to a claim. Let A describe assumptions about its environment. Let $\mathrm{Beh}_A(I)$ denote its possible behaviors under those assumptions. Depending on the property, behaviors include complete

finite executions and infinite executions, so that termination, deadlock and progress are not silently discarded.

Let S be a formal specification and let $\mathrm{Beh}(S)$ be the behaviors it permits. An abstraction α maps implementation observations to the concepts used by S. The basic implementation obligation is:

Implementation correspondence:

$$\alpha(\mathrm{Beh}_A(I)) \subseteq \mathrm{Beh}(S).$$

For a requirement r, a separate adequacy obligation states that the specified behavior, interpreted under the relevant domain assumptions, satisfies r:

Specification adequacy:

$$A \wedge S \models r.$$

The interpretation must preserve the distinctions on which the concrete requirement depends. If q is the corresponding property of abstract behaviors, applying the result to a concrete behavior t requires justification that $q(\alpha(t))$ implies r(t) under A. An abstraction that hides a second financial effect cannot establish a requirement excluding that effect merely because its abstract history satisfies q.

This notation summarizes proof obligations; it is not an algorithm for deciding arbitrary programs. Refinement mappings provide an established foundation for relating lower-level behaviors to higher-level specifications. Their construction and the conditions for their use are substantive verification work. [12]

A semantic foundation for implementation assurance comprises the accepted obligations R, the specification S, the assumptions A, the interpretation α and the evidence J supporting adequacy and correspondence. Denote it by:

$$B = (R, S, A, \alpha, J).$$

The specification can be realized through several connected artifacts. J can contain different kinds of evidence, provided their strength remains explicit. A proof, a bounded model-checking result and a successful test execution establish different conclusions. Merely assembling all five components does not establish that their claims are true.

Before a target application exists, eduction can justify R, S and relevant domain assumptions, define the intended interpretation of future observations, and establish a checking strategy. That semantic basis can guide agent design and implementation. Correspondence to a concrete I becomes an evidential obligation when the first implementation exists; a prospective mapping or a proof about S alone cannot establish it. The same distinction applies to each later implementation.

The terms describe different capabilities. A **formal specification** has a defined mathematical interpretation. An **executable specification** can compute results or enact modeled transitions for supported inputs. A **machine-checkable obligation** has a supported mechanical means of checking the relevant evidence or property. A theorem can be mechanically checked even when its specification is not an executable decision procedure for arbitrary program behavior. A test can execute a precise check while covering only selected behaviors.

The behavioral inclusion above directly addresses trace properties. Properties relating several executions, such as some information-flow requirements, need an appropriate relational or hyperproperty model. A choice of notation cannot silently narrow the intended obligation to make it easier to verify.

Architectural obligations can also concern implementation structure independently of externally visible behavior. Two programs may produce the same outputs while only one respects an accepted dependency boundary. For example, let $\mathrm{Dep}(I)$ be a justified extraction of the relevant dependencies and $\mathrm{Allowed}(S)$ the permitted dependency relation. A structural correspondence obligation can be $\mathrm{Dep}(I) \subseteq \mathrm{Allowed}(S)$. The meaning of a dependency, the mapping from source elements to components, and the extraction's coverage must be established. Behavioral refinement alone does not prove this structural claim; architecture recovery and conformance analysis address it explicitly. [40]

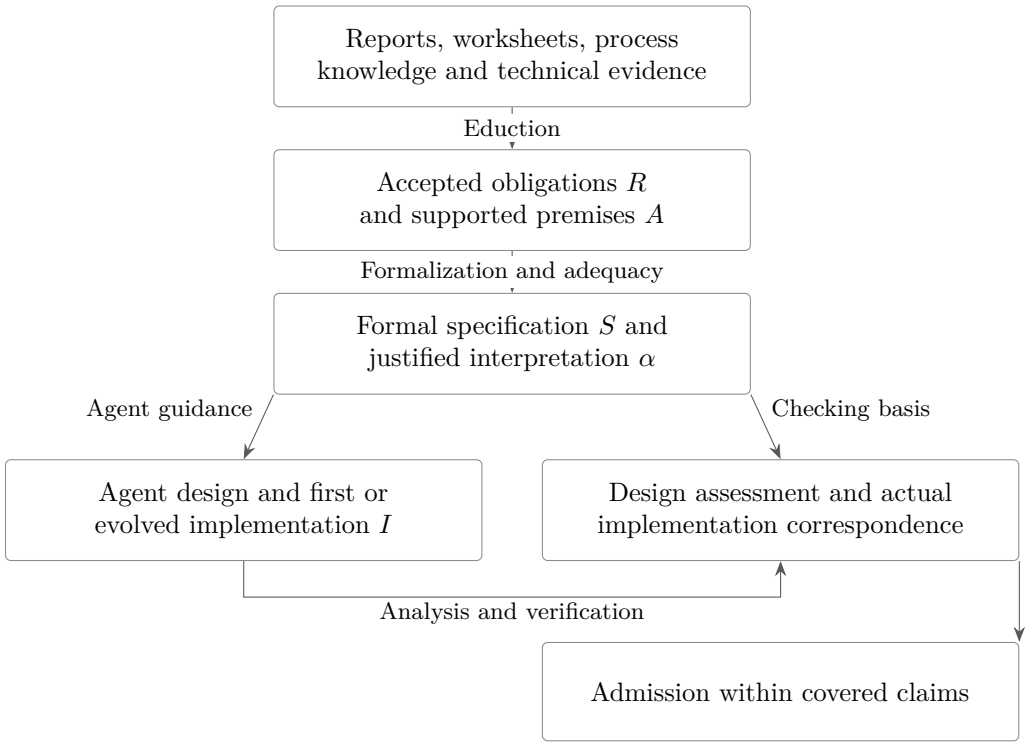


Figure 1: Eduction can establish meaning before a target application exists. The justified contract guides agent design and implementation; admission requires evidence about the actual candidate's behavior and structure.

2.4 A useful contract must be consistent and realizable

A useful assurance result covers a realizable service in its promised operating domain. Zave and Jackson's completion criteria require consistency of specification and domain knowledge as well as entailment of requirements, and distinguish what the machine can control from what its environment controls. Entailment alone is therefore insufficient. [1]

Applying that distinction to the foundation introduces four completion obligations:

- **Consistency.** The specification and assumptions admit relevant behavior together. A contradiction in A or S cannot supply evidence that the service satisfies R.
- **Supported starting conditions.** The actual initial or recovered state lies within the justified model, and the relevant operating domain is nonempty. A model that starts with empty history does not cover a service recovering outstanding instructions.
- **Required capabilities and inputs.** The contract retains every promised class of legitimate request and environmental interaction. Demonstrating one satisfactory execution does not

establish that all promised classes can be served.

- **Realizability and progress.** The implementation can meet its obligations using the observations and controls it actually possesses, for the admitted environmental behaviors. Required progress has explicit scheduling, availability and resolution premises; it cannot depend on knowing an unavailable future outcome.

These are obligations of the proposed method, not an assertion that arbitrary specifications can be decided automatically. They also distinguish a service guarantee from a selected safety theorem. A proof that no duplicate payment occurs can be sound while an implementation that rejects every payment fails the required service contract. Required capabilities belong in R alongside forbidden behaviors, and their preservation must survive changes to assumptions and input restrictions.

3 Why latent objectives cannot support independent automated assurance

3.1 The acceptance decision

Consider a set X of candidate implementations in their relevant operating contexts. A requirement r classifies each candidate as acceptable or unacceptable. Let o(x) be the information available to an acceptance mechanism: the code it analyzes, the observations it records, its specifications, configuration and other trusted inputs. Let g(o(x)) be its decision.

Calling the mechanism sound for r means that every accepted candidate satisfies r. Soundness alone admits the useless mechanism that rejects everything. Useful assurance also requires that the mechanism can accept candidates whose relevant correctness has actually been established.

Proposition 1. Indistinguishable candidates cannot receive a sound distinguishing decision. Suppose candidates x_{good} and x_{bad} have the same available representation, $o(x_{\text{good}}) = o(x_{\text{bad}})$, but r($x_{\text{good}}$) is true and r($x_{\text{bad}}$) is false. An acceptance mechanism depending only on o cannot both accept x_{good} and reject x_{bad}.

Proof. The mechanism receives the same input for both candidates and therefore makes the same deterministic decision. If it accepts x_{good}, it accepts x_{bad} and is unsound for r. If it rejects x_{bad}, it also rejects x_{good}. A randomized mechanism has the same output distribution for both inputs and cannot guarantee the distinguishing decision. This establishes the claim.

Equivalently, sound acceptance of an observation z requires every candidate compatible with that observation to satisfy the requirement:

$$g(z) = \textsc{accept} \;\Rightarrow\; \big(\forall x,\; o(x) = z \Rightarrow r(x)\big).$$

The mechanism can reject ambiguous classes while accepting sufficiently established cases. A distinction absent from its inputs requires additional evidence or an authorized restriction of the admitted scope. A distinction already present but difficult to interpret or verify is a different problem, concerning analysis capability or cost. The proposition establishes the information limitation; it does not select a unique eduction method or show that a particular agent workflow has that limitation without examining its actual inputs.

3.2 The specification is the decision basis

Barr et al.'s survey of the test oracle problem distinguishes generating executions from determining whether their results are correct. It reviews specified, derived, implicit and human-supported oracles. The relevant lesson is that exercising code and judging its meaning are different operations. [2]

A machine-checkable specification supplies a precise part of that decision basis. It can state an invariant, a permitted transition relation, a precondition and postcondition, a temporal obligation or a relation between old and new behavior. An executable reference model can also supply expected results. Which form is suitable depends on the property and the intended strength of assurance.

A passing test suite supplies evidence about the executions it checks and the assertions it contains. Property-based testing broadens the explored inputs. Differential testing compares implementations. These are valuable forms of executable semantics, but their evidential scope depends on the oracle and coverage. A test suite generated from the same mistaken interpretation as the implementation can consistently endorse the wrong behavior.

A trusted mechanism can enforce a property without representing it in a separate specification file. Examples include capability restrictions and invariant-preserving operations. Such a mechanism still embodies a particular interpretation of the permitted behavior. Its relevance, enforcement and applicability to the claimed objective require justification.

3.3 What the result establishes for coding agents

A coding agent can use architectural documents, examples and natural-language instructions to propose a correct design, implementation or change. Those inputs supply context for its interpretation. Proposition 1 does not deny that capability. It identifies why the acceptance process must establish the relevant semantics rather than equating a plausible proposal with evidence of conformance.

For software development and evolution with coding agents, the educed semantic foundation supplies the explicit obligations against which initial designs, implementations and subsequent changes are assessed. It makes selected behavioral and architectural commitments inspectable and supports renewed checking through change. Failures can then be explained as specific disagreements about design, behavior, assumptions or implementation correspondence.

The conclusion is narrower and stronger than a claim that software without formal specifications is necessarily unreliable: **an automated semantic guarantee needs an adequate checkable basis, and its scope cannot exceed that basis.** This is the defensible meaning of Claim A.

3.4 Document-centered specification workflows

Spec-driven development (SDD) places an explicit specification upstream of design and implementation. In its document-centered forms, a corpus can contain business or product requirements, architectural descriptions, implementation plans and tasks. GitHub Spec Kit describes requirements and technical plans as inputs to generation, with maintained specifications guiding evolution. The term identifies a development orientation; it does not by itself specify a formal semantics or an assurance level. [28]

There is now substantial public artifact evidence for this practice. The SpecMine preprint reports 470,795 specification files across 73,030 public repositories in its broad collection, alongside a

separate collection of Kiro requirements, design and task artifacts. Its authors explicitly identify tutorials, template copies and one-off projects among the collected repositories. These results document activity; they do not measure industry-wide production adoption or protection against semantic drift. [27]

Knowledge-management tools address the availability and organization of this context. Notion documents wiki ownership, page verification and custom agents that use workspace documents and support knowledge-maintenance workflows. Andrej Karpathy's LLM Wiki describes an agent-maintained Markdown knowledge base that synthesizes sources, updates related pages and flags contradictions, with agent instructions and optional search tools supporting its operation. These are examples of information management capabilities, not empirical studies of software conformance. [29], [29], [30]

Document structure, retrieval and custom plugins can make the relevant material accessible to an agent. Their contribution to semantic assurance depends on the evidence and checks they actually supply. SDD can include such checks: Spec Kit's contract-development guide covers behavioral obligations, provider and consumer verification, negative cases, integration validation, contract versions and compatible rollout and rollback. The relevant comparison is therefore between assurance capabilities, rather than between methodology labels. [28]

3.5 Access to a specification and justified interpretation

Natural-language specifications communicate requirements through expressions whose interpretation depends on linguistic and domain context. Gervasi and colleagues distinguish ambiguity, abstraction and absent information, and explain that even formal expressions can be misunderstood or inadequately connected to real-world entities. Clear syntax and a well-organized corpus do not settle every consequential question of meaning. [31]

Fischbach and colleagues provide direct empirical evidence. In a survey of 104 requirements practitioners interpreting 12 conditional statements, readers disagreed about whether a condition was sufficient or also necessary, and about the temporal relationship between events. Disagreement persisted in familiar-domain statements. The sample was largely German-based and non-native English-speaking; it demonstrates differing interpretations in the studied setting, not a universal ambiguity rate. [32]

Language-model research identifies related interpretation failures. Liu and colleagues' AmbiEnt study evaluated ambiguity recognition and disambiguation using 1,645 annotated examples; the evaluated models struggled to represent alternative meanings. Laban and colleagues studied more than 200,000 simulated conversations across six tasks, including code generation, and found premature assumptions and reliance on earlier incorrect answers when requirements were supplied incrementally. These studies establish failures in their tested models and settings. They do not measure long-term drift in maintained specification repositories or establish that every current coding agent behaves identically. [33], [34]

The resulting engineering concern is a specific failure path: an unsupported interpretation can enter an implementation plan, its generated tests and subsequent documentation. Agreement among those artifacts would then reflect their shared premise, without independently establishing its legitimacy. This is a consequence of how the artifacts depend on one another, consistent with the oracle problem in Section 3.2; it is not a measured propagation rate from those studies. The evidence supports addressing this failure path, rather than claiming that language models are inherently optimized to introduce drift.

