Inside AgentGuard:
A Four-Stage Pipeline for Structured LLM Generation

Ramón Labbé

May 2026 • Paper 003

Context. *Framework-level discussions of LLM-assisted code generation tend to treat a pipeline as a chain of prompts and leave architecture implicit. When a pipeline graduates from a demo to a production dependency, architectural decisions — what the boundary between stages is, what crosses it, what is persisted, what is idempotent — drive most of the operational behaviour a user eventually cares about.* ***Problem.*** *Readers of the companion work by Labbé (2026c) and Labbé (2026b) need to understand the structure of the specific pipeline whose mechanism the former describes and whose multi-model behaviour the latter measures. A prose description buried in a source tree is not enough; the design decisions — why four stages and not one or seven, why a declarative archetype rather than a code-native DSL, why sessions are optional — deserve an explicit statement because they shape every downstream claim.* ***Method.*** *We document the AgentGuard pipeline at version `0.13.0` of the public `rlabs-agentguard` package. For each of the four stages (`skeleton`, `contracts_and_wiring`, `logic`, `validate`) we describe inputs, outputs, carried state, failure modes, and the contract with neighbouring stages. We then position the architecture against three adjacent approaches — monolithic single-prompt generation, reflective self-critique loops, and graph-based multi-agent frameworks — and name the boundaries where the approaches diverge.* ***Key result.*** *The pipeline has exactly four externally observable stages and two cross-cutting services (session store and archetype registry). The four-stage decomposition is the minimal division that lets every stage carry the quality-contract block introduced by Labbé (2026c): fewer stages merge skeleton and contract decisions, which violates the "show the rubric before you ask for the work" principle; more stages introduce boundaries that do not correspond to a distinct artefact transition, inflating payload size (we measure approximately 374 tokens per contract-carrying stage) without a corresponding drop in terminal validation failures.* ***Conclusion.*** *The architecture is deliberately smaller than an agent framework and deliberately larger than a single prompt; the trade-off is intentional. We document the boundaries so the benchmark by Labbé (2026b) and the enterprise argument by Labbé (2026a) can be evaluated against a specific, named system rather than an abstract idea of "pipeline".*

Keywords: LLM pipelines, software architecture, code generation, agent frameworks, quality assurance

1. Introduction

The public discussion of LLM-assisted code generation operates at two levels. At one level, practitioners share prompt tricks, model comparisons, and feature requests; the unit of discussion is the prompt. At another level, researchers publish frameworks and agent architectures; the unit of discussion is a graph of agents with tools (Wu et al., 2023; Chase, 2024; Shinn et al., 2023). Both levels underspecify the middle ground in which most production code actually lives: a pipeline of a few specific stages with explicit contracts, session persistence, and a deterministic validation boundary.

This paper operates at that middle level. Labbé (2026c) introduced a mechanism — embedding an evaluation rubric in every generation stage — and reported its cost. Labbé (2026b) measures the mechanism across four models. Neither is fully interpretable without knowing where the stages live, what crosses their boundaries, and why the boundaries are where they are. We supply that missing context here.

What this paper does

Section 2 places the AgentGuard pipeline against three adjacent approaches: monolithic single-prompt generation, reflective self-critique loops, and graph-based multi-agent frame-

works. Section 3 documents the four pipeline stages (`skeleton`, `contracts_and_wiring`, `logic`, `validate`) with their inputs, outputs, carried state, contract with neighbouring stages, and failure modes. It also documents the two cross-cutting services (the session store and the archetype registry) and the boundary conditions for pause, resume, and replay. Section 4 is a qualitative design analysis: four concrete design choices and the alternatives we rejected. Section 5 names the limits of the architecture and the open questions we have not answered.

Contributions

- An explicit stage-by-stage documentation of the AgentGuard pipeline at version `0.13.0`, with inputs, outputs, carried state, and neighbouring-stage contracts per stage. A reader who later wants to build a competing pipeline has a reference architecture they can concretely diverge from.
- A named positioning against three adjacent approaches, with an articulation of the boundary at which each approach stops and AgentGuard continues (or vice versa). The positioning is concrete enough that a reader can decide whether the architecture's design choices match their own use case.
- A written-down justification for four stages, rather than one or three or seven. The argument is anchored in the quality-contract cost reported by Labbé (2026c) and connects to the benchmark protocol of Labbé (2026b).
- A set of boundary conditions (session persistence, stage idempotence, cross-stage contract) that an engineer adopting AgentGuard into a larger system needs to be aware of before integration, stated in one place rather than inferred from source.

