Archetypes as Enterprise Primitives: Why Form Beats Function for AI-Assisted Development

Ramón Labbé

May 2026 • Paper 004

Context. *AI-assisted software development has crossed the threshold of functional competence. Commercial models in 2026 (OpenAI, 2025; Anthropic, 2025; Google DeepMind, 2025) produce code that compiles, tests that run, and APIs that ship; the open question for large organisations is no longer whether a model can write a feature but whether an organisation can integrate that capability into its delivery practice without losing the properties — consistency, auditability, repeatability, skill transfer — that make software engineering scale beyond the individual developer.* ***Problem.*** *The prompt — the object the AI tooling industry has defaulted to — is the wrong unit for those properties. A prompt is private to the person who writes it, non-versioned unless the user chooses, non-composable with other prompts, and not independently review-able. A large organisation consisting of many teams each operating many prompts has the same accretion of implicit, drifting knowledge that pre-modern software teams accumulated in tribal form before specifications were writeable.* ***Method.*** *We compare three objects of governance around LLM-assisted development — the prompt, the code template, and the declarative archetype — against six properties that an enterprise AI programme should be able to assert: repeatability, composability, review-ability, version-ability, skill-transferability, and audit-traceability. We proceed by conceptual analysis drawing on prior work in Design by Contract (Meyer, 1992), refinement types (Freeman and Pfenning, 1991; Rondon et al., 2008), architectural specification (Lamport, 1994; Taylor et al., 2009), and domain-driven design (Evans, 2003).* ***Key result.*** *The prompt satisfies zero of the six properties. The code template satisfies four. The declarative archetype, as implemented in AgentGuard and documented by Labbé (2026a), satisfies all six. The analysis does not depend on AgentGuard specifically; any declarative-specification substrate with a fixed schema would reach the same conclusion.* ***Conclusion.*** *For enterprise adoption, the unit of AI governance should be a declarative archetype, not a prompt. This argument closes a body of companion work by the author that began with the mechanism (Labbé, 2026c) and worked outward through measurement (Labbé, 2026b) and architecture (Labbé, 2026a). The thesis this work has been building toward is here stated plainly: LLMs will always solve the function; the discipline of form is what makes them scale.*

Keywords: AI-assisted development, enterprise software, design by contract, specification-driven development, maintainability, governance

1. Introduction

Every generation of software tooling eventually discovers that the obstacle to scale is not capability but form. Early compilers could translate any program; organisational scale came from coding conventions (Brooks, 1987). Early web frameworks could serve any page; organisational scale came from directory layouts, naming, and dependency discipline (Fowler, 1999; Evans, 2003). The same pattern is unfolding in AI-assisted development. The 2022–2024 cohort of tooling (GitHub, 2021; Anysphere, 2024; Anthropic, 2024) demonstrated that LLMs can write the function; the 2025–2026 enterprise-adoption wave is running into the organisational form question, and the industry has not yet named a primitive at the right level to answer it.

This paper makes that naming explicit. We propose that the primitive is the *declarative archetype*: a version-controlled, schema-validated document that states, before any generation happens, what the artefact to be produced must look like, what quality criteria it must satisfy, and what its structure must be. A prompt is not this; a template is closer but falls short; an archetype, as deployed in AgentGuard (Labbé, 2026a), is.

The function / form distinction

Let us stipulate an informal definition for the argument that follows. *Function* is whether the produced artefact does what was asked. *Form* is whether the produced artefact looks like it belongs in the set of other artefacts the same organisation produces. An organisation that cares only about function ships features and cannot scale beyond the horizon at which any individual can hold the whole system in their head; every new hire learns the tribal conventions and drifts them. An organisation that cares about form ships features *and* preserves the properties that let software practice scale past the individual. No one doubts this for hand-written code. We argue that the same logic applies, unchanged, to AI-generated code — and that the prompt-level tooling dominant today does not equip organisations to assert it.