Retrieval and semantic justification consequently answer different questions. Retrieving the intended paragraph establishes access. Establishing which behaviors that paragraph permits

requires interpretation; establishing that production preserves those behaviors requires correspondence. Better context can supply missing evidence and resolve an ambiguity. Where conflicting interpretations remain compatible with all available evidence, organization or repeated generation alone cannot establish which is intended, as Proposition 2 formalizes. Controlled natural language can also supply precise semantics, as the FRETish work discussed in Section 7.6 demonstrates.

3.6 What natural language can establish

Research supports precise computational statements expressed in carefully defined subsets of natural language. Kuhn's survey distinguishes controlled languages intended to improve communication from those with semantics as precise as formal logic. A restricted vocabulary or writing template alone does not establish which kind has been obtained. The relevant language definition and interpretation rules matter. [37]

There is also empirical evidence for understandability. In a study with 64 students, Kuhn found that Attempto Controlled English representations of logical statements were understood better than a simplified Manchester OWL notation. Participants judged statements against explicitly depicted situations after training. This supports a natural-language interface to formal meaning in the tested setting; it does not establish faithful capture of a complex system's undocumented objectives. [38]

Three claims must therefore remain separate: readers understand an expression; the expression has a defined machine interpretation; that interpretation faithfully captures the applicable intent. The FRETish translation proof addresses the second claim for its analyzed language. The third remains the adequacy obligation in Section 2.3, even when the first two hold. This paper neither assumes that ordinary prose is always ambiguous nor grants it a general guarantee of unambiguous, complete computational meaning. Its scoped assurance claim requires the relevant interpretation to be fixed, justified and checkable.

An intended contract can deliberately permit several implementations. That freedom is compatible with precise semantics. The contract-reconstruction problem is uncertainty about which obligations apply, rather than the mere existence of multiple correct designs. This preserves the useful abstraction identified in requirements research. [31]

3.7 Why rigorous, machine-readable documentation is insufficient by itself

The strongest objection grants excellent documentation: approved requirements, a controlled vocabulary, explicit exceptions, architectural diagrams, examples, traceability, structured fields, consistent maintenance and reliable retrieval. Assume additionally that the relevant intent is completely and unambiguously stated. **Even under that assumption, the quality of the description does not establish conformance of a separately produced implementation.** This is a different failure boundary from linguistic ambiguity.

Let $Q(D)$ mean that corpus D satisfies its documentation-quality conditions. Let $C(D, I)$ mean that implementation I conforms to the obligations expressed by D. A guarantee concerns $C(D, I)$, not $Q(D)$. If the same valid document can accompany both a conforming implementation and a nonconforming one, a document-quality check cannot distinguish them.

Documentation corollary. Suppose $Q(D)$ holds and an implementation I_{bad} violates $C(D, I_{\text{bad}})$. Then $Q(D)$ alone cannot imply $C(D, I)$ for every candidate I. Changing the encoding of D without supplying additional semantic or implementation evidence does not change this result.

Proof. Instantiating the proposed implication with D and I_{bad} would establish $C(D, I_{\text{bad}})$, contradicting the premise. For a deterministic re-encoding h, equal available evidence remains equal after applying h. Reformatting therefore cannot separate the indistinguishable cases in Proposition 1 or 2. This does not deny that better organization can make existing evidence easier to use; it denies that organization supplies a missing justification.

For a constructed counterexample, assume a specification precisely requires that each payment instruction produce at most one provider charge, including across retries and crashes. One implementation durably preserves the identity that the provider uses to deduplicate the instruction. Another replaces it with a fresh identifier after recovery. Under a provider contract permitting distinct identifiers to create distinct charges, the latter can charge twice. The requirement can remain perfectly clear, correctly indexed and schema-valid throughout. The fault lies in the implementation's failure to preserve the required identity. The documentation's validity cannot rule it out.

Machine-readable structure changes this conclusion only when it supplies a relevant additional capability. A field containing the sentence about payment identity makes the sentence addressable. A defined predicate over payment histories makes its interpretation checkable. Evidence that every admitted implementation satisfies that predicate establishes correspondence at the evidence's stated scope. These are distinct contributions. A graph of document links is not automatically a graph of actual service dependencies; a diagram of intended dependencies is not automatically evidence that forbidden dependencies are absent.

Independent research demonstrates the practical importance of this distinction. In their AWS experience report, Newcombe and colleagues identify missing concurrency and partial-failure details in conventional prose, diagrams and pseudocode. For DynamoDB, extensive testing and detailed written correctness arguments preceded TLA+ checking that found a possible data-loss sequence. The report also records a liveness bug missed where liveness had not been checked. This is industrial evidence for explicit semantics and checking at the required scope; it is not a controlled comparison of every possible documentation discipline or a study of coding agents. [39]

Architectural conformance research supplies the corresponding structural argument. Murphy, Notkin and Sullivan's reflexion models connect an engineer's intended high-level model to an extracted source model through an explicit mapping, then expose agreements and divergences. The mapping and comparison perform work that possession of the high-level diagram alone does not. Their reported applications include NetBSD and Microsoft Excel. [40]

There is important positive counterevidence to a broader claim that documentation can never provide this capability. Kuhn and Bergel used controlled English, a source-code model and automated reasoning to check architectural statements against implementation. Their Mondrian case identified about ten improper component dependencies and detected a newly introduced dependency that contradicted an existing architectural rule. The documents worked as checkable constraints because their interpretation and connection to code were implemented. The result concerns modeled source structure, not complete business semantics or all runtime behaviors. [41]

Consequently, this paper supports a specific impossibility: **where the available evidence and admission mechanism do not distinguish a prohibited change from an allowed one, neither documentary rigor nor a different document encoding can supply a guaranteed distinguishing decision.** It does not support the universal proposition that prose and diagrams can never express intent, or that a document-centered workflow can never preserve software correctly. A verified generator can establish conformance by construction; an English-like specification can have formal semantics; a diagram can denote an executable model. Each must still establish adequacy, correspondence and the applicable assumptions. An agent's

assurance that it followed the documents, without justified evidence for that claim, does not discharge those obligations.

This distinction fixes the paper's research claim. The necessary capability is an adequate semantic contract with a maintained, justified connection to the system being changed. The proposed method establishes that capability where it is missing. The claim applies equally to a Lean specification: a proved model disconnected from production code cannot guarantee production conformance.

4 Why intended semantics is not generally identifiable from partial evidence

4.1 Describing behavior and establishing its legitimacy

An exact account of implemented behavior answers what the program does under its modeled conditions. A contract for development and evolution states which behavior must be realized or remain valid through change. These accounts overlap, but they are not identical. Existing behavior may include defects, obsolete policies and accidental restrictions. Conversely, an unimplemented requirement can still be an applicable obligation.

The distinction is established in reverse-engineering research. Chikofsky and Cross describe reverse engineering as recovering components, relationships and representations at useful abstraction levels. Their account of design recovery incorporates domain knowledge and external information to reconstruct abstractions beyond those obtained from direct inspection alone. [3]

4.2 The contract-reconstruction problem

Let w describe a possible situation containing a domain or business process, its environment, its intended requirements and an implementation where one exists. Let E(w) be the evidence supplied to a reconstruction procedure. Let R_w be the intended contract in that situation. Different situations can produce identical documents, reports, worksheet formulas, code or recorded observations while having different intended requirements.

Proposition 2. Intended-contract reconstruction is underdetermined when observationally identical situations require different contracts. Suppose $E(w_1) = E(w_2)$, and there is a behavior t for which $R_{w_1}(t)$ differs from $R_{w_2}(t)$. No reconstruction procedure F using only E can guarantee an output contract that correctly classifies t in both situations.

Proof. F receives the same evidence in both situations and must return the same result. A contract that permits t is wrong in the situation that forbids it; a contract that forbids t is wrong in the situation that permits it. An output that records the ambiguity may be correct as a partial result, but it has not established the required distinguishing contract. Randomization cannot make the same evidence determine different answers with certainty. This establishes the claim.

The proposition does not depend on the size of the codebase or the intelligence of the reconstruction procedure. More computation can expose consequences of existing evidence. It cannot distinguish situations that are identical with respect to every input available to that procedure.

4.3 A concrete ambiguity

Suppose a system permits several successful payment captures against one order. In one business context this is a defect: one order must be charged once. In another it is required: separate shipments permit distinct partial captures. The source code and recorded executions can be identical. The intended policy differs.

A miner can correctly reconstruct the repeated-capture behavior. A theorem prover can establish that the implementation permits it. Neither result identifies which business interpretation is authorized. Relevant product terms, existing requirements, provider contracts or an explicit policy decision can supply the missing distinction.

An architectural document may already resolve the ambiguity. In that case it is useful evidence, and automatic formalization may be feasible. Its applicability, interpretation and correspondence to the system still need to be established. The paper therefore does not treat documentation as absent merely because it targets people.

4.4 Expert-led discovery and reconstruction

Expert-led discovery reconstructs how the relevant business and technical environment produces its outcomes. Practitioners examine code and configuration where present, documents, data, reports, worksheets, operating practices and subject-matter knowledge. They follow consequential relationships across business domains, reconcile meanings and identify the mechanisms, exceptions and historical decisions that explain observed behavior. The contribution includes finding the sources and relationships needed to formulate a justified eduction task.

A controller's recurring worksheet adjustment illustrates the issue. The same formula can implement an applicable accounting policy or compensate for a defective upstream calculation. The formula alone does not identify which situation holds. Investigation of the calculation, reporting process, operating history and policy supplies the distinguishing evidence. Classifying the adjustment determines whether its effect must be retained, its mechanism replaced or its defect removed. This is a constructed example of Proposition 2.

Human judgment is exercised through this examination and the resulting explanation. Unsupported recollection or professional standing does not establish an external fact. When investigation leaves a genuine policy choice unresolved, an authorized prospective decision establishes the future obligation and is distinguished from historical reconstruction.

Supported programs with adequate contracts, protocols with suitable learning interfaces and standard algorithms with authoritative specifications can permit highly automated analysis. These cases already supply important parts of the semantic path. For cross-domain eduction from latent knowledge, the proposed method requires practitioners to establish and justify that path, with tools assisting discovery and carrying out supported analyses. Section 6.1 gives its substantive form as an eduction map.

4.5 Useful assurance under remaining uncertainty

Eduction can establish the complete intended contract within a defined scope, the particular obligations needed for a change, or the acceptability of that change under every remaining supported interpretation. These are different conclusions. Proposition 2 excludes guaranteed exact reconstruction under its premise; it does not require every unrelated uncertainty to be resolved before justified engineering can proceed.

Let $K(E)$ be a nonempty collection of candidate contracts compatible with evidence E, expressed

over a common interpretation of the relevant complete behaviors. Identities, effects, time and operating conditions must have compatible meanings across the candidates before their permitted behaviors can be compared. For contracts represented as sets of permitted complete behaviors, a conservative envelope is their intersection:

$$C_{\text{robust}} = \bigcap_{C \in K(E)} C.$$

The guarantee concerning actual intent is conditional on the actual intended contract remaining in $K(E)$. An omitted interpretation receives no protection from this construction. If contradictory evidence leaves $K(E)$ empty, the construction is inapplicable: the set-theoretic convention that an empty intersection yields the ambient universe must not become permission for arbitrary behavior. The contradiction requires resolution before this reasoning can justify a change.

A candidate whose relevant behaviors lie in that intersection satisfies the shared behavioral restrictions. This is useful only if the candidate also retains the required capabilities of every applicable interpretation and satisfies the consistency and realizability conditions in Section 2.4. The intersection may be empty or demand incompatible progress; it is not automatically an implementable specification. Removing an optional behavior can be a valid refinement, while removing a required ability to serve a legitimate request is a contractual loss.

For example, uncertainty over whether an order permits one or several capture instructions need not prevent a change that preserves the same accepted instructions and their existing effect identities under either policy. Introducing additional captures does depend on that unresolved distinction. The admissible conclusion is consequently local to the obligation and change, while the original policy uncertainty remains explicit. This is a derivation from the possible-contract account, not evidence that a tool can enumerate all such contracts for an arbitrary system.

5 What automated specification mining can establish

The following approaches contribute candidates, interpretations and verification evidence to eduction. Source analysis requires source code, and execution-based mining requires suitable observations or access to behavior. Those conditions apply to the particular technique; the wider method can also reconstruct meaning from business artifacts and process knowledge. An existing application expands the analysis routes without defining the method's scope.

5.1 Dynamic invariant detection

Dynamic mining observes executions and proposes properties consistent with them. Daikon, for example, discovers likely invariants involving values, relationships and collections. Its outputs are useful candidates for understanding and verification, but observed consistency alone does not establish validity on unobserved behavior. [4]

For eduction, the important questions are which states were observable, which executions occurred and which properties the miner could express. A rare recovery path absent from the evidence cannot be protected merely because the mined invariants fit common executions. Candidates can be strengthened or rejected through independent evidence and static verification.

5.2 Static inference and compositional analysis

Static inference reasons from program structure and a modeled language semantics. Infer's account of separation logic and bi-abduction shows how preconditions, postconditions and local memory reasoning can be inferred and composed. This is substantial automation of specification-related work, including cases where programmers have not supplied those contracts. [5]

An inferred contract has the scope of the analysis and its assumptions. A memory-safety result, for example, does not establish the meaning of a payment or the legitimacy of a business rule. Abstract interpretation can soundly approximate supported behaviors while losing precision. Solver failure or an unproved obligation is not automatically evidence of a real defect.

5.3 Behavioral learning and temporal mining

State-machine and temporal miners infer patterns of interaction. Neider and Roy survey LTL mining through constraint solving, enumeration and learning techniques, including the role of examples, templates and other supplied information. A specification consistent with the supplied examples is not thereby a complete account of intended system behavior. [6]

Active learning can obtain additional observations by querying a system. Its useful guarantees depend on its model class and on how membership and equivalence questions are answered. Practical conformance tests approximate an ideal equivalence oracle. Abstraction, reset behavior, observation fidelity and environmental nondeterminism affect what has actually been learned.