What this paper is not

This is not a comparative benchmark against AutoGen, LangGraph, or CrewAI: Labbé (2026b) is the empirical companion. This is not a replacement for the source code or the API reference; it is complementary to both and assumes a reader who wants design rationale rather than invocation examples. And this is not a description of every feature of the AgentGuard platform — the marketplace, session state, team tier, and usage telemetry are out of scope for the generation-pipeline architecture, which is the entire subject of the paper.

2. Related Work

We organise related work by approach rather than chronologically, so that the architectural positioning of AgentGuard is legible by contrast.

2.1 Monolithic single-prompt generation

The default of LLM-assisted code generation is the single prompt: a user asks for an artefact and the model emits it end-to-end. Chain-of-thought prompting (Wei et al., 2022) remains a single prompt whose output happens to include intermediate reasoning. Retrieval-augmented variants (Lewis et al., 2020; Gao et al., 2023) enrich the prompt with context but do not split generation into stages. This approach wins on simplicity and loses on two axes: inspectability (there is no intermediate artefact to evaluate) and controllability (the user cannot intervene between "decide the file layout" and "fill in the function bodies"). The AgentGuard pipeline exchanges the simplicity of a single prompt for staged inspectability at deliberate cost.

2.2 Reflective self-critique and self-refine loops

Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and SelfCheck (Miao et al., 2023) take the opposite direction: same prompt, repeated critique-and-edit cycles until a stopping criterion. Self-consistency (Wang et al., 2023) samples the same prompt n times and picks the majority answer. These approaches are *post-hoc*: the rubric for evaluating the output is applied

after the output exists. The thesis of Labbé (2026c) is that applying the rubric after the fact is the expensive case; applying it before each stage *prevents* the correction cycle these methods recover from. AgentGuard does retain a terminal self-review (the `validate` stage), but the per-stage contracts mean most runs reach `validate` without a rewrite round.

2.3 Graph-based multi-agent frameworks

AutoGen (Wu et al., 2023), LangGraph (Chase, 2024), and CrewAI (Moura, 2024) generalise beyond a linear stage sequence to arbitrary graphs of agents. An agent has a role, a tool set, and a message protocol; the framework orchestrates conversations. These are strictly more expressive than a four-stage linear pipeline — you can encode a four-stage pipeline in any of them — and strictly more operationally complex in return. Two consequences matter for us. First, with an arbitrary graph, the concept of "the rubric the current stage is about to be evaluated against" is not well defined; there is no fixed sequence of stages. Second, the failure modes of a multi-agent system include agent disagreement, loop non-termination, and role-leakage — classes of failure that a linear pipeline cannot experience by construction. AgentGuard sits on the less-expressive, more-predictable side of this trade-off.

2.4 Declarative specification and contract-based software

The notion that a declarative specification should precede code has deep roots. Design by Contract (Meyer, 1992) made precondition / postcondition / invariant the first-class objects of a programming language (Eiffel). Refinement types (Freeman and Pfenning, 1991; Rondon et al., 2008) move the contract into the type system. TLA+ (Lamport, 1994) specifies behaviour at the model level. Architecture description languages (Taylor et al., 2009) specify a system's decomposition before implementation. The AgentGuard archetype is a declarative specification in this family: a YAML file states the tech stack, the expected structure, and the self-challenge criteria *before* the pipeline runs. The companion work of Labbé (2026a) argues that this archetype is the right granularity for enterprise AI; the contribution of the present paper is that the four-stage pipeline is the right execution substrate for it.

2.5 Model Context Protocol and external tools

AgentGuard's tooling integration is built on the Model Context Protocol (Anthropic, 2024). MCP supplies a transport-agnostic way for a model to invoke tools exposed by a host; we use it as the surface via which the four pipeline stages are exposed to the model. This is orthogonal to the architectural decisions we describe; we flag it here because the concrete code that exercises the pipeline calls MCP endpoints, and the reader replicating our setup will encounter it.

3. Architecture

We use the *Methodology* section of the IMRaD structure to document the architecture, because the IMRaD frame expects a section that a reader could use to reproduce the work, and in a design-rationale paper that maps to documenting the artefact itself.