What this paper argues

Section 2 traces the prior art in specification-driven development and contract programming, which collectively establish that declarative specification ahead of implementation is a recurring, well-understood pattern for preserving form at scale. Section 3 defines six properties that an enterprise AI governance primitive should support and describes the comparative method. Section 4 applies the method to three candidate primitives — the prompt, the code template, and the declarative archetype — and reports how each scores. Section 5 names the limitations of the argument and three classes of pushback a reader might reasonably raise.

Contributions

- A named framing — function / form — for the current bottleneck of enterprise AI-assisted development, with references to the prior instances of the pattern in software history.
- Six concrete properties (repeatability, composability, review-ability, version-ability, skill-transferability, audit-traceability) that operationalise the claim "supports form at enterprise scale", each stated in enough detail that an auditor could check it.
- A comparative analysis of the prompt, the code template, and the declarative archetype against those six properties, establishing that the archetype is the minimum primitive that satisfies all six.
- A principled linkage between the archetype primitive and the broader contract-programming and specification-driven traditions, grounding the claim in four decades of software engineering research rather than presenting it as a novel invention.

What this paper is not

It is not a tool review of AgentGuard, though the argument is informed by how AgentGuard implements archetypes. The argument does not depend on AgentGuard: a different declarative-specification substrate with a fixed schema and a review-able document format would satisfy the same six properties. The paper is also not a claim that archetypes are sufficient for enterprise AI governance; they are necessary, not sufficient, and Section 5 names several orthogonal concerns (security review, data governance, deployment provenance) that are entirely out of scope here.

2. Related Work

We organise related work by topic. The consistent thread across the topics is that *specifying the artefact before producing it* recurs as a mechanism whenever a software-engineering discipline graduates from individual-scale to organisation-scale, under many names and with many technical substrates.

2.1 Contract programming

Design by Contract, formulated by Meyer (1992) and realised in Eiffel, raised preconditions, postconditions, and invariants to first-class language constructs. A method's contract is inspectable, testable, and versioned alongside the method itself. The discipline scaled Eiffel teams beyond the size at which tribal convention held. Behavioural subtyping (Liskov and Wing, 1994) generalised the concept across inheritance hierarchies. The JML (Leavens et al., 2006) and Spec# (Barnett et al., 2005) projects brought the discipline to Java and C#. All share the thesis that an explicit contract is the artefact that lets code scale across contributors.

2.2 Refinement types and typed specifications

Refinement types (Freeman and Pfenning, 1991) generalise the type system to predicates, so that a type is not only "integer" but "non-negative integer less than the array length". Liquid Types (Rondon et al., 2008) automated inference of such predicates. Dependent types (Xi and Pfenning, 1998; The Coq Development Team, 2024) take the idea further, permitting types that depend on values. The throughline with the present paper is that a type is a contract: a declared property the implementation must satisfy, which is inspectable independently of the implementation.

2.3 Specification-based verification

TLA+ (Lamport, 1994) specifies system behaviour at the state-transition level, independently of implementation, and has been adopted in production by AWS (Newcombe et al., 2015) and Azure. Alloy (Jackson, 2012) specifies at the structural level with a relational logic. SPIN (Holzmann, 2003) verifies concurrent protocols. These tools do not generate code; they generate assurance that the specified behaviour is consistent and that the code can be checked against it. The shared premise is that a specification is *worth writing down* even though it does not execute.

2.4 Architecture description languages

Architecture Description Languages (ADLs) such as Wright, Rapide, and AADL (Taylor et al., 2009) specify a system's decomposition, the components, and the connectors between them. They have had uneven adoption in industry, with the successful instances (notably AADL in avionics) concentrated in regulated domains. The observation we borrow is that a decomposition, written down and version-controlled, is a governance object: it outlives the engineers who wrote it.