Schuts, Hooman and Vaandrager report learning behavioral models during the refactoring of a Philips legacy component and comparing them with an equivalence checker. The process found defects missed by the existing regression tests, including a defect in the old implementation. The final learned models were equivalent for the selected interface; the paper also discusses learning cost and abstraction. This is direct evidence that automation can support preservation in legacy engineering, rather than evidence of automatic reconstruction of an entire enterprise contract. [7]

5.4 Process mining

Process discovery uses recorded events to infer workflow models; conformance analysis compares recorded behavior with a model. Van der Aalst explains how log and model abstractions connect concrete observations to represented behavior. This is relevant to operations spanning several applications. [8]

The inferred account depends on event identity, correlation and abstraction. Two logs containing the same activity names can conceal different business meanings or missing events. Workflow conformance therefore contributes evidence at its observed level; it does not automatically establish the semantics of all data or the validity of every permitted transition.

5.5 Semantic extraction and formalization

Semantic extraction translates supported source programs into a representation suitable for formal reasoning. It can provide a valuable implementation connection, but it does not independently decide which implemented behaviors should be required.

The September 2026 Aeneas report describes extracting Rust through Charon and Aeneas into Lean, while formalizing cryptographic standards separately and proving conformance. It also identifies trusted translations, library models and specification fidelity as parts of its assurance

argument. This is an especially relevant example of useful automation combined with independently justified semantics. [9]

5.6 LLM-assisted synthesis

The 2026 VeriSpecGen preprint generates Lean specifications from natural-language requirements using requirement-level attribution and repair. It reports 86.6 percent pass@1 on VERINA SpecGen using Claude Opus 4.5. The evaluated pipeline permits up to 25 refinement iterations, so pass@1 here is not one-shot generation accuracy. The result concerns the supplied benchmark tasks, rather than the proportion of an enterprise's hidden requirements identified. [10]

The method proposed here accommodates improvements in such tools. Automatically produced specifications are candidate interpretations with evidence obligations. Automation can reduce formalization effort without eliminating the need to establish that the formalized requirement is applicable, sufficiently complete for the claim and faithfully related to production behavior.

5.7 Evidence for human-guided abstraction

Mashhadi and colleagues studied interactive mining in an industrial autopilot system with more than 500,000 lines of C/C++. Practitioners selected relevant state fields, functions and constraints; those selections established the perspective and abstraction from which the tool generated focused behavioral models. Eight technical staff participated: six software developers, a team leader and a control engineer, using issues drawn from a pool of 17 real bugs. The study supported feasibility and usefulness for debugging and reported difficulties with the evaluated fully automated alternatives. Its qualitative, single-system setting does not establish comprehensive contract reconstruction or general human superiority. The authors explicitly leave open the possibility of suitable fully automated approaches. [11]

5.8 Business rule extraction in legacy software

Sneed and Verhoef describe rule extraction from a 6.4-million-line COBOL/IMS/DB2 leasing system. Users selected business-critical output variables that became starting points for automated data slicing. Analysis followed their input data, conditions and control paths. Developers familiar with the code identified obsolete variables and rules left by earlier date and currency conversions. Expert selection therefore established the route of extraction and helped interpret its results. [43]

For semantic eduction, the relevant mechanism is the connection between a business result and the implementation paths that produce it, followed by a determination of which extracted rules remain applicable. The report documents such work in a large industrial system; it leaves the usefulness of the extracted rules incompletely assessed. It is evidence of an engineering activity and its implementation, not a comparison with present-day coding agents or a proof that semantic selection must always be manual.

Together these results support selective automation with explicit applicability conditions. The unresolved unit is the particular semantic obligation and the evidence required to establish it. System size alone does not determine the appropriate division of work.

6 Semantic eduction for development and evolution with coding agents

6.1 Definition and responsibility

Semantic eduction is the expert-led examination and reconstruction of latent domain and process knowledge into a justified semantic foundation for developing and evolving software with coding agents. Practitioners comprehensively examine the relevant business and technical environment, construct a justified eduction map, and use it to guide machine analysis, formalisation and verification. They establish which meanings and relationships must be preserved, which structures are incidental or defective, and which conclusions remain unsupported. No existing target application is required. The resulting foundation guides initial agent work and, once connected to the actual implementation, enables checking of that implementation and subsequent changes.

An eduction map is a substantive account of the path from distributed evidence to executable semantics within a defined scope. It connects:

- **Domain phenomena and sources:** the consequential outcomes, concepts and effects, and the code, configuration, documents, data, reports, formulas, practices and subject-matter knowledge that explain them.
- **Mechanisms and dependencies:** the causal, data, control and decision relationships across domains, including manual interventions, external commitments and failure paths where relevant.
- **Interpretations and abstractions:** the meanings, identities, units, exceptions and distinctions needed by the covered obligations, together with the reasons for retaining or changing observed structures.
- **The path of eduction:** the evidence to examine together, the supported analyses and representations, and the reasoning needed to establish the adequacy and implementation connection of the resulting semantics.

A map is justified at its stated strength when its consequential relationships and premises have supporting evidence, its interpretations distinguish applicable obligations from observed behavior, and its abstractions preserve the distinctions needed by the claim. Dependencies are examined directly or represented through applicable, justified boundary contracts. An unsupported dependency limits the corresponding conclusion. Comprehensive examination follows these dependencies across the relevant environment; repository ownership and the convenience of a local proof do not determine its scope. Section 6.4 explains the examination, and Section 6.5 states the sufficiency criterion.

Practitioners perform the discovery and integrative judgment that establish this map. Automated search, slicing, inference and model construction contribute evidence and candidate interpretations. The map gives those tools a justified semantic task and a path through the relevant sources. Machine-discovered dependencies or counterexamples can extend or correct it; its justification develops with the investigation and the five activities in Section 6.3.

Corollary. A map constructed solely from ambiguous evidence cannot establish the missing intended distinction. Let G construct a map from the complete evidence available to a procedure, and let H derive a behavioral classification from that map. If $E(w_1) = E(w_2)$, then:

$$H(G(E(w_1)))(t) = H(G(E(w_2)))(t).$$

If the intended classifications $R_{w_1}(t)$ and $R_{w_2}(t)$ differ, this common result cannot be correct in both situations. **Proof.** Apply Proposition 2 to the composed procedure $F = H \circ G$. The intermediate map adds no distinguishing information. The companion Lean theorem noMapResolvesAmbiguousEvidence checks this deterministic implication.

Consequently, a justified path must supply evidence for each consequential intended distinction or identify the authorized decision establishing a future obligation. Where neither resolves the distinction, Section 4.5 describes the narrower conclusions that remain available. Formalisation and successful checking of a selected interpretation establish its consequences; they do not establish that the interpretation represents the applicable intent.

Specification mining within this method discovers, forms and assesses candidate rules from evidence, including implementation where present. Eduction additionally reconstructs meanings and dependencies across business and technical sources and justifies the map and the resulting contract. A new application also requires prospective design choices, which are distinguished from educed domain knowledge.

Conditions of applicability. Eduction requires access to evidence sufficient to assess the covered obligations and their consequential dependencies, together with the domain, stack, modeling and verification competence needed to establish the semantic connections. That competence may be distributed across the practitioners applying the method. Appendix B develops these practical requirements. Responsibility identifies who performs and integrates the work; supporting evidence establishes the justification.

6.2 Evidence and inference

Let $W(E)$ be the set of situations compatible with the evidence E and the assumptions justified so far. A factual claim derived from the evidence is warranted at the stated strength only if the evidence supports it; a logically entailed claim holds throughout $W(E)$. In practical work, evidence can also support qualified probabilistic or provisional conclusions, which must retain that weaker status.

The eduction process distinguishes four operations:

- **Deduction:** derive a consequence of existing evidence and accepted assumptions. This reveals implications without introducing new empirical facts.
- **Hypothesis formation:** propose a candidate rule, interpretation or abstraction from observed regularities and other evidence. Generalizing from finite executions introduces an inference whose validity must be assessed; consistency with those executions alone does not prove the candidate.
- **Evidence acquisition:** inspect an additional source, dependency, report, worksheet, existing record or account of established practice that can distinguish previously compatible situations. Subject-matter knowledge is assessed for the claim it supports, including its relationship to the artifacts and process.
- **Authorized determination:** establish the future requirement where legitimate choices

remain. This changes the accepted contract rather than proving which choice was historically intended.

Automated hypothesis formation can be connected to mechanical verification. In ICE, a learner proposes invariants while a verifier supplies examples, counterexamples and implication constraints; the target program and property are already given. This establishes an important automation route for proof support. A proved inductive invariant and an inferred business obligation have different roles: generating the former does not determine an otherwise unspecified business objective. [44]

If evidence is contradictory, indiscriminately treating every statement as an axiom permits meaningless conclusions. Eduction must resolve applicability, version, interpretation or credibility at the disputed claim. If the evidence cannot resolve it, the corresponding conclusion remains limited. Formal notation does not repair an unsound premise.

6.3 Five activities that establish a usable semantic contract

The following activities construct and refine the eduction map and the semantic foundation needed to guide coding agents and assess drift in their initial designs, implementations and subsequent changes. Practitioners integrate the cross-domain examination and justify the meanings, mechanisms and abstractions on which machine work depends. Analysis and verification tools contribute within the supported tasks. The activities apply the requirements, mining and refinement foundations to this engineering purpose. Scope follows the obligations the application must realize or preserve and the required assurance strength; repository and team boundaries do not exclude consequential dependencies.

The activities are iterative: evidence discovered while tracing mechanisms or establishing correspondence can reopen the selected obligations, interpretation, assumptions or specification. Reconsidering an interpretation does not itself authorize changing the applicable policy.

1. Establish which behaviors and design commitments must be realized, preserved or deliberately changed. Relate domain practice, reports, formulas, data and documented intent, together with implemented behavior where present, to applicable product terms, requirements, architectural decisions and historical evidence. Surface implicit obligations and distinguish them from defects and incidental choices. Identify what must be true and which design and implementation choices the agent may make. Existing adequate specifications can supply these distinctions. For a new application, establish the domain obligations and make prospective design choices explicit. Where evidence leaves a genuine policy choice, the enterprise supplies or delegates the relevant decision.

The result is an explicit set of applicable obligations R within the defined scope, including accepted architectural commitments relevant to the agent's work. This classification is essential because copying current software or worksheet behavior into a contract can protect a defect, while removing it can destroy a legitimate capability. Multiple captures against one order illustrate the distinction: they may be an error or required partial-shipment behavior.

AI can extract rules and compare relevant records. Where those records do not distinguish incompatible interpretations, the same information limit applies to practitioners and automated tools. Additional evidence or an authorized prospective decision supplies the distinction; confidence in an inferred intention does not. If exact preservation of selected existing behavior is itself the accepted objective, that behavior can serve as a reference without separately deriving every property it entails.

2. Connect business concepts to domain and system identities, states and events.

Establish how instructions, records, worksheet entries, reports, process actions and external effects represent the concepts used by the contract. Establish their identities, units, timing and dependencies. Where software exists, follow these relationships across serialization, persistence and service boundaries. For a first application, define the intended representation of those concepts and subsequently establish its relationship to the actual implementation. The result is a supported domain interpretation and, for implementation assurance, a justified abstraction α.

Architectural concepts also require a supported interpretation. A component responsibility, transaction boundary or owner of an external effect must be connected to intended interfaces, state ownership and effect paths. A dependency rule becomes checkable on an implementation when its relevant source or runtime elements and their permitted relationships are identified. For a new application these are prospective commitments until realized; the checking mechanism must then establish that the actual implementation respects them. This connects the design supplied to the agent with the implementation the mechanism examines.

This connection is essential because a precise formula can protect the wrong distinction. A condition permitting at most one capture per provider key allows two charges for one business instruction if every retry receives a fresh key. The specification must relate provider effects to the accepted logical instruction whose preservation is required.

AI can discover candidate correspondences and derive relationships from documents, formulas, code and data definitions. Their validity depends on evidence of the actual relationships and applicable domain meaning. A suggestive field name, worksheet label or familiar design pattern is a hypothesis until that connection is established. Human interpretations are subject to the same evidential requirement.

3. Establish the mechanisms and premises on which the obligation depends. Trace how relevant information, decisions and effects arise in the business process, including formula dependencies, manual adjustments, approvals, timing and external commitments. Where software exists, trace the relevant state, data, control and effects through code, configuration, persistence, queues, concurrent handlers and recovery paths. Determine what the process and application control, what other parties control, and which environmental guarantees the claim requires. Static analysis, slicing, learned models and existing operating records contribute according to their supported scope. Before implementation, establish which observations and controls a realizable design must possess.

The result is an account of relevant operational mechanisms and supporting premises A, with an account of implemented behavior where available. It distinguishes recorded practice, observed executions, reachable behaviors and behaviors admitted by an approximation. This is essential because a contract assuming permanent provider deduplication cannot justify retries after the provider's actual protection expires. A model omitting a recovery job cannot establish that job's behavior; a financial model omitting a controller's adjustment can misstate the reported result. A proposed technical mechanism requires justification of its realizability and later evidence of its actual operation.

AI can trace dependencies, retrieve contracts and analyze failure paths. It cannot establish an unavailable provider guarantee by deduction from unrelated implementation evidence. A practitioner must likewise find support, identify a realizable alternative or limit the assurance claim. Knowing where the effect occurs and what controls it determines which evidence can answer the question.

4. Construct the specification and justify its adequacy. Select the abstractions and representation needed to express the accepted invariants, permitted transitions, preconditions, postconditions, structural constraints and progress obligations. Explain why omitted distinctions do not invalidate the covered requirement. Assess whether the specification excludes pro-

hibited behavior, retains required capabilities and requires promised progress under its stated assumptions, including applicable failures and exceptions.

For a design obligation, S can express permitted dependencies, responsibility boundaries, state ownership or interface behavior in a supported structural or behavioral representation. The proposed design must preserve these commitments outside the authorized change. For an implementation obligation, S specifies the states, transitions and effects the code must realize. The established interpretation connects the two levels.