3.1 Overview

The AgentGuard generation pipeline has four externally observable stages and two cross-cutting services. Figure **??** sketches the flow; each stage is documented in the subsections below.

`skeleton` Takes a specification plus an archetype name. Returns a file-tree proposal with per-file responsibilities and the archetype's quality contract.

`contracts_and_wiring` Takes the skeleton plus the specification. Returns a declaration of every public function that needs to exist, its signature, its pre- and post-conditions, and the import graph that wires modules together. Carries the quality contract.

`logic` Takes a single function declaration and produces its body. Called once per declared function. Carries the quality contract scoped to that function.
`validate` Takes the full generated artefact and returns a self-review against the archetype's declared criteria, structural checks, and validation checks.

Two services cut across the four stages:
Archetype registry A declarative loader that resolves an archetype name (built-in or marketplace) to a YAML specification of `tech_stack`, `structure`, `self_challenge`, and `validation`.
Session store Optional; when enabled, persists the running state of a pipeline invocation so it can be paused, resumed, or replayed deterministically.

3.2 Stage 1 — `skeleton`

Input. A natural-language specification of the artefact (one paragraph is typical), the archetype name, and an optional maturity flag that biases the skeleton toward minimal versus comprehensive file layouts.

Output. A structured JSON response: an ordered list of paths to create, one-line per-path responsibility description, and the quality contract scoped to skeleton criteria.

Carried state. The archetype reference and the specification hash. The skeleton response is stateless with respect to prior invocations; repeating the stage with the same inputs yields the same response up to model non-determinism.

Contract with the next stage. The paths in the skeleton are the file-level commitment that `contracts_and_wiring` must respect. If the next stage proposes a function living in a path not declared by `skeleton`, the pipeline flags a boundary violation and re-invokes `skeleton` with a rewrite directive.

Failure modes. Context-window overflow for very large specifications; JSON-mode parse failure if the model emits prose instead of structure; model-side refusal on specifications that trip content-policy filters.

3.3 Stage 2 — `contracts_and_wiring`

Input. The skeleton response plus the original specification.

Output. A list of function declarations, one per exported symbol, each carrying a signature, a prose contract (preconditions, postconditions, invariants), an exception taxonomy, and the module's import graph. The quality contract is carried.

Carried state. The skeleton response is replayed in the prompt so that declarations are consistent with the file layout.

Contract with the next stage. Every declaration is a unit of work for `logic`. The logic stage must produce a body that respects the signature and contract exactly; it cannot add parameters or change return types without escalating to a re-wiring round.

Failure modes. Signature-vs-contract contradiction (the declared return type cannot satisfy the postcondition); import-graph cycle; function-count mismatch against skeleton.

3.4 Stage 3 — `logic`

Input. One function declaration at a time from Stage 2.

Output. The body of the function, in the archetype's declared language, plus inline tests where the archetype requires them. The quality contract is carried, scoped to criteria that apply to function-body quality (e.g. "no bare `except:`", "all side effects go through declared adapters").

Carried state. The function's declaration and the declarations of any other functions in the same module that the current function calls, so the model can produce a body consistent with its siblings.

Contract with the next stage. Each logic output must match its declaration byte-for-byte at the signature boundary. `validate` assembles all logic outputs into a coherent source tree and applies the archetype's structural and syntactic checks.

Failure modes. The body violates a declared precondition's exception; the body introduces a dependency not listed in Stage 2's import graph; the body is a stub (`raise NotImplementedError`) because the model could not infer behaviour from the declaration.

3.5 Stage 4 — `validate`

Input. The full generated source tree.

Output. A self-review report: for each criterion in the archetype's `self_challenge.criteria`, a pass/fail and a one-line rationale. Structural checks (every file declared in `structure` exists; no file outside the declared layout) are applied deterministically. Syntactic checks are applied via the archetype's declared parser.

Carried state. None; validate is deliberately stateless so that it is an independent check rather than a rubber-stamp on prior decisions.

Contract with the caller. A boolean overall pass plus a structured report. The caller may re-enter the pipeline with a rewrite directive targeted at specific failures; the re-entry is at `logic` for per-function failures or at `contracts_and_wiring` for declaration-level failures.