2.5 Domain-driven design and ubiquitous language

Evans (2003) articulated the "ubiquitous language" idea: that a shared vocabulary between domain experts and engineers, encoded into the code itself, is a durable enterprise asset. The method succeeds when the domain vocabulary is present as a first-class structural element, not only as comment or convention. An archetype's `tech_stack` and `structure` blocks are the AI-era analogue: a written commitment that resolves ambiguity between "what an API project means here" and "what an API project means in the model's training distribution".

2.6 Property-based and specification-based testing

QuickCheck (Claessen and Hughes, 2000) and its descendants test code against declared properties rather than specific examples. Metamorphic testing (Chen et al., 2018) does something similar for systems where the oracle is not known. The shared pattern: *state a property, let the machine search for counterexamples.* An archetype's `self_challenge.criteria` play the same role: a declared property the generated artefact will be asked to satisfy.

2.7 The no-silver-bullet thread

Brooks's 1987 essay (Brooks, 1987) argued that the essential difficulty of software is the specification of behaviour, not the production of code against a specification. If this observation

was correct in 1987 it is *more* pertinent in 2026: a model that can produce code at the speed of the prompter has moved the bottleneck decisively to the specification. Archetypes are one response to that diagnosis.

2.8 Positioning

The present paper does not invent a new mechanism; it argues that a specific, already-implemented specification substrate — the declarative archetype with a fixed schema — is the minimum-viable governance primitive for AI-assisted development at enterprise scale. The claim is empirically grounded by Labbé (2026b), architecturally grounded by Labbé (2026a), and mechanistically grounded by Labbé (2026c).

3. Method

This is an argumentative paper rather than an empirical one; the *Methodology* section therefore defines the terms of the argument rather than a data-collection protocol. We state the six properties we claim an enterprise AI governance primitive must support, define each operationally enough that a reader could audit a candidate primitive against it, and then describe the three candidate primitives we will compare.

3.1 The six properties

A governance primitive for AI-assisted development, operating at enterprise scale, is useful to the extent it supports the following six properties. Each property is stated with a one-sentence definition and a concrete failure mode that would indicate the property is not met.

Repeatability. The same input produces the same output across runs, machines, and people. Failure mode: two engineers on the same team, starting from the same stated intent, produce two different artefacts, and neither can reconstruct exactly why.

Composability. Two primitives can be combined into a larger primitive with predictable combined behaviour. Failure mode: the intent "an API backend with our auth layer" cannot be expressed as the composition of "API backend" and "our auth layer"; each combination is written from scratch.

Review-ability. The primitive is an artefact that can be opened, read, critiqued, and changed by a person other than the author without executing anything. Failure mode: to understand what a primitive does, a reviewer must run it.

Version-ability. The primitive lives in version control with a meaningful diff semantics. Failure mode: the primitive's changes cannot be represented as a line-level diff; migration between versions is either invisible or undecidable.

Skill-transferability. A new engineer joining the organisation can learn the primitive well enough to use it within a measurable onboarding window. Failure mode: the knowledge lives only in the people who originally wrote the primitive and is not transmissible without personal instruction.

Audit-traceability. Given an artefact produced by the primitive, an auditor can determine which version of which primitive produced it, with which inputs, at what time. Failure mode: an artefact's provenance is reconstructible only by interrogating the author's memory.

The six are not orthogonal — version-ability is a precondition for audit-traceability, for example — but each names a distinct failure mode, and an enterprise programme that lacks any one will report the corresponding failure within a quarter of serious adoption.

3.2 The three candidate primitives

We compare three candidate primitives against the six properties.

The prompt. Natural-language input to an LLM, typically one or several paragraphs, describing the artefact to be produced. Prompts may be saved in a file, but there is no schema, no validation, and no guarantee of consistency across prompters. Current state of practice in most AI tooling (GitHub, 2021; Anysphere, 2024; Anthropic, 2024).

The code template. A scaffolded starter repository with conventional directories, stub files, and sometimes a scaffold-generation CLI (e.g. Cookiecutter (Roy Greenfeld, 2014), Yeoman (The Yeoman Team, 2023), the Rails generator (Ruby on Rails Team, 2024)). Templates are versioned and review-able; their limitation is that they produce a fixed starting point and then step aside, leaving the subsequent AI-assisted generation unspecified.