The result is S together with evidence that its meaning and premises support R at the stated assurance strength. This is essential because a condition excluding duplicate charges can be satisfied by never charging anyone. Similarly, an abstraction that discards crash state can conceal precisely the execution the contract needs to exclude. Concrete allowed and prohibited cases help expose these errors; a finite selection alone does not establish universal adequacy. Contradictory premises and impossible preconditions also require examination.

AI can generate formal statements, find counterexamples, synthesize invariants and construct proofs. The required distinction is between demonstrating a logical consequence and justifying the premises and interpretation to which it is applied. A successful proof that assumes a business interpretation cannot, by that success alone, establish that the interpretation is applicable. The eduction account supplies the connection, irrespective of who or what performed the reasoning.

5. Connect the contract to agent implementations and maintain the checking capability. Establish correspondence for the first implementation as well as later changes. Connect relevant source operations and dependencies to S through a supported semantic translation, refinement, source-level verification or explicitly bounded checking. Identify trusted models, extraction assumptions, build conditions and unsupported implementation boundaries. Architectural claims additionally require the appropriate connection to source structure or runtime topology. Assess an initial or revised design against the represented obligations, and assess its realization against the affected behavioral and architectural obligations.

The result is correspondence evidence within J and a means of checking the affected design and implementation relationships after subsequent agent changes. This fifth activity makes the educed foundation usable for drift mitigation from the first application onward: a justified specification can guide generation while a generated implementation still violates it, and an unchanged model can continue to pass after production diverges. Generating a specification and establishing its implementation relevance are separate achievements.

AI and verification tools can automate correspondence proofs where the required models and tool support exist. The evidence must concern the implementation being admitted and all effect paths relevant to the covered claim. Changes to configuration, dependencies, migrations or recovery paths can reopen the obligation even when the formal requirement is unchanged. Section 8 develops this continuity requirement.

6.4 Making eduction judgments inspectable

The eduction map is constructed through examination of consequential obligations and the domain or system phenomena on which they depend. Scope follows the intended outcome and the consequences of getting it wrong. A financial-reporting claim includes the formulas and manual adjustments that determine its figures; a payment-preservation claim includes the paths that can create financial effects, even if the easiest proof concerns only the request handler. Existing reports, process accounts, contracts, incident histories, architecture and effect interfaces help locate those consequences. Ease of formalization does not determine which obligations the claim covers.

Resolve evidence at the disputed claim. Product authority establishes which policy applies; code and configuration establish what particular mechanisms implement; operating records establish the events they actually record. Their applicability depends on version, domain and coverage. An approved rule permitting partial captures and code permitting them corroborate different propositions. A log showing two captures does not decide whether they were legitimate. Where records disagree, the eduction explanation identifies the incompatible propositions, the evidence resolving them and any remaining policy choice. No source category has universal precedence over every kind of claim.

Follow the consequential paths in both directions. From each consequential report figure or business effect, trace the formulas, source records, decisions and adjustments that determine it; from each contributing input or intervention, trace the results it can affect. Where software implements the process, follow accepted inputs through persistence, dispatch, concurrency, retries and recovery to their effects. From each effect-producing interface, trace every discovered caller back to its authorization and identity, including scheduled jobs, administrative operations and migrations. Compare those accounts with schemas, deployment configuration and existing operational evidence. A missing path in an analysis with unsupported dynamic dispatch remains a coverage limitation; a plausible diagram cannot close it. The purpose is to establish the dependency boundary of the obligation, not to read every file indiscriminately.

Justify the abstraction through the distinction it preserves. Explain which concrete events become accepted instructions, state changes or effects, and why their identities and units have the required meaning. Challenge a proposed mapping with an acceptable and a prohibited execution. If both map to the same abstract behavior, admitting that behavior would violate the property-transfer obligation in Section 2.3. Rejecting it can remain sound while also excluding some acceptable executions. Refinement is necessary where that loss prevents a required capability; otherwise conservative exclusion can be sufficient. Examples expose mistakes, while the general criterion remains that the abstract property entails the concrete requirement under the stated assumptions.

Make the decisive inference explicit. Connect each selected obligation to the evidence that makes it applicable, each premise to its supporting contract, mechanism or process, and each implementation claim to the checks that address it. Identify conclusions that share the same upstream interpretation: generated requirements, code and tests can all repeat one mistake. Several such artifacts supply no independent confirmation of that premise. Section 9 develops these relationships through payment and Excel examples.

Uncertainty is material when a supported alternative changes whether an admitted effect is permitted, a promised request can be served, or a required check establishes its claim. That conclusion remains unresolved at the requested strength unless further evidence resolves it or the change is justified under all remaining interpretations as in Section 4.5. A difference confined to legitimate implementation freedom need not block the change. This criterion directs the remaining work by consequence rather than by a demand to eliminate every uncertainty.

6.5 Completion of an eduction claim

For a particular obligation r, the semantic basis is sufficient to guide initial agent work when the eduction map justifies its meaning, applicable authority, consequential dependencies and premises, the chosen representation captures it at the claimed scope, and its consistency, required capabilities and realizability have been justified as in Section 2.4. It specifies the intended design and implementation connections and how they will be assessed. A model can therefore be a completed eduction result for that use before target code exists.

Assurance about an implementation additionally requires the actual design or implementation

connection relevant to r, together with evidence at the claimed strength and a means of checking affected changes. This applies to admission of the first implementation and every subsequent one. Each activity can reuse a justified existing result. The foundation consists of substantive obligations, interpretations, assumptions, specifications and evidence; completion is relative to its stated use, not established by a practitioner's title, a generated artifact's formality or a count of extracted rules.

A bounded result can be complete for a bounded objective. It is not complete for a broader obligation that depends on an unresolved provider guarantee or an unexamined failure path. This criterion prevents both indefinite exploration of the whole estate and premature claims based on a convenient local model.

The conclusion must preserve its evidential strength. Evidence can establish the covered claim, establish only a narrower claim, contradict the claim, or leave it unresolved. A concrete counterexample contradicts the property at its applicable scope; a solver timeout or unsupported language feature leaves an obligation unproved. Neither a weaker result nor an unresolved one constitutes successful admission at the original strength. These are meanings of the evidence, not a required record format or a substitute for the reasoning itself.

7 Artifact forms and target technologies

7.1 A foundation can have several useful forms

The artifact of eduction is a justified semantic foundation usable by coding agents and the mechanisms assessing their work. Its eduction map supplies the substantive account connecting sources, domain phenomena, consequential dependencies and justified abstractions to the path of formalisation. The map develops during examination and remains part of the explanation of why the resulting semantics are appropriate. Human-readable explanations establish vocabulary, rationale, evidence and unresolved limits. Architectural accounts describe relationships and responsibility for effects. Formal specifications make selected design and behavioral obligations precise. Executable models, proofs, static contracts and runtime checks provide different connections to behavior. Explanations and justified models can precede target code and guide its initial construction. Correspondence evidence then connects the actual first implementation and subsequent versions to the foundation, at the strengths those connections establish.

A human-readable architectural specification is a legitimate and valuable part of the foundation supplied to a coding agent. Mechanical assurance against drift additionally needs the relevant architectural or behavioral obligation represented in a form the checking mechanism can evaluate. For example, a dependency rule can be enforced as a structural constraint, while a transaction invariant may require reasoning about state transitions and concurrency.

The artifact is selected by the subject and assurance claim. A set of invariants can be sufficient for one preservation obligation. Another may require a state machine, a relational model, an executable function or a proof connecting several abstractions. Completeness is relative to the stated obligation; file count and language choice are not measures of semantic adequacy.

7.2 Proof assistants

Lean, Rocq and Isabelle support expressive formal models and proofs. They are suitable where the required semantics and implementation relationship justify that investment. Proof checking establishes the formal theorem under its accepted foundations; the interpretation of the theorem remains consequential. Lean's documentation explicitly distinguishes proof validation from the

meaning intended by the user. [13]

Rocq provides formal specifications, proofs and executable extraction. Its ecosystem includes the Verified Software Toolchain, which connects C-level reasoning with CompCert semantics. Isabelle's AutoCorres2 supplies another connection to C verification. These implementation capabilities can be more decisive than a general preference among proof assistants. [14], [15], [16]

Proof assistants are not interchangeable drop-in validators for arbitrary existing source. The supported language subset, imported axioms, external calls and mapping to compiled behavior determine the actual assurance boundary.

7.3 Verification languages

Dafny combines program-like specifications with automated verification of preconditions, post-conditions, invariants, frame conditions and termination obligations. Its executable programs can compile to several production languages. Why3 provides WhyML and connections to automated and interactive provers; WhyML can serve as an implementation or a verification intermediate language. [17], [18]

F* combines expressive types with automated and interactive reasoning and supports extraction through several target toolchains. It is relevant when verified components can become part of the production implementation. [19]

These approaches can reduce the distance between specification and executable implementation. A separately maintained model still needs correspondence evidence for legacy code written elsewhere. Compilation from a verified source and verification of an existing independent source are different integration paths.

7.4 Behavioral and relational models

TLA+ supports reasoning about systems as state transitions and temporal behavior. Its TLC checker explores a chosen model, making it useful for concurrency, retries and failure interactions. Passing such a check establishes the result for the checked model and assumptions; it is not automatically a proof about every production instance. [20]

Alloy supports relational and temporal constraints, useful for identities, permissions, ownership and structural relationships. Analysis remains relative to its selected scope. B and Event-B, supported by tools such as ProB, are also relevant where mathematical state models and refinement fit the problem. [21], [22]

These models often expose semantic defects before implementation verification is attempted. The resulting insight remains useful even where only part of the model can currently be connected to code with strong assurance.

7.5 Contracts close to source

ACSL describes C behavior for tools such as Frama-C. JML supplies Java contracts used by OpenJML. Verus supports specifications and proofs for a supported Rust subset. Such approaches can connect obligations directly to the implementation being changed, avoiding an unnecessary independent replica of its logic. Their practical scope follows tool and library support. [23], [24], [25]

A source-level annotation can still encode the wrong rule. Closeness to implementation strengthens correspondence but does not establish requirement adequacy. An inferred precondition that excludes legitimate requests can make a proof easy while failing the actual contract.

7.6 Executable checks and other formal representations

Property tests, model-based tests, replay checks, schema constraints and runtime monitors can embody useful fragments of semantics. Their role is determined by what they establish. A monitor that detects a bad event after it occurs is observational evidence; a guard that blocks it before the relevant effect has different enforcement semantics. Some progress properties cannot be guaranteed merely by stopping executions. Schneider's characterization of execution-monitoring policies demonstrates why enforcement mechanisms have property-specific limits. [26]

Controlled natural language is another possible target. NASA's FRETish has a defined semantics and translation into temporal logic. Conrad and colleagues proved in PVS the correspondence of the analyzed translation to past-time metric temporal logic over finite traces. Its English-like surface therefore coexists with mechanical meaning. The proof establishes preservation of the defined semantics, not that the author selected the right requirement. The remaining obligations concern the chosen requirement, its domain interpretation and its connection to the implementation. [35]

Graphical modeling is also compatible with defined execution semantics. OMG's fUML specification defines a foundational subset of executable UML models and publishes separate normative machine-readable syntax and semantics models. This is evidence that a diagram-based notation can carry precise meaning; it is not evidence that an arbitrary architecture diagram is executable, or that a separately generated implementation conforms. The relevant choice is the model's actual semantics and supported implementation connection. [42]

7.7 Choosing the target representation

The engineer selects a representation together with its implementation connection and checking method. A language's expressiveness alone does not establish that combination. The following procedure applies the adequacy, correspondence and capability obligations developed above; it is not an empirically validated ranking of tools.

1. Determine what must be established. Identify the covered obligation, legitimate capabilities, operating domain and required assurance strength. A dependency restriction concerns structure; an account invariant concerns state; finality under retries concerns histories and external effects; progress concerns permitted infinite behavior. State whether the claim requires a universal conditional proof, analysis within a selected model, or evidence from exercised or observed cases. This determines which semantics and evidence can satisfy the claim.

2. Establish the route to implementation and its evidence. For existing software, inspect the relevant source languages, libraries, concurrency mechanisms, external calls and build path against the documented scope of candidate tools. For a new application, choose a realizable implementation and verification path together, using the intended mechanisms and constraints, and subsequently check the actual source and build. Determine whether the obligation can connect through source contracts, supported semantic extraction, generation from verified source, structural analysis, or a separate model with justified correspondence. Unsupported code or dependencies require an adequate boundary contract or further correspondence work. Language support in name alone does not establish coverage of the actual features.

3. Narrow the representation by the required reasoning. For obligations expressible

over supported existing source, consider source contracts such as ACSL, JML or Verus. For mathematical models, induction or refinement proofs, consider Lean, Rocq or Isabelle together with an available implementation connection. Verification languages such as Dafny, WhyML or F* fit where the verified program or its justified translation can form the delivered implementation. Temporal models such as TLA+ address interacting transitions and progress; Alloy supplies relational and temporal modeling. Their analysis scope must match the assurance claim. A structural restriction may need only a precise component interpretation and supported dependency extraction. FRETish, executable UML, tests and monitors remain candidates when their defined semantics and checking strength suffice. Sections 7.2 to 7.6 establish these capabilities and their limits.

4. Exclude candidates that cannot meet the obligation. A candidate is unsuitable for the promised result if it omits a consequential distinction, lacks the needed implementation connection, relies on unjustified premises, excludes required capabilities, or supplies weaker evidence than the claim requires. A bounded analysis cannot by itself satisfy an unbounded guarantee. A monitor cannot establish a preventive guarantee merely by observing an irreversible effect afterward. If no candidate meets the required result, the gap remains explicit; selecting an expressive language cannot close it.

5. Choose the most direct maintainable assurance path. Among candidates meeting those conditions, prefer the one with the strongest justified connection to the actual or planned implementation and the least additional translation or duplicated semantic model to maintain. Resolve remaining tradeoffs through supported language features, trusted dependencies, proof automation, available practitioner competence and the expected obligations created by routine changes. Use the inspected system or justified prospective design and established tool capabilities to explain the choice; ease of writing the initial specification is only one cost. Reconsideration is warranted when a changed obligation, implementation feature or dependency invalidates the reasons for selection.