Failure modes. False-positive pass (the model's self-score is generous); structural check failure due to a file renamed by `logic`; parser error on syntactically invalid code.

3.6 Cross-cutting: archetype registry

Archetypes are YAML documents with a fixed schema (rlabs, 2026a). They declare the tech stack (language, framework, database, testing), the expected directory structure, the self-challenge criteria, and the validation checks. The registry resolves names to archetype documents and caches loaded archetypes in memory. Two classes of archetype exist: built-in (shipped with the `rlabs-agentguard` package) and marketplace (downloaded on demand, subject to purchase).

3.7 Cross-cutting: session store

When enabled, the session store persists the running state of a pipeline invocation: the specification, the skeleton response, the contracts-and-wiring response, the per-function logic responses, and the validate report. This enables three behaviours: pause (stop after any stage, resume later), replay (re-run a stage deterministically against the stored prior state), and fork (create a branch where one stage is re-run with a different prompt or a different model). The session store is optional because the stages are individually idempotent given their inputs; sessions exist to make the idempotence practically accessible, not to make it possible.

3.8 Boundary conditions

Three boundary conditions govern when the pipeline can pause, resume, or replay deterministically. First, every stage must be idempotent modulo model non-determinism: repeating a stage with the same inputs produces the same output up to sampler-level variation. Second, cross-stage state must be explicit: anything that matters downstream must appear in the stage's declared output, not in model-internal context. Third, the archetype document must be version-pinned: if the archetype changes between pause and resume, resume is no longer deterministic. The architecture enforces the first and third conditions by construction (stages are pure functions of their inputs; archetypes are loaded by content hash); the second is enforced by the fixed output schema of each stage.

4. Design Analysis

A design-rationale paper uses the IMRaD *Results* slot to present the design analysis — the concrete decisions that the architecture embodies, and the alternatives explicitly rejected. Four decisions merit individual treatment; we present each with the alternative we considered and the reason we chose as we did, without interpreting the choices in terms of general principles (interpretation belongs in Section 5).

4.1 Why four stages

A pipeline could have one stage (monolithic prompt), three stages (skeleton, body, validate), or n stages for an arbitrary n. We chose four.

Rejected: monolithic. Monolithic generation forfeits the rubric-prevention mechanism of Labbé (2026c) entirely: there is no boundary at which to present the rubric before generation. Every failure becomes a post-hoc correction round.

Rejected: three stages (skeleton, body, validate). A three-stage version merges function declarations into either the skeleton or the body. Merging into the skeleton inflates the skeleton response (now carrying signatures, preconditions, and import graphs) to the point where the token budget for the quality contract competes with the structural declaration; in pilot experiments we observed contract clipping at this scale. Merging into the body entangles function-level decisions (what does this function do) with artefact-level decisions (which functions exist at all), producing inconsistent artefacts where the set of declared functions drifts mid-generation.

Rejected: more than four. Adding a fifth stage — examples we considered were "prompt engineering" between skeleton and contracts, or "refactor" between logic and validate — introduces a boundary that does not correspond to a distinct artefact transition. The boundary must be paid for in both cross-stage contract complexity and per-stage contract overhead (approximately 374 tokens, per Labbé, 2026c). The pay-off of a fifth stage would have to exceed this cost; we could not identify a pay-off that consistently does.

Chosen: four. The four-stage decomposition corresponds to four distinct artefact transitions: specification to file tree (`skeleton`), file tree to declarations (`contracts_and_wiring`), declaration to body (`logic`, one per function), and complete tree to verdict (`validate`). Every boundary is a place where a human reviewer — or a re-entry rewrite — has a legible intermediate artefact.

4.2 Why declarative archetypes rather than code

Archetypes could have been Python classes or a DSL embedded in Python. They are instead YAML documents.

Rejected: Python classes. A Python-defined archetype ties authorship to Python engineers and couples the archetype's lifecycle to the interpreter. The marketplace depends on archetypes being inspectable and diff-able without execution; arbitrary code fails that bar.

Rejected: embedded DSL. An embedded DSL (in the Racket / Rhombus lineage) preserves declarativity while gaining expressivity but requires a toolchain that most contributors do not have. The user research behind (Labbé, 2026a) suggests that the authorship population of a useful archetype library is broader than the developer population of the host language; YAML lowers the barrier without giving up the declarativity that makes the artefact review-able.