The declarative archetype. A schema-validated YAML document stating the tech stack, directory structure, `self_challenge` criteria, and `validation` checks, which drives an entire generation pipeline. The substrate instantiated in AgentGuard (Labbé, 2026a); our argument generalises to any declarative-specification substrate with a fixed schema.

3.3 The comparative method

For each property we state the question an auditor would ask ("can two people on the same team produce the same artefact from the same input?"), apply it to each of the three candidate primitives, and record a pass/fail with a one-sentence rationale. No attempt is made to quantify degree; the output of the comparison is a 3-by-6 table of pass/fail with rationale, reported in Section 4.

We expect, and welcome, disagreement on individual cells; our goal is to make the judgment itself a review-able artefact. A reader who disagrees with a particular pass/fail can substitute their own and re-run the argument; the six properties themselves are the durable part of the contribution.

4. Comparative Analysis

We apply the method of Section 3 to the three candidate primitives. Table 1 summarises the result; the prose paragraphs below give the per-cell rationale, without interpretation (interpretation belongs in Section 5).

Table 1: Pass/fail per primitive against six enterprise-governance properties. "✓" indicates the property is supported by construction; "×" indicates it is not.

Property	Prompt	Template	Archetype
Repeatability	×	✓	✓
Composability	×	partial	✓
Review-ability	×	✓	✓
Version-ability	×	✓	✓
Skill-transferability	×	✓	✓
Audit-traceability	×	×	✓

4.1 Repeatability

Prompt. Fails. Natural-language prompts carry implicit assumptions that are not preserved across authors or re-prompts. Two engineers asking an LLM to "build a Python API backend with our auth" will produce materially different artefacts. The prompt-text itself may be saved, but the conversation context, prior turns, and model-internal state are not.

Template. Passes. A template instantiation at the same version with the same parameters produces the same starter repo byte-for-byte. The subsequent AI-assisted customisation may vary, but the template-level artefact is deterministic.

Archetype. Passes. The archetype document is content-addressed by hash (rlabs, 2026c); re-running the pipeline against the same archetype and specification yields the same response modulo model sampler variation, and the quality contract enforces a consistent rubric at every stage.

4.2 Composability

Prompt. Fails. "API backend" and "our auth layer" as prompts do not compose; the result of invoking both is not the result of the composition of their individual outputs, and there is no formal join operation.

Template. Partial. Cookiecutter-style templates support sub-templating with limited success; the limitation is that two independently-authored templates are likely to disagree on directory layout, naming conventions, and dependency versions.

Archetype. Passes. Archetypes declare their `structure` and `tech_stack` explicitly; two archetypes can be composed by schema-level merge of their declarations with conflict resolution rules defined at schema level. The current AgentGuard implementation supports archetype inheritance; full cross-tree composition is a documented design direction (Labbé, 2026a).

4.3 Review-ability

Prompt. Fails. A prompt's effect depends on the conversation history, the model version, and sometimes invisible system prompts. A reviewer cannot predict the prompt's effect by reading the prompt.

Template. Passes. Templates are readable directory trees with literal files.

Archetype. Passes. Archetypes are YAML documents with a fixed schema; a reviewer can read the `self_challenge` criteria and predict what the generated artefact will be evaluated against, without executing the pipeline.

4.4 Version-ability

Prompt. Fails. Prompts can be stored in files, but their meaningful semantics depend on untracked context; line-level diffs of a prompt do not faithfully track behaviour change.

Template. Passes. Templates are directories of files under standard version control.

Archetype. Passes. Archetypes are text documents; semantic versions track schema-level changes, and the schema validator rejects malformed versions at load time.

4.5 Skill-transferability

Prompt. Fails. Effective prompting is a tacit skill that current tooling gives few tools for transmitting across engineers; prompt galleries and team-internal cookbooks help at the margin but do not amount to a teachable discipline.