6. Combine representations only where the obligations justify it. A distributed protocol model and source-level contracts can address different parts of one guarantee. Their shared identities, events, units and failure meanings need compatible interpretations; the mappings between them must carry the relevant properties. Component assumptions require the composition reasoning in Section 8.4. Two independently passing models do not establish their agreement or their combined applicability to production.

For example, where an existing C component and its relevant dependencies are covered by Frama-C/WP, an ACSL contract can provide a direct deductive route for a suitable source-level obligation. A detached Lean model would add a separate correspondence obligation unless a justified translation supplied that connection. This favors ACSL for that case, not for every C system or property. Conversely, a requirement about distributed finality needs the external events and interleavings represented; a local function contract alone does not establish it. The selected combination follows those obligations. [50]

8 Mitigating design and implementation drift

The educed foundation supplies an accepted basis for the agent's initial and subsequent design and implementation work. Design proposals are assessed against applicable responsibilities, interfaces, ownership and structural or behavioral constraints. The first implementation and later changes are assessed through the justified connection between those obligations and the actual code and operating behavior. Authorized evolution establishes the revised basis and the obligations retained across versions. The admission result applies to the first implementation

without requiring an earlier codebase.

Mitigation depends on the properties represented, the evidence available and the checking capability actually connected to the changed system. The conditional result below explains when a covered property is preserved. It does not measure the complete method's effectiveness across arbitrary agent changes.

8.1 A conditional assurance result

Let q(t) mean that an abstract behavior satisfies a covered semantic requirement. Let V(B, I, e) mean that the configured verifier accepts implementation I against foundation B using evidence e. Assume that every admitted implementation passes through this boundary, and that accepted verification soundly establishes its correspondence obligation under A.

Proposition 3. Sound admission and adequate correspondence preserve the covered property. Suppose acceptance establishes $\alpha(\mathrm{Beh}_A(I)) \subseteq \mathrm{Beh}(S)$, and every behavior permitted by S satisfies q. Then every admitted implementation's abstract behavior satisfies q under A.

Proof. Take any admitted implementation I and any behavior t in $\mathrm{Beh}_A(I)$. Sound admission and correspondence give α(t) in Beh(S). Specification adequacy gives $q(\alpha(t))$. Since I and t were arbitrary, the covered property holds for every admitted implementation and every behavior within the stated assumptions.

Applying this abstract result to a concrete requirement r also needs the justified property-transfer relationship in Section 2.3: under A, $q(\alpha(t))$ must imply r(t). This is where the established meaning of identities, events and effects enters the claim about the actual system.

A structural design obligation requires its own justified interpretation and conformance evidence. It can be checked directly on a supported source or runtime representation; behavioral preservation establishes it only when the modeled observations include the relevant structural distinctions.

This result requires all admitted implementations to pass through the boundary; it does not require the verifier to prove every correct program. It is a conditional assurance rule, not a guarantee that its premises are automatically obtainable. Its practical value is to locate exactly where a claimed protection can fail: admission, correspondence, adequacy or environmental assumptions.

8.2 The connection must survive change

Checking an unchanged formal model after changing production code establishes no new implementation result. The affected correspondence must be regenerated or re-established. Changes to configuration, dependencies, external protocols, build behavior or data migrations can also invalidate an earlier argument without changing the domain model's theorem statements.

Proof evidence belongs to the implementation and assumptions it actually addresses. A source-level theorem needs a justified connection to the delivered artifact; an artifact-level claim needs the relevant deployment conditions. A passing check cannot cover a production path that bypasses the checked implementation.

The acceptance boundary depends on the whole foundation B and its checking machinery: requirements, specification, assumptions, abstraction, evidence, external-library models, checked scope, verifier configuration and build and deployment conditions. Weakening any of these can turn a passing result into a different claim. Removing an effect from α or excluding a recovery path from analysis can conceal a violation without changing the theorem statement.

Agent delegation consequently specifies the authorized semantic change and the implementation or proof work permitted within it. An agent can propose a change to requirements, assumptions or interpretation, but that proposal cannot justify itself through the passing checks it has altered. The changed basis needs the relevant authority and renewed adequacy and correspondence evidence before admission against it. Proof development against an accepted contract permits deriving lemmas; adding an assumption that excludes the difficult case changes the contract's basis. The same distinction applies to generated tests and changes to the build or verification scope.

The Aeneas report supplies a concrete precedent: verification campaigns separate stages and restrict agents from editing specifications, axioms and implementations during proof development without escalation and explicit human approval. Its workflow combines expert review of formalized standards with automated extraction and independently checked proofs. [9]

8.3 Authorized evolution of the contract

Semantic preservation does not mean freezing all behavior. Let R_v be the requirements for version v, and let an authorized change Δ identify the intended differences. The next contract must retain the applicable obligations outside Δ and establish the new obligations introduced by it. Migration and compatibility conditions are part of that change where relevant.

For the retained obligations P_{keep}, the preservation requirement is:

$$\forall r \in P_{\text{keep}}, \quad A_{v+1} \wedge S_{v+1} \models r.$$

The new implementation must separately satisfy its correspondence obligation to S_{v+1}. Adding functionality need not mean that every new trace is a trace of the old system. Comparison uses the relevant old observations and compatibility promises.

Preservation also concerns the promised operating domain. Let D_{keep} contain the previously supported input classes and environmental conditions retained outside Δ, expressed through a justified common interpretation. The new contract must continue to support them:

$$D_{\text{keep}} \subseteq D_{v+1},$$

with the retained capabilities and obligations preserved.

This domain condition prevents a misleading success in which the theorem still holds because the new assumptions exclude previously legitimate requests. Evidence that a narrower assumption is true does not authorize withdrawal of a promised capability. New premises need both factual justification and compatibility with the accepted change. Outstanding operations carry their applicable obligations through the transition; a policy cutover cannot silently discard them.

This creates a controlled route from a change request to an amended contract and then to implementation evidence. For design evolution, the authorized differences identify which responsibilities, interfaces, ownership or dependency constraints may change and which remain applicable. A coding agent can perform substantial design, translation and proof work within that route. Authority to decide the business or design change remains distinct from capability to edit the specification.

8.4 Assurance strength and its limits

A deductive proof can establish a property for all behaviors represented by its semantics and assumptions. Finite model checking establishes a result for the explored model. Tests establish observed outcomes for exercised cases; broader statistical claims require an appropriate sampling and measurement argument. Runtime evidence establishes behavior at its observed boundary and conditions.

These forms can be combined, but their combination does not silently upgrade the weakest material connection. A fully proved reference model connected to production only by sample tests remains dependent on that sampled connection. The result should explain the covered properties and the strength of each consequential inference.

Evidence applies to the source or artifact, models, assumptions and tool configuration it actually concerns. Delivery must preserve the connection from that checked subject to the built artifact and its deployment conditions. A proof about one library model cannot silently transfer to a different dependency; source verification cannot establish a runtime configuration it never addressed. The method requires this substantive connection without prescribing a universal manifest or toolchain. Where evidence shares a generator or premise, that dependency remains part of the argument rather than being counted as independent corroboration.

Reusing component contracts additionally requires a sound composition argument. Internal assumptions must be discharged by compatible component guarantees and justified environmental conditions, using a rule applicable to the properties and interaction semantics. Shared identities and events must retain compatible meanings across the contracts. Assumptions left outside the composition remain explicit conditions of the resulting system claim. Abadi and Lamport's composition research establishes why these connections require more than separately valid component specifications. [49]

In particular, a component that completes if another completes and a second that completes if the first completes may both satisfy their contracts while neither makes progress. Mutually conditional progress claims do not establish their own premises. Circular reasoning can be sound under appropriate composition rules, but its property-specific conditions must be met. The missing evidence in this example concerns the joint progress claim, not the validity of either conditional claim or Proposition 3. [49]

8.5 Semantic preservation and reduction of entropy

Successful eduction resolves specified uncertainties and makes remaining ones explicit. Investigation can also reveal previously hidden conflicts, so uncertainty need not decrease monotonically. Maintained conformance addresses unauthorized changes to covered meaning. Architectural simplification is a separate engineering operation: two implementations can be behaviorally equivalent while differing greatly in coupling, duplication and maintenance cost.

A semantic foundation can support restructuring by making preservation obligations explicit. Claims that entropy has been reduced should identify what changed: unresolved rule conflicts, undocumented dependencies, duplicated decision logic, unsupported assumptions or structural complexity. Neither the existence of a specification nor a successful proof measures overall evolvability, engineering productivity or DORA performance by itself.

8.6 Preserving meaning through corpus and system change

Change can affect required meaning, its representation or the implementation relationship, including when the system becomes simpler. A clear document can become false of the implementation. An unchanged requirement can acquire new environmental dependencies. Replacing persistent instruction state with request-local state, for example, can reduce code while losing the identity required across crashes. Eduction addresses the meaning and premises; renewed correspondence addresses the changed implementation.

Formal specifications themselves also evolve. Luckcuck, Farrell and Sheridan examined duplicated information and dependencies in FRETish requirements for an aircraft engine controller. After abstraction of detailed terms to Boolean variables, they used Spot to compare original and refactored temporal-logic formulas: 24 of 30 were equivalent; the remaining six refactored formulas implied their originals. This is evidence, within that abstraction and case, for distinguishing equivalence from strengthening while restructuring specifications. It does not show that formal artifacts maintain themselves or that ordinary documentation must deteriorate. [36]

An unresolved ambiguity remains subject to Section 4.5 even if the corpus has been reorganized. A revised formal contract remains subject to adequacy, capability preservation and correspondence even if its statements are clearer. The relevant continuity is the justified relationship among meaning, implementation and operating scope.

8.7 Deployment is part of the behavior being preserved

Continuous preservation concerns the history spanning an upgrade. Suppose both the old and new payment implementations satisfy their contracts from separately initialized state. A migration that deletes historical deduplication records can nevertheless let a delayed message create a second capture. Version-by-version correctness does not exclude that history. This is a constructed consequence of the distinction between initialized executions and the actual transition.

Let s denote the old state and h the relevant history, including accepted instructions and external effects. Let M(s, h, s', h') describe a migration and its history interpretation. A necessary invariant-transfer obligation is:

$$\mathrm{Inv}_v(s,h) \wedge M(s,h,s',h') \;\Rightarrow\; \mathrm{Inv}_{v+1}(s',h').$$

The interpretation of h' must retain the relevant earlier facts; deleting a local record cannot erase an earlier financial effect. The migration model must correspond to the actual data transformation and its failure paths. Invariant transfer alone also leaves effects during migration, progress, and intermediate mixed-version states to be established.

The full subject is consequently the permitted upgrade process U: old execution, data transformation, mixed-version execution where allowed, new execution and any supported rollback. With S_U expressing the authorized contract across that history, continuous assurance requires:

$$\alpha_U(\mathrm{Beh}_A(U)) \subseteq \mathrm{Beh}(S_U).$$

This applies the existing correspondence principle to the deployment process rather than treating the final executable as its complete subject. In particular:

- **State and history:** identities, retained obligations and prior effects survive migration with their meaning intact; a partially applied migration cannot create a fresh logical instruction from an old one.

- **In-flight work:** delayed messages, pending requests and retries retain their applicable contract and identity through cutover, or are completed before a justified boundary that excludes their later return.
- **Mixed versions:** every permitted producer, consumer and storage combination preserves the retained semantics. An incompatible combination must be excluded by the actual rollout mechanism.
- **Rollback:** a supported older binary can interpret the state and messages produced during the attempted upgrade, or a justified transformation restores that compatibility. Restoring the binary alone does not restore the earlier world.
- **Service continuity:** progress and previously promised operating scope remain satisfied outside the authorized change, including any explicitly permitted interruption.

The necessary evidence follows the real deployment strategy. A system that never permits mixed versions has a different proof obligation from one using rolling deployment, but it still needs evidence that the exclusion and handover hold. These are obligations derived from the stated continuity claim; this paper does not claim to have verified a particular deployment process.

8.8 Continuing applicability of operating assumptions

A material assumption can cease to hold without a software change passing through admission. A provider may change an external guarantee while the application's binary and proofs remain unchanged. The theorem remains valid as a conditional result, but its applicability to the running service is then unsupported. Continuing assurance therefore includes the continued applicability of the premises used by the claim. ACCESS's distinction between development-time and runtime evaluation provides a related treatment of assurance elements whose validity depends on operating data. [47]

For each material premise, the assurance account explains what sustains it: an enforcing mechanism, an effectively pinned dependency, evidence from relevant observations and reassessment, or a retained external condition. Pinning an API version does not by itself preserve a provider's operating behavior. An observation supports only the conditions and interval its evidence covers; periodic checks alone cannot establish uninterrupted validity. Unobservable premises remain explicit conditions rather than implied runtime guarantees.

Where operational protection is required, the system's response to invalidation must preserve the obligations supported by the remaining premises, including outstanding work. Suspending an unsafe effect may preserve safety while leaving progress unresolved. Detection after an irreversible effect cannot establish its prevention. Continued operation, reaction timing and any withdrawal of service therefore need their own justified conditions. An unconditional assurance claim cannot continue on invalid premises; restoration requires evidence that the premises hold again or that a revised foundation supports the applicable obligation.

9 Analytical examples and Signal experience report

9.1 Evidence that resolves the initial statement

This is a constructed analytical example. Its stipulated evidence and deductions demonstrate the eduction reasoning; they are not a report of an executed payment implementation. The system has an order service, payment adapter, database, queue and external provider. Its initial

requirement says that payments must be idempotent, and its tests show that an HTTP retry returns the same local response.

Three additional sources give that statement a usable interpretation:

- **Applicable product policy:** separate shipments may have distinct capture instructions. Each instruction permits at most one successful capture. A retry of that instruction creates no new entitlement to charge.
- **Persistence and dispatch semantics:** acceptance commits a merchant-scoped instruction identity k, immutable payload and unique provider key f(k), together with a durable dispatch record. A uniqueness constraint makes concurrent acceptance of the same k reuse that record. A different payload under the same identity is rejected. The provider performs the financial effect before the application necessarily knows its outcome.
- **Provider contract:** repeated requests using the same key and payload are deduplicated within a specified window; distinct keys can produce distinct captures. A separate outcome-resolution guarantee, where available, supplies an authoritative eventual result and excludes later completion after a definitive failure. Timing and transport bounds are needed to justify that a repeated request reaches the provider within its protected window.