Template. Passes. Understanding a template requires understanding its directory conventions, its parameter surface, and the scaffold CLI that drives it — all learnable from documentation.

Archetype. Passes. An archetype is a YAML document with a schema; onboarding reduces to reading the schema specification and a handful of examples. The AgentGuard onboarding flow documents this explicitly (rlabs, 2026b).

4.6 Audit-traceability

Prompt. Fails. A generated artefact's provenance (which prompter, what model version, at what time, with what context) is not reliably recoverable after the fact.

Template. Fails. A template produces a starter; subsequent edits lose the template-provenance link. Trackers like Cruft (Cruft contributors, 2024) attempt to preserve the link but are not standard practice.

Archetype. Passes. Every pipeline invocation records the archetype's content hash, the specification, and the model identifier per stage, in a session store (Labbé, 2026a). An auditor can reconstruct which archetype version produced which artefact, with what inputs, at what time.

4.7 Summary of the 3-by-6 table

The prompt satisfies zero of the six properties. The template satisfies four (failing composability and audit-traceability). The archetype satisfies all six.

5. Discussion

5.1 Interpretation of the 3-by-6 table

The table in Section 4 reads cleanly in one direction: the archetype dominates the prompt and the template on every axis, losing nothing. A reader might reasonably ask why archetypes have not, in 2026, displaced prompts as the default unit of AI-assisted development tooling. Three answers co-exist.

First, the absence of schema. Until a schema exists, a declarative substrate is a suggestion, not a contract. The AgentGuard schema (rlabs, 2026a) is one concrete proposal; others are possible, and a mature market will likely standardise on a small set.

Second, the cost of authorship. Writing an archetype is more expensive than writing a prompt, in the same way that writing an Eiffel class with contracts is more expensive than writing a Python class with docstrings. The cost is paid by the author and refunded to every subsequent consumer; in an organisation of N engineers, the refund ratio scales with N. Small organisations and individuals reasonably choose the prompt; the thesis of this paper is that enterprise-scale organisations should not.

Third, the absence of review culture. A prompt is submitted privately; an archetype is merged into a shared repository with a review. Organisations without a review culture for AI artefacts do not yet have a slot where the archetype lives. The architecture work of Labbé (2026a) makes archetypes slottable into existing Git-based review flows; the culture question is orthogonal and out of scope for this paper.

5.2 What the argument does not prove

The argument establishes that the archetype is *necessary* for enterprise AI governance, not that it is *sufficient*. An enterprise AI programme additionally requires, at minimum: a security review process for generated code, a data-governance policy for training-data provenance, an evaluation suite independent of the self-challenge criteria, a deployment-provenance chain, and a change-management discipline for model-vendor updates. None of these are produced by archetypes per se; archetypes are one piece of a larger enterprise machinery.

5.3 Expected pushback

We anticipate three classes of reasonable disagreement.

"This is just scaffolding." An informed reader may claim that archetypes are a dressed-up version of Cookiecutter or Yeoman. The distinction we draw is the persistence of the archetype through generation — scaffolding steps aside after instantiation; the archetype remains as the rubric against which every stage of generation is evaluated (Labbé, 2026c). A template produces one artefact and is done; an archetype produces an artefact and *is the reason that artefact looks the way it does.*

"Models will solve form on their own." A frontier-model optimist may argue that models trained with sufficient post-training feedback will eventually produce artefacts in consistent enterprise form without explicit declaration. We do not dispute the possibility; we argue that enterprise adoption does not wait for that possibility to materialise. The archetype substrate lets an organisation assert form today, whether or not model capability converges to it.

"We have this covered with code review." A traditional software engineer may argue that review of generated code at merge time is sufficient and archetypes are over-engineering. The counter-argument is that review at merge time catches failures of form only *after* they are written; the archetype's quality-contract mechanism (Labbé, 2026c) catches them during generation. Prevention is cheaper than detection; the empirical measurement of the cost difference is the subject of Labbé (2026b).

5.4 Threats to Validity

Internal validity. The six properties are stated operationally but not formally. A reader could propose additional properties (latency, cost, tooling ecosystem) that would shift the comparative table. We chose the six because they are the properties at which enterprise AI adoption has, in practice, broken down; other selections are defensible.

External validity. The argument generalises best to typed, structured software artefacts (APIs, libraries, scripts). Its purchase is weaker on unstructured artefacts (prose, diagrams, copy) and unclear on artefacts with strong domain conventions not yet encoded in schemas (data pipelines, ML training loops). The archetype concept is extensible to these domains; the comparative advantage over prompts may not be as stark.

Construct validity. "Form" and "function" are informal terms. A sceptical reader can reject the dichotomy as arbitrary; we defend it on the narrower ground that the two properties can be assessed independently (a functionally correct artefact can violate organisational form; a form-respecting artefact can be functionally wrong) and that their decoupling is the phenomenon enterprise AI teams report in practice.

Selection bias. The author's organisation builds and sells one implementation of the archetype primitive. A sceptical reader should weight the argument accordingly. Our defence is that the argument is stated at the primitive level, not the product level: a different vendor's or an open-source archetype substrate with a fixed schema would reach the same conclusion about the 3-by-6 table.

6. Conclusion

We have argued that the unit of governance for AI-assisted development at enterprise scale is the declarative archetype, not the prompt. The argument is comparative: applied against six enterprise-governance properties (repeatability, composability, review-ability, version-ability, skill-transferability, audit-traceability), the prompt fails on all six, the code template passes four, and the archetype passes all six. The asymmetry is not marginal; it is categorical, and it is grounded in four decades of prior art on contract programming, refinement types, specification-based verification, architecture description, and domain-driven design.

This argument is the closing step of a line of companion work. Labbé (2026c) described the mechanism — quality contracts as prevention — that makes archetypes effective at generation time. Labbé (2026b) measured the mechanism across frontier and local models on three archetypes with a reproducible protocol. Labbé (2026a) documented the four-stage pipeline architecture that implements the mechanism and supports the archetype substrate. The progression — mechanism to measurement to architecture to argument — is deliberate. Each piece stands alone but each refers its ground to the others; the thesis this work has been building toward is here stated in its sharpest form.

Function is solved; form is the bottleneck. Large language models have crossed the threshold at which they reliably produce functionally correct code for well-specified tasks. Enterprise adoption at scale is not bottlenecked by that capability. It is bottlenecked by the organisational-form questions that have recurred at every previous inflection in software tooling: repeatability across teams, consistency across releases, review and audit at the boundary between generated and hand-written code, and the governance that accumulates around them. The declarative archetype is not the only possible response to that diagnosis, but it is a sufficient one, it exists today, and its cost to adopt is measurable. This paper argues that enterprises should adopt it, and the preceding three papers provide the mechanism, the measurement, and the architecture that make the adoption concrete.

The work continues. A five-paper series is not planned, but a next paper — should the series continue — would report on the operational experience of running archetype-driven generation in production over a quarter or more, which is an empirical contribution the present four cannot yet make. The rest of the form-discipline agenda (security review, data governance, deployment provenance) is out of scope for this series but actively on the research roadmap. We publish the four papers together because they argue together; the individual reader is invited to disagree with any cell of the comparative table in Section 4 and to extend the benchmark of Labbé (2026b) with their own model. The artefacts are open; the thesis is a provocation rather than a conclusion.

References

Anthropic (2024). *Claude Code*. https://www.claude.com/claude-code.

— (2025). *Claude Sonnet 4.6*. https://www.anthropic.com.

Anysphere (2024). *Cursor — the AI-first code editor*. https://www.cursor.com.

Barnett, Mike, K. Rustan M. Leino, and Wolfram Schulte (2005). "The Spec# Programming System: An Overview". In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*.