The policy selects the business identity. The persistence account connects it to concrete events. The provider contract determines what retry and resolution promises are realizable. None of those conclusions follows from the local response test alone.

9.2 Reconstructing the obligation and its identities

The acceptance event is the durable commit of the instruction and dispatch record, before external dispatch. Let k include the merchant account and logical instruction identity. Its payload contains payment identity, amount in minor currency units and currency. The persisted mapping f assigns different provider keys to distinct accepted instructions within the provider's key scope, and preserves one key for every retry of the same instruction.

Let Effects be the successful provider captures caused through the application's covered payment interfaces, including recovery and administrative paths. Every effect e must have exactly one attributable accepted instruction owner(e), and its payload must match that instruction. The attribution is established through the dispatch identity and authoritative provider evidence; an unassociated effect cannot disappear from the property merely because no local record names it.

The principal obligations are therefore:

Every e in Effects has a unique accepted owner(e) with matching payload.

For every accepted k,

$$\left|\{e \in \text{Effects} : \text{owner}(e) = k\}\right| \leq 1.$$

These preserve separate captures for separate shipment instructions, including within one order. Required capabilities also include processing legitimate instructions and making progress under the agreed conditions. Under eventual provider resolution and fair execution of the required recovery work, each accepted instruction must eventually have a known successful or definitively failed outcome. A provider decline is compatible with that promise; eventual successful charging is a stronger, unjustified promise.

Establishing identity also determines retention: the system cannot discard its mapping while a delayed request, outstanding effect or supported recovery path can still refer to that instruction. Migration must preserve this meaning as required by Section 8.7.

9.3 Deriving transitions and environmental premises

The specification S distinguishes a committed instruction, a dispatched instruction with potentially unknown outcome, a confirmed success, a definitive failure and a held instruction awaiting resolution. Its critical transitions are:

- **Accept:** commit k, its payload, f(k) and the dispatch record atomically. Concurrent retries reuse the same identity and immutable contents.
- **Dispatch:** send the persisted key and payload and retain durable evidence of the attempt. A crash after the provider's effect but before the response leaves a potentially successful instruction, not proof of failure.
- **Retry:** reuse f(k) and the payload only while the provider's deduplication guarantee demonstrably covers the request, including arrival and concurrent or delayed attempts. A local timer alone is insufficient without the relevant timing and transport premises.
- **Resolve:** use authoritative provider evidence to establish success or definitive failure. A negative lookup that permits an earlier request to complete later is not a definitive failure.
- **Hold:** when an outcome remains unknown beyond the justified retry window, preserve the instruction and stop unsafe resubmission. Held work remains unresolved; renaming it a completed failure would misstate the financial outcome.

External capture is an event in the model, governed by the stated provider contract rather than by the arrival of a local response. Crash and restart preserve the committed identity and recover its pending state. The supporting assumptions A concern database durability and uniqueness, coverage of effect-producing paths, provider key scope and deduplication, and the timing and resolution guarantees actually used.

If deduplication has expired and no authoritative resolution is available, safety can require holding the instruction. The system cannot guarantee both safe repetition and eventual successful capture from that evidence: in one compatible situation a capture already occurred, while in another it did not. A prospective decision to permit a terminal unresolved outcome changes the promised service; it does not prove a known financial result. This is an application of the information argument to a concrete specification choice.

9.4 The foundation and change reasoning

Here R contains effect attribution, uniqueness, legitimate separate captures and conditional progress. S contains the state distinctions and transitions above. A contains the supported persistence and provider premises. The interpretation α maps durable acceptance commits to accepted instructions, dispatch records to attempts, and provider capture evidence to financial effects owned through f(k). It preserves unknown outcomes rather than treating the absence of a response as the absence of an effect. J consists, in this analytical example, of the stipulated premises and the deductions they support; it contains no claim of production verification.

To illustrate design drift, extend the analytical example with an accepted design obligation that only the payment adapter may initiate provider captures. A coding agent proposes a recovery design in which a worker calls the provider directly, outside the authorized change. That proposal

departs from the responsibility boundary even if the behavioral obligations remain satisfied. A structural check covering the effect-producing dependency can expose the departure; a uniqueness proof alone does not check that design obligation. An authorized redesign would instead justify the revised ownership, preserve the relevant behavior and establish correspondence for every new dispatch path.

Consider changing recovery to generate a fresh provider key. A prohibited history follows directly: accept k, capture using f(k), lose the response, restart with a fresh key, then capture again. The provider premise permits the second effect, while both effects belong to k. The uniqueness obligation is violated. A correspondence check that models this path must therefore fail; a check of an unchanged model alone would miss the changed mapping. This is a derived counterexample, not a reported tool execution.

A refactoring that moves dispatch between workers while preserving the same abstract transitions, identities, premises and fair scheduling has a different result. If each new execution maps to the permitted transition histories, behavioral correspondence transfers the obligations by Proposition 3. This establishes the behavioral preservation condition; the refactoring must also satisfy the accepted design obligations, including any retained adapter boundary. Production acceptance still requires evidence that the implementation and deployment possess those properties.

This derivation exposes the substantive eduction work: selecting the instruction rather than the order as the semantic identity, locating the actual effect, deriving the required persistence and retention, separating unknown from failed, and qualifying progress by the provider's capabilities. Implementation protection then depends on evidence covering every relevant dispatch and recovery path, external assumptions and the transition between versions.

9.5 Signal: the system and its documentation

Signal is a language-centric data streaming system in which authored statements compile into released execution plans for seven operator families. Its semantics cover append-only versioned products, temporal aggregation, completeness, replay, external context and delivery of results. Inspection of the repository at commit 64829b9614df4e929fd11a59c88d1b44e31b5a3d found 49 Markdown documents under the documentation directory, including language semantics, logical and technical architecture, operator specifications and use cases. Source code and tests accompany that corpus. These observations establish the system's relevant engineering context; document volume alone establishes neither specification adequacy nor implementation correctness. [45]

The documentation contains substantive semantic distinctions. The Signal Expression Language specification defines a denotation over authored statements, evidence and committed input history, and requires append-only corrections and replayable decision evidence. The Consistent-Cut Finalization specification distinguishes a transport commit from business finality: committing a Kafka transaction does not establish that all required business dependencies have closed. It also distinguishes an empty dependency result from a missing closure and an allowed failure from an unresolved outcome. These distinctions determine when an output may be treated as final. [45]

The inspected closure-gate implementation has corresponding decision points: it checks released dependency requirements, coverage, closure evidence and permitted terminal status before admitting a gate. This is a concrete connection between a documented distinction and implementation structure, rather than proof of complete conformance. A practitioner applying the method must understand what the evidence represents and which conditions permit finality to justify a specification of that behavior. The required knowledge concerns Signal's meaning of completion, beyond the generic capability to generate streaming code or tests. [45]

9.6 Reported comparison with six coding agents

The author reports that Signal was developed using a spec-driven approach and that its extensive documentation was supplied to coding agents to reconstruct system intent. The comparison involved six agent instances, identified as GPT-6 Astra configured at Max reasoning. The reported outcomes concern both written interpretations and generated or modified implementations. [46]

With documentation as the source of intent, the agents produced interpretations and implementations that the author assessed as departing from the intended semantics. The author then used semantic eduction to construct a Lean foundation. In subsequent uses described by the author, the assessed interpretations and implementations aligned with the intended semantics. Here, drift and alignment denote the author's assessment; the available account establishes neither a success rate nor that each implementation was proved to refine the Lean foundation. [46]

9.7 What the Signal evidence supports

The repository inspection establishes the documented system context; the practitioner's report supplies a firsthand qualitative observation about agent outputs. The Lean foundation is outside the inspected repository. Original prompts, outputs and assessment records were unavailable, and the inspected snapshot does not establish the exact documentation supplied in either condition. The six instances share a reported model configuration; separate run counts for each condition were not supplied. No inspectable before-and-after output is presented, and the comparison was not rerun.

The causal mechanism remains unresolved. A relevant rule could have been absent, ambiguous, inconsistent or already explicit but misinterpreted. Eduction could have added information, changed its representation or enabled mechanical rejection. The available account does not distinguish those contributions from changed instructions or other differences between uses. The repository's transport-commit and business-finality distinction illustrates system semantics; it is not identified as an observed agent failure. Drift despite documentation therefore does not establish an information deficit, and improved alignment does not isolate the effect of Lean or proof enforcement.

The case illustrates a reported application, without establishing a general reliability rate or continuous implementation correspondence. The conditional necessity claims rest on the arguments and research elsewhere in the paper. Signal's reported Lean foundation is distinct from this paper's companion proofs in Appendix A.

9.8 Educing latent semantics from Excel financial operations

This constructed analytical example concerns a financial firm whose books and reporting are maintained through Excel workbooks, reports and the practices of accountants and controllers. No dedicated target application exists. The stipulated evidence illustrates eduction before coding-agent implementation; it adds no empirical claim about industry prevalence or the method's effectiveness.

The knowledge is distributed across three sources. Reports show balances, classifications, periods and relationships between figures. Workbook formulas implement calculations and dependencies, including ranges, lookups, links and copied values. Accountants and controllers hold process knowledge about timing, adjustments, approvals, exceptions and manual interventions. None of the sources alone supplies the complete meaning: a reported balance may depend on both a formula and a controller's adjustment whose trigger and rationale are unstated in the workbook.

Establish the obligation and its dependencies. Start from a consequential report figure and reconstruct which records, formulas and process actions determine it. In this example, tracing a closing balance exposes an adjustment entered before publication, after the ordinary transaction extract was prepared. The formula explains how its value enters the total; the process account explains why it belongs to that period, who may approve it and how it appears in later reports. Applicable reporting policy, workbook relationships and retained adjustment records support those different claims. A hard-coded value or remembered convention is examined for its role and authority rather than automatically promoted to a requirement.

For this illustration, the accepted policy requires each included transaction or adjustment to have a stable identity, entity, account, currency, reporting period and signed amount. Each record contributes at most once to a report revision. An adjustment must have the required approval before posting, and each report revision must identify its included records. Corrections to a closed report create an attributable revision rather than silently replacing its prior meaning. These are stipulated obligations for the example; unresolved choices about approvals, cutoffs or corrections would require additional evidence or an authorized prospective decision.

Produce a formal and executable semantic basis. Let q identify an entity, account, currency and reporting period. Let T(q) be the set of transactions assigned to that report scope and Adj(q) the set of approved, posted adjustments assigned to it. A simplified educed balance obligation is:

$$B_{\text{close}}(q) = B_{\text{open}}(q) + \sum_{x \in T(q)} \delta(x) + \sum_{a \in \text{Adj}(q)} \delta(a).$$

The interpretation must establish each set's membership and the sign and unit of every amount. The opening balance has an identified source and period relationship; missing source evidence is not interpreted as a zero balance. Identity prevents a repeated import or repeated posting from contributing a new amount. Approval, cutoff, rounding and any currency conversion used by the process have explicit meanings. A formula that sums a convenient range cannot establish the obligation if a legitimate adjustment lies outside it.

The specification defines states and transitions for recording transactions, proposing and approving adjustments, posting them and closing or revising a report. An executable reference function can calculate balances for the identified record sets; a state model can express authorization, inclusion and revision rules. Explanatory material retains the reconstructed process, reasons and source relationships. This is a usable artifact before target code exists. Its adequacy rests on those meanings and premises; a successful model calculation alone supplies no correspondence evidence about future code.

Use the basis for first implementation and later evolution. Supply the coding agent with that vocabulary, state model, calculation semantics and accepted constraints. For this example, an authorized initial design assigns changes to book state to a posting component and makes reporting read an identified revision. A design that lets the reporting component alter adjustments directly departs from that commitment. A first implementation that calculates a closing balance from transaction rows alone omits approved adjustments and fails the behavioral contract. Either departure can occur without a prior target codebase.

Implementation assurance now needs an actual mapping from records, identifiers, authorization checks, amount representations and report revisions to the specification, and evidence covering the relevant import, posting and reporting paths. An executable model can supply an oracle for exercised cases; stronger claims require correspondingly stronger checking. Historical reports help assess educed meanings and calculations at their recorded scope, while agreement on a few totals cannot establish coverage of every exception or transition.

Subsequent changes use the same semantic basis. Replacing cell-address calculations with queries grouped by entity, account, currency and period can preserve meaning if it retains the justified membership, identity, amount, approval and revision rules and establishes correspondence. Treating an absent approval as approval, or dropping adjustment identity during re-import, changes that meaning unless explicitly authorized and justified. Eduction establishes what the workbooks and process mean; checking establishes whether the agent's actual application realizes and continues to preserve it.

10 Using the foundation in development and evolution with coding agents

10.1 The foundation and its use in drift mitigation

Semantic eduction produces the justified foundation used to guide coding agents and assess design and implementation drift from initial development through subsequent evolution. It includes the eduction map and can include documents, architecture, formal models, executable specifications, contracts and proofs. The covered design and behavioral obligations determine which forms and implementation connections are necessary.

A supported semantic account is a substantive eduction result. It can guide initial design and implementation before target code exists. Readiness for implementation assurance additionally requires the connections and checking capability needed to assess the actual agent-produced implementation. Continuous mechanical assurance requires relevant checkable semantics and an established connection to the changing system. The required artifacts and evidence follow the stated use and assurance strength.

The foundation has three connected roles in development and evolution with coding agents:

- **Guide agent work.** Supply educed domain meaning, applicable behavior, accepted design commitments, relevant assumptions and the intended development or change so that the agent reasons from a justified account.
- **Assess design proposals.** Check initial and revised designs against represented responsibilities, interfaces, ownership and dependency constraints, with evidence appropriate to the proposed design and the claim being made.
- **Check implementations and changes.** Establish conformance evidence for the first implementation and re-establish affected evidence against the accepted or deliberately revised contract afterward, including relevant code, configuration, dependencies and transition behavior.

The method can be applied across teams developing or evolving software with coding agents because its reasoning obligations recur. The particular invariants, design constraints, abstractions, languages, tool support and effect boundaries vary with each application. Standardization concerns the discipline of establishing meaning and evidence for this purpose. It does not require imposing one modeling language or one development workflow on every team.

For a team using SDD with coding agents, eduction addresses latent domain and process meaning, justified formalization and design and implementation correspondence. Existing documents remain evidence and explanation. An SDD workflow that already supplies the required semantic contract and justified conformance needs no duplicate assurance layer. Another team may need substantial work to establish payment identity, worksheet dependencies, adjustment rules,

transaction boundaries, external guarantees, component mappings or historical exceptions. The necessary work follows the missing connections for the application.

For initial development, first supply the agent with the educed vocabulary, obligations, domain assumptions and justified specification. Assess its proposed design against that basis, establishing or explicitly authorizing prospective architectural commitments. The agent implements the design and supplies the required verification evidence; admission then assesses the actual implementation and its relevant build and operating conditions. Later tasks identify the authorized semantic difference, retain the applicable obligations, and renew affected correspondence and transition evidence as in Section 8. A change to the contract or checking assumptions is a substantive proposal requiring its own justification, rather than a way to make an implementation pass.

10.2 Understanding operational and implementation mechanisms

Establishing why the selected specification and checking method support the covered claim requires understanding the relevant domain and operational mechanisms, and the implementation mechanisms when present. Before target code exists, a worksheet formula, reporting cutoff or manually approved adjustment can determine a required distinction. In software, a transaction may end before an external capture, a queue may redeliver an instruction, and a recovery job may use a different provider endpoint. Those mechanisms determine the relevant identity, permitted histories and implementation connection. The five activities in Section 6.3 establish them; the selection procedure in Section 7.7 turns them into a justified technical choice.

Engineers, documentation and automated analysis can supply the evidence. The practitioner accepting the result needs enough understanding to judge its applicability and explain what a passing check establishes. Appendix B specifies the required access and competence, scoped to consequential dependencies rather than exhaustive knowledge of the estate.

10.3 Human leadership and the division of work

Human leadership requires practitioners to perform and integrate the comprehensive examination that establishes the eduction map. This includes locating sources of latent knowledge, reconstructing relationships across business and technical domains, investigating consequential mechanisms and judging which distinctions and structures the contract must retain. Policy and prospective design choices remain with the responsible authority or its delegates. The human contribution establishes the justified semantic task that automated eduction carries out.

For a delegated analysis, the map supplies the relevant phenomena and sources, their supported relationships, the abstraction to construct and the claim its result must support. Tools can search, extract dependencies, propose interpretations, generate models, synthesize invariants and construct proofs. Practitioners integrate new findings into the domain account and justify changes to the map. A discovered path or counterexample can reopen an interpretation; a passing check supports the represented obligation under its assumptions.

Existing work demonstrates substantial automated discovery within defined tasks. Code Researcher searches large systems code and commit history to investigate supplied crash reports and produce patches. Its authors distinguish crash suppression from fixing the underlying issue and preserving uncovered functionality. This supports automated assistance with consequential code exploration; it does not establish comprehensive eduction across business domains, documents, workbooks and latent process knowledge. [53]

Sutton's Bitter Lesson argues from historical AI results for scalable search and learning and for

agents capable of discovery. That argument supports increasing automation of discovery. It does not establish that a particular enterprise's sources, intentions or abstractions have already been discovered. For the present method, the consequence is to retain the discovery and justification obligations as automation improves: more computation over indistinguishable evidence cannot resolve the missing distinction proved in Section 6.1. [52]

The cited studies establish particular assisted and automated activities. The proposed method integrates them through required expert-led examination and a justified eduction map. Its net effect on delivery and maintenance costs remains unmeasured.

11 Evidence strength and research contribution

11.1 What is established

Requirements engineering establishes the role of domain knowledge in connecting a specification to its required outcome. Oracle research distinguishes producing executions from judging correctness. Refinement research supplies a formal basis for implementation correspondence. Specification-mining research establishes automated extraction and invariant synthesis within stated conditions. Industrial reports document human-guided selection of relevant state and business rules. These results support the activities of the proposed method; their relationship to its eduction and assurance obligations is developed explicitly in Section 6.

The information arguments establish conditional limits. Proposition 1 applies when the acceptance representation fails to distinguish a required difference. Proposition 2 applies when available evidence leaves intended semantics underdetermined; the eduction-map corollary shows that an intermediate map cannot supply the absent distinction. Establishing these premises for a particular system requires examining its evidence. Complexity alone does not establish them. Expert-led examination and map construction are explicit requirements of the proposed method, supported by the discovery mechanisms and bounded industrial precedents described in Sections 4 to 6. The logical results establish the information obligation, rather than biological exclusivity; the quality and effectiveness of the method's human judgments remain empirical properties.

Evidence on document-centered workflows has distinct roles. SpecMine documents public artifacts; product documentation establishes described capabilities; requirements and language-model studies show interpretation failures in their evaluated settings. AWS supplies industrial experience of design checking, while reflexion models and controlled-English architecture checking show implementation connections. Together with the positive counterexamples in Section 3, these support the capability-based distinction between specification quality and conformance. The documentation corollary establishes logical insufficiency where that connection is absent; it does not supply an empirical error rate or a universal threshold of agent reliability.

The practitioner evidence is confined to the Signal account in Sections 9.5 to 9.7. It illustrates application without independently validating the complete method or identifying the cause of the reported improvement.

11.2 Closest approaches and the contribution

Assurance-case engineering organizes claims, arguments and supporting evidence. ACCESS develops systems around evolving assurance cases connected to heterogeneous engineering artifacts, formal analysis, change-impact analysis and runtime evaluation, with an autonomous-underwater-vehicle case study. It is a close methodological comparator: integration of evidence and continuous assurance are already established research subjects. The proposed method em-

phasizes the eduction work needed to construct justified obligations, meanings and premises when business and technical knowledge is latent, fragmented or conflicting, with or without an existing target application. That work can supply an assurance case; merely arranging unsupported claims within one does not resolve their missing meaning. The distinction describes this paper's emphasis, not a claim that assurance-case methods are incapable of such reconstruction. [47]

Proof-carrying code separates an untrusted code producer from a consumer that checks proof against an established safety policy. It is a direct precedent for accepting generated code through independently checked evidence rather than confidence in the generator. The proposed method applies this separation to agent delegation while addressing an earlier question: which system-specific policy, environmental premises and interpretation should the evidence concern? Proof checking answers a defined obligation; semantic eduction supplies justification for selecting and grounding that obligation. [48]

The Aeneas workflow gives a concrete implementation of several related activities: independently formalized standards, source extraction, correspondence proofs and constrained delegation to agents. Its published standards and supported Rust translation provide inputs that eduction from latent business knowledge or heterogeneous existing systems may first need to establish. The report is a precedent for the division of work and its trusted boundaries. [9]

The contribution is a synthesis for enabling drift mitigation during software development and evolution with coding agents. It constructs the semantic foundation by surfacing and reconstructing latent knowledge in business and technical evidence: distinguishing obligations from accidents, establishing applicable design commitments, grounding identities and events, establishing environmental premises, selecting a suitable representation and connecting it to agent design proposals, first and subsequent implementations, and deployment. Sections 6 and 9 make the reasoning inspectable; Section 8 derives the resulting admission and change obligations. Its common structure can yield different artifacts across stacks because the evidence and effect boundaries belong to the particular application. Comparative superiority over established approaches is not claimed.

11.3 What remains an empirical claim

The complete method's effectiveness at mitigating design and implementation drift during initial development and across sequences of coding-agent changes requires evidence from its application. Its effects on delivery speed, defect escape, maintenance cost and engineering performance also require such evidence. The Signal account concerns qualitative agent alignment; the payment and Excel examples are analytical constructions. This paper reports no quantified drift reduction or measurements of those broader outcomes and does not infer them from unrelated benchmarks.

Feasibility across heterogeneous stacks, agreement among independent practitioners, coverage of consequential obligations and the quality of eduction decisions are also empirical properties of the method. The logical results do not establish them, and the Signal account does not establish their generality. The method is designed for repeatable reasoning; independent reproducibility remains an open evidential question.

Existing delivery records can substantiate particular claims through educed requirements and their sources, resolved ambiguities, actual implementation connections, discovered defects, recorded changes and maintenance effort. The initial-development claim includes eduction from pre-application evidence and correspondence of the first implementation; evolution additionally includes maintained obligations across changes. Attribution must account for simultaneous code changes, practitioner experience and other plausible explanations. The relevant question is what those records establish about the claimed result, not whether the elementary information

arguments can be demonstrated again.

11.4 Conditions that limit or change the method

An adequate existing formal contract can remove much of the eduction work. A tool with a justified complete model for the relevant component can automate more of the work. Missing evidence can prevent a human-led process from establishing the desired contract. Unsupported implementation features can limit the strength of correspondence. Contradictory requirements can require a policy decision rather than further inference.

None of these conditions invalidates the method's basic reasoning. They determine its applicable work and the claims it can support. Applying the method requires making the resulting coverage and uncertainty specific to the relevant obligation.

12 Conclusion

Semantic eduction is proposed here as an enabler for mitigating design and implementation drift when coding agents develop and evolve software. Practitioners comprehensively examine the relevant business and technical environment and construct a justified eduction map connecting latent knowledge, consequential dependencies and abstractions to executable semantics. That map directs machine analysis and establishes why the specification guiding the agent's work is appropriate to the application. The method reconstructs applicable behavioral and design obligations from reports, formulas, data, documents, operating practices, subject-matter expertise and existing software where present. It grounds domain concepts, establishes mechanisms and premises, justifies a specification and connects it to initial designs, implementations and subsequent changes. Where evidence leaves a consequential distinction unresolved, additional evidence, an authorized decision or a change justified across the remaining interpretations determines what can proceed.

The foundation takes the forms needed by the covered agent work: explanatory documents, architectural accounts, formal models, executable specifications, contracts and proofs. It can guide a first application before target code exists. Design drift mitigation connects accepted responsibilities, interfaces, ownership and constraints to the proposed and realized design. Implementation drift mitigation establishes conformance of actual code and behavior to that basis, beginning with the first implementation. Continuous assurance additionally requires preservation of capabilities and operating scope, sound composition of component evidence, and coverage of the transition between versions. A guarantee remains conditional on those connections and the continuing applicability of its premises. The method's effectiveness remains an empirical claim.

The methodology can operate within spec-driven development and use existing verification capabilities. Its result is an educed and usable semantic foundation, together with the connections and checks needed to guide coding agents, assess their design proposals and check their implementations at the stated assurance strength. The proposed discipline is designed for repeatable application to this purpose; the meaning it establishes and the evidence needed to realize and preserve it belong to the domain and application under consideration.

A Mechanically checked arguments

The companion source, SemanticRecovery.lean, accompanies the manuscript as a standalone file. It formalizes the deterministic core of the paper using Lean's Init library and contains seven results:

- sameObservationSameDecision proves that equal observations produce equal deterministic acceptance decisions.
- noSoundDistinguishingAcceptance proves the contradiction between accepting an indistinguishable forbidden candidate and the claimed soundness of the gate.
- noUniqueRecoveryFromAmbiguousEvidence proves that one inferred classification cannot match conflicting requirements in two situations with identical evidence.
- noMapResolvesAmbiguousEvidence proves that constructing and interpreting an intermediate map from that evidence cannot resolve the conflicting classification.
- documentQualityAloneDoesNotEstablishConformance proves the documentation corollary under its explicit premise that a valid document can accompany a nonconforming implementation.
- reencodingPreservesIndistinguishability proves that deterministic re-encoding preserves equality of available evidence.
- admittedBehaviorSatisfiesCoveredRequirement proves that admission, implementation correspondence and specification adequacy compose to establish the covered property.

The source was checked on 5 October 2026 with the installed Lean 4.33.0 toolchain. Compilation succeeded. Its seven embedded axiom-inspection commands each reported that the named theorem "does not depend on any axioms". Running the source reproduces those messages. The proofs use no admitted goals or asserted custom axioms.

The source supplies reusable logical implications, not a formal model of a particular system. Its assumptions remain parameters. It does not prove that a particular system has missing information, that a chosen requirement is legitimate, that a concrete extraction tool is sound, or that semantic eduction outperforms an automated alternative. The remarks about randomized procedures, conservative reconstruction, realizability and deployment continuity remain prose arguments and obligations; the Lean development covers the seven deterministic statements listed above. It is independent of Signal's separately reported Lean foundation.

From the directory containing the companion source, the verification command is:

```
elan run leanprover/lean4:v4.33.0 lean SemanticRecovery.lean
```

B Evidence access and practitioner competence

The following requirements are practical implications of the five eduction activities and their completion criterion. They specify the evidence access and competence needed to establish the semantic foundation for guiding and checking coding-agent development and evolution. The necessary depth follows the behavioral and design obligations being established and the evidence connecting them to the domain, process and implementation.

B.1 Access to domain, process and implementation evidence

For each covered obligation, access must support its meaning, consequential dependencies and environmental premises, together with relevant implementation paths when assessing software. This follows from the relationship between domain knowledge, specification and requirements in Section 2.1. A dependency can be examined directly or represented by an applicable, justified contract. The evidence must support whichever boundary the assurance argument uses. Access to target code is required for claims relying on that code; its absence does not prevent eduction of a semantic basis for a first application. [1]

- **Business and process evidence.** Documents, reports, data, worksheets and formulas, together with relevant subject-matter accounts of established processes, decisions, exceptions and manual interventions. Eduction relates these sources to surface meanings and dependencies not explicitly stated in any single artifact.
- **Meaning and environmental commitments.** Applicable requirements, business rules, data meanings, architectural decisions, historical records and external service contracts needed to distinguish obligations from incidental or defective practice. Prospective policy and design decisions are identified as such.
- **Implementation evidence when software is assessed.** Readable source and related schemas, migrations, configuration, dependency versions, tests and build information for the covered behavior. The scope includes consequential services, background jobs, recovery paths and external effects, even when they belong to another repository or team. A proposed implementation plan supplies no substitute for evidence about the actual implementation.
- **Operating evidence where the claim depends on it.** Existing traces, incident records, deployment facts and relevant state or data observations needed to assess operating premises. Sanitized records are suitable where the transformation preserves the distinctions needed by the claim.
- **Analysis and verification capability.** The ability to perform the selected analyses and run the retained checks, in an authorized environment, with results attributable to the relevant source, configuration and tool assumptions.