Brooks, Frederick P. (1987). "No Silver Bullet — Essence and Accidents of Software Engineering". In: *IEEE Computer* 20.4, pp. 10–19.

Chen, Tsong Yueh, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou (2018). "Metamorphic Testing: A Review of Challenges and Opportunities". In: *ACM Computing Surveys* 51.1.

Claessen, Koen and John Hughes (2000). "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Cruft contributors (2024). *Cruft — Allows you to maintain all the necessary cruft for packaging and building projects separate from the code*. https://github.com/cruft/cruft.

Evans, Eric (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Freeman, Tim and Frank Pfenning (1991). "Refinement Types for ML". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

GitHub (2021). *GitHub Copilot — Your AI pair programmer*. https://github.com/features/copilot.

Google DeepMind (2025). *Gemini 2.5 Pro*. https://deepmind.google/technologies/gemini/.

Holzmann, Gerard J. (2003). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

Jackson, Daniel (2012). *Software Abstractions: Logic, Language, and Analysis*. MIT Press.

Labbé, Ramón (2026a). *Inside AgentGuard: A Four-Stage Pipeline for Structured LLM Generation*. https://papers.rlabs.cl/003-pipeline-architecture.pdf. Companion paper.

— (2026b). *Same Task, Different Models: Measuring Archetype-Driven Generation Across Local and Frontier LLMs*. https://papers.rlabs.cl/002-local-model-benchmark.pdf. Companion paper.

— (2026c). *Show the Rubric Before You Ask for the Work: Quality Contracts as a Prevention Layer in LLM Generation Pipelines*. https://papers.rlabs.cl/001-quality-contracts.pdf. Companion paper.

Lamport, Leslie (1994). "The Temporal Logic of Actions". In: *ACM Transactions on Programming Languages and Systems* 16.3, pp. 872–923.

Leavens, Gary T., Albert L. Baker, and Clyde Ruby (2006). "Preliminary Design of JML: A Behavioral Interface Specification Language for Java". In: *ACM SIGSOFT Software Engineering Notes* 31.3, pp. 1–38.

Liskov, Barbara H. and Jeannette M. Wing (1994). "A Behavioral Notion of Subtyping". In: *ACM Transactions on Programming Languages and Systems* 16.6, pp. 1811–1841.

Meyer, Bertrand (1992). "Applying Design by Contract". In: *IEEE Computer* 25.10, pp. 40–51.

Newcombe, Chris, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff (2015). "How Amazon Web Services Uses Formal Methods". In: *Communications of the ACM* 58.4, pp. 66–73.

OpenAI (2025). *GPT-5*. https://platform.openai.com/docs/models.

rlabs (2026a). *AgentGuard archetype schema (source)*. https://github.com/rlabs-cl/agentguard-lib/blob/main/agentguard/archetypes/schema.py.

— (2026b). *AgentGuard documentation — Learn*. https://agentguard.rlabs.cl/docs/learn.

— (2026c). *rlabs-agentguard (public source mirror, v0.13.0)*. https://github.com/rlabs-cl/agentguard-lib.

Rondon, Patrick M., Ming Kawaguchi, and Ranjit Jhala (2008). "Liquid Types". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Roy Greenfeld, Audrey (2014). *Cookiecutter: A command-line utility that creates projects from templates*. https://cookiecutter.readthedocs.io.

Ruby on Rails Team (2024). *Creating and Customizing Rails Generators & Templates*. https://guides.rubyonrails.org/generators.html.

Taylor, Richard N., Nenad Medvidović, and Eric M. Dashofy (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley.

The Coq Development Team (2024). *The Coq Proof Assistant Reference Manual*. https://coq.inria.fr/refman/.

The Yeoman Team (2023). *Yeoman — The web's scaffolding tool for modern webapps*. https://yeoman.io.

Xi, Hongwei and Frank Pfenning (1998). "Eliminating Array Bound Checking Through Dependent Types". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.