Eduction can use read-only access to existing evidence with analysis performed on controlled copies. Versioned document, worksheet and data exports, and sufficiently informative results from authorized analyses, can supply evidence without personal credentials to every service. Subject-matter accounts are assessed for their evidential fit, consistency and limits. Integrating persistent implementation checks requires the corresponding authorized source or build changes; production administration is not a prerequisite of the method.

If decisive evidence is unavailable, the affected obligation remains unsupported at the intended strength. A qualified result must make that gap explicit; it cannot count as completion of a broader objective that requires the unresolved guarantee. Access sufficiency is therefore a property of the evidence for the claim, rather than the breadth of permissions granted.

B.2 Competence required to apply the method

Applying the method requires five connected capabilities:

- **Cross-domain discovery and judgment.** Locate and examine the relevant business and technical artifacts and subject-matter knowledge; reconstruct their consequential relationships; compile and justify the eduction map; interpret identities, units, exceptions and

required outcomes; distinguish essential domain complexity, legitimate obligations, defects, incidental structure and unresolved policy choices.

- **Process and system behavior analysis.** Reconstruct relevant data, formulas, state, decisions and effects across business processes and software components, including manual interventions, concurrency, failures, retries and recovery where these affect the obligation.
- **Formal modeling.** Construct invariants, permitted transitions and progress conditions; justify abstractions; detect vacuous satisfaction, contradictory assumptions and the exclusion of required behavior.
- **Verification engineering.** Select and apply appropriate checking methods, establish implementation correspondence, and distinguish the conclusions supported by tests, bounded checking and proof. Assess tool coverage and trusted assumptions, including those behind AI-generated results.
- **Relevant technical competence.** Understand worksheet and data semantics where they contain latent knowledge, and the selected language, runtime, persistence, framework, protocol and deployment semantics on which implementation claims depend. The required depth follows the mechanism: a reporting claim may depend on formula ranges and adjustment timing, a financial-effect claim on transaction boundaries and provider behavior, and a low-level memory claim on aliasing and the memory model.

The industrial rule-extraction study in Section 5.8 illustrates the combination of business selection and technical extraction: users selected consequential outputs while code familiarity helped identify obsolete rules. It supports that combination in its reported setting; the competence profile above is derived from this paper's engineering obligations. [43]

B.3 Applying a portable method across different stacks

The method transfers across stacks used for development and evolution with coding agents through its common obligations: establish meaning, premises, adequacy and design and implementation correspondence. Each application requires the domain and technical competence relevant to those obligations. Engineers and subject-matter specialists can supply that competence together. The person directing eduction needs sufficient understanding to integrate their contributions, justify the eduction map and explain the resulting assurance.

A practical competence criterion is the ability to justify the chain from latent domain or process knowledge through an accepted obligation to its formal representation, and then to actual system effects when implementation assurance is claimed, identifying where supporting evidence ends. Expertise in a specification language alone does not establish that chain.

References

[1] Zave, P., and Jackson, M. (1997). **Four Dark Corners of Requirements Engineering.** ACM Transactions on Software Engineering and Methodology 6(1), 1-30. The linked author manuscript is dated 1996. Author manuscript.

[2] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). **The Oracle Problem in Software Testing: A Survey.** IEEE Transactions on Software Engineering 41(5), 507-525. Author copy.

[3] Chikofsky, E. J., and Cross, J. H. II (1990). **Reverse Engineering and Design Recovery: A Taxonomy.** IEEE Software 7(1), 13-17. Paper.

[4] University of Washington Program Languages and Software Engineering Group. **The Daikon Invariant Detector.** Maintained project documentation, consulted 4 October 2026. Documentation.

[5] Infer project. **Separation Logic and Bi-abduction.** Maintained technical documentation, consulted 4 October 2026. Documentation.

[6] Neider, D., and Roy, R. (2025). **What Is Formal Verification without Specifications? A Survey on Mining LTL Specifications.** arXiv:2501.16274, version 1. Survey.

[7] Schuts, M., Hooman, J., and Vaandrager, F. (2016). **Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report.** Integrated Formal Methods, IFM 2016. Full paper.

[8] van der Aalst, W. M. P. (2018). **Process Discovery from Event Data: Relating Models and Logs Through Abstractions.** WIREs Data Mining and Knowledge Discovery 8(3), e1244. Author manuscript.

[9] Ho, S., Fournet, C., Protzenko, J., et al. (2026). **Scaling Verification of Cryptographic Software with Aeneas, Rust, and Lean.** Draft technical report, September 2026, arXiv:2609.15648, version 1. Preliminary industrial report; cited at its stated scope. Report.

[10] Ye, Z., et al. (2026). **Intent-aligned Formal Specification Synthesis via Traceable Refinement.** arXiv:2604.10392, version 1. Preprint describing VeriSpecGen. Paper.

[11] Mashhadi, M. J., Siddiqui, T. R., Hemmati, H., and Loewen, H. (2019). **Interactive Semi-automated Specification Mining for Debugging: An Experience Report.** Information and Software Technology 113, 20-38. DOI: 10.1016/j.infsof.2019.05.002. Author preprint.

[12] Abadi, M., and Lamport, L. (1988). **The Existence of Refinement Mappings.** Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science, 165-175. DOI: 10.1109/LICS.1988.5115. Conference paper.

[13] Lean project. **Validating a Lean Proof.** Lean language reference, consulted 4 October 2026. Documentation.

[14] Rocq project. **The Rocq Prover.** Official documentation, consulted 4 October 2026. Project documentation.

[15] Appel, A. W., and collaborators. **Verified Software Toolchain.** Project account of Verifiable C and its soundness connection, consulted 4 October 2026. Project and publications.

[16] **AutoCorres2.** Archive of Formal Proofs entry and associated sources, consulted 4 October 2026. Project entry.

[17] Dafny project. **The Dafny Programming and Verification Language.** Official documentation, consulted 4 October 2026. Documentation.

[18] Why3 project. **Why3 and WhyML.** Official technical overview, consulted 4 October 2026. Documentation.

[19] F* project. **F*: A Proof-Oriented Programming Language.** Official technical overview, consulted 4 October 2026. Documentation.

[20] TLA+ project. **Model Checking.** TLA+ Toolbox User's Guide, consulted 4 October 2026. Documentation.

[21] Alloy project. **Alloy Language Specification.** Official reference, consulted 4 October 2026. Specification.

[22] ProB project. **ProB Animator and Model Checker.** Official technical overview, consulted 4 October 2026. Documentation.

[23] Frama-C project. **ANSI/ISO C Specification Language.** Official ACSL overview, consulted 4 October 2026. Documentation.

[24] OpenJML project. **OpenJML.** Official Java verification overview, consulted 4 October 2026. Documentation.

[25] Verus project. **Verus Guide.** Official Rust verification documentation, consulted 4 October 2026. Guide.

[26] Schneider, F. B. (2000). **Enforceable Security Policies.** ACM Transactions on Information and System Security 3(1), 30-50. Author copy.

[27] Agarwal, S., Singhal, A., Breaux, T., and Vasilescu, B. (2026). **SpecMine: A Large-Scale Corpus of Spec-Driven Development Artifacts.** arXiv:2608.25202, version 3, 1 September 2026. Preprint and dataset account; repository counts are not production-adoption estimates. Paper.

[28] GitHub Spec Kit project. **Specification-Driven Development** and **Contract-Driven Development.** Official project documentation, consulted 4 October 2026. Development philosophy, Contract guide.

[29] Notion. **Wikis & Verified Pages** and **Custom Agents.** Official product documentation, consulted 4 October 2026. Wiki documentation, Custom Agent documentation.

[30] Karpathy, A. (2026). **LLM Wiki.** Author's description of an agent-maintained knowledge-base pattern, consulted 4 October 2026. Original gist.

[31] Gervasi, V., Ferrari, A., Zowghi, D., and Spoletini, P. (2019). **Ambiguity in Requirements Engineering: Towards a Unifying Framework.** In From Software Engineering to Formal Methods and Tools, and Back. Author manuscript.

[32] Fischbach, J., Frattini, J., Mendez, D., Unterkalmsteiner, M., Femmer, H., and Vogelsang, A. (2021). **How Do Practitioners Interpret Conditionals in Requirements?** Product-Focused Software Process Improvement, PROFES 2021, 85-102. Author preprint.

[33] Liu, A., Wu, Z., Michael, J., Suhr, A., West, P., Koller, A., Swayamdipta, S., Smith, N. A., and Choi, Y. (2023). **We're Afraid Language Models Aren't Modeling Ambiguity.** Proceedings of EMNLP 2023, 790-807. DOI: 10.18653/v1/2023.emnlp-main.51. Paper and publication record.

[34] Laban, P., Hayashi, H., Zhou, Y., and Neville, J. (2025). **LLMs Get Lost In Multi-Turn Conversation.** arXiv:2505.06120, version 1. Simulation study; cited at the scope of its evaluated tasks and models. Paper.

[35] Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., and Dutle, A. (2022). **A Compositional Proof Framework for FRETish Requirements.** Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2022. DOI: 10.1145/3497775.3503685. Author manuscript.

[36] Luckcuck, M., Farrell, M., and Sheridan, O. (2022). **Why just FRET when you can Refactor? Retuning FRETISH Requirements.** arXiv:2202.05816, version 1. Paper.

[37] Kuhn, T. (2014). **A Survey and Classification of Controlled Natural Languages.** Computational Linguistics 40(1), 121-170. DOI: 10.1162/COLI_a_00168. Paper and publication record.

[38] Kuhn, T. (2013). **The Understandability of OWL Statements in Controlled English.** Semantic Web 4(1), 101-115. DOI: 10.3233/SW-2012-0063. The linked author manuscript is dated 2012. Author manuscript.

[39] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). **How Amazon Web Services Uses Formal Methods.** Communications of the ACM 58(4), 66-73. DOI: 10.1145/2699417. Industrial experience report. Full paper.

[40] Murphy, G. C., Notkin, D., and Sullivan, K. (1995). **Software Reflexion Models: Bridging the Gap Between Source and High-Level Models.** Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, 18-28. DOI: 10.1145/222124.222136. Author's publication page and paper.

[41] Kuhn, T., and Bergel, A. (2014). **Verifiable Source Code Documentation in Controlled Natural Language.** Science of Computer Programming 96, Part 1, 121-140. DOI: 10.1016/j.scico.2014.01.002. The linked author preprint is dated 2013. Author preprint, Publication record.

[42] Object Management Group (2021). **Semantics of a Foundational Subset for Executable UML Models, Version 1.5.** Normative specification and machine-readable models; consulted 4 October 2026. Specification.

[43] Sneed, H. M., and Verhoef, C. (2020). **From COBOL to Business Rules: Extracting Business Rules from Legacy Code.** In Integrating Research and Practice in Software Engineering, Studies in Computational Intelligence 851, 187-208. DOI: 10.1007/978-3-030-26574-8_14. Industrial rule-extraction report; cited for its rule-extraction activities, not established whole-method effectiveness. Publisher's paper via the authors' institution.

[44] Garg, P., Löding, C., Madhusudan, P., and Neider, D. (2014). **ICE: A Robust Framework for Learning Invariants.** Computer Aided Verification, CAV 2014, Lecture Notes in Computer Science 8559, 69-87. DOI: 10.1007/978-3-319-08867-9_5. Author copy.

[45] Guliyev, U. **Signal research and development repository.** Private repository, snapshot 64829b9614df4e929fd11a59c88d1b44e31b5a3d, dated 11 June 2026; inspected 4 October 2026. Static inspection of documentation and selected implementation paths; no execution or reproduction of the reported agent comparison. Access requires repository permission. Repository snapshot, language specification, finalization specification, closure-gate implementation.

[46] Guliyev, U. (2026). **Firsthand account of semantic eduction and agent alignment in Signal.** Account and clarifications supplied for this whitepaper on 4 October 2026. Reports a comparison involving six GPT-6 Astra agent instances at Max reasoning; outcomes concerned both written interpretations and implementations. The Lean foundation and original comparison records were not inspected. Cited as practitioner testimony, with the evidential scope explained in Sections 9.6 and 9.7.

[47] Wei, R., Foster, S., Mei, H., Yan, F., Yang, R., Habli, I., O'Halloran, C., Tudor, N., and Kelly, T. (2024). **ACCESS: Assurance Case Centric Engineering of Safety-critical Systems.** arXiv:2403.15236, version 1. Methodology, tool support and autonomous-underwater-vehicle case study. Author manuscript.

[48] Necula, G. C. (1997). **Proof-Carrying Code.** Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, 106-119. DOI: 10.1145/263699.263712. Author's account of the approach, author's publication record.

[49] Abadi, M., and Lamport, L. (1993). **Composing Specifications.** ACM Transactions on Programming Languages and Systems 15(1), 73-132. Author manuscript.

[50] Frama-C. **WP: deductive verification of ACSL contracts.** Official plugin documentation, consulted 4 October 2026. Documentation.

[51] Brooks, F. P., Jr. (1986). **No Silver Bullet: Essence and Accidents of Software Engineering.** University of North Carolina at Chapel Hill technical report TR86-020, September 1986. Cited for the distinction between conceptual and representational difficulties, and the consequences of abstraction. Author report.

[52] Sutton, R. S. (2019). **The Bitter Lesson.** Essay, 13 March 2019. Historical argument for scalable search and learning in AI. Primary text preserved by the University of Texas at Austin.

[53] Singh, R., Joel, S., Mehrotra, A., Wadhwa, N., Bairi, R. B., Kanade, A., and Natarajan, N. (2025). **Code Researcher: Deep Research Agent for Large Systems Code and Commit History.** Microsoft Research technical report. Cited for automated code exploration and the stated limits of crash-based validation. Report.