Chosen: YAML with a schema. The cost is that expressivity is bounded by the schema; complex archetypes have to be factored rather than scripted. The benefit is that the entire corpus of archetypes can be rendered, diffed, reviewed, and diffused without a code environment.

4.3 Why the session store is optional

A system where the session store is mandatory is simpler in one way (every invocation persists) and harder in every other way (every deployment needs a database, every operation requires schema migration, every replay requires state-access concerns). We made the store optional.

Chosen: optional. Because each stage is pure modulo sampler variation, the session store is a performance-and- convenience feature, not a correctness feature. A user who wants pause/resume/replay pays the infrastructure cost; a user who does not, gets a simpler deployment at the cost of re-running stages deterministically-enough on a crash.

4.4 Why terminal validate remains

Given that every prior stage carries the quality contract, a design question we explicitly considered was whether the terminal `validate` stage is redundant.

Rejected: drop validate. In pilot runs, removing validate increased the false-acceptance rate: logic-stage bodies that individually satisfy their declared contract can, in aggregate, violate a whole-artefact criterion (for example, "every public function has a unit test" is not checkable by any single logic call). Validate remains as the whole-artefact check.

Chosen: retain validate as reinforcement. The relationship between pre-contract stages and validate is reinforcement, not duplication: pre-contract stages prevent per-component failures, validate catches whole-artefact failures. Labbé (2026c) makes this claim at the mechanism level; the four-stage architecture embodies it at the implementation level.

5. Discussion

5.1 What the architecture does well

A four-stage linear pipeline with declared artefact boundaries and an optional session store occupies a usefully narrow band of the LLM-orchestration design space. It inherits from monolithic prompts the operational simplicity of a linear call graph — no agent scheduling, no loop non-termination, no role leakage — while buying back from multi-agent frameworks the inspectability of intermediate artefacts. The boundary conditions (Section 3) give deployment teams a small set of invariants to reason about during integration; this is cheaper than reasoning about an emergent agent graph, and the empirical pay-off is the lower wall-clock time to a passing artefact that Labbé (2026b) measures.

5.2 What the architecture does not do

The architecture is not a general-purpose agent framework. It does not support open-ended tool use during generation: a stage either completes with the declared contract or fails. It does not support reflection loops between stages: the rewrite pathway is coarse-grained and re-enters at a whole stage, not at an intermediate thought. It does not support parallel agents with disagreement resolution: there is exactly one model active per stage. Each of these omissions is by design and each is a limitation for a class of use case that AgentGuard is not aimed at.

5.3 Open design questions

Granularity of `logic`. The current pipeline invokes `logic` once per declared function. For archetypes with many small functions this is a lot of round trips; for archetypes with few large functions it is a lot of context per trip. A planned variant (Labbé, 2026c) shards large functions by `applies_when` predicates, which the present pipeline does not yet support.

Cross-session reuse. If two generation runs share a specification, they share exactly zero computed state today. A future pipeline could memoise at the stage boundary: if two runs produce the same skeleton response, the downstream stages could short-circuit on cache hits. We have not built this; the design question is whether the saved cost is worth the complexity of a content-addressable intermediate store.

Multi-model per-stage routing. The current pipeline uses a single model for all four stages. A plausible refinement is routing: use a strong model for `skeleton` and `contracts_and_wiring`, a cheap model for `logic` bodies, and a strong model again for `validate`. The benchmark of Labbé (2026b) gives the evidence base to decide the routing; the architecture does not today expose routing as a first-class concept.

5.4 Threats to Validity

Internal validity. This paper describes a specific version (`0.13.0`) of a specific pipeline. Future versions will move the boundaries described here; the abstract claim about four stages being the right count is an artefact of the current mechanism and may not survive the planned shard-by-predicate variant.

External validity. The pipeline's design is fit for archetype-driven generation of structured software artefacts in typed languages. It is unclear how much of the architecture survives at the boundary of unstructured natural-language generation (where there is no `structure` block to validate against) or of unstructured exploratory coding (where the user does not know the skeleton at the start).

Construct validity. "Right granularity" and "minimal" are load-bearing terms in our justification of four stages. These terms are evaluated against the quality-contract cost reported by Labbé (2026c) and the wall-clock cost measured by Labbé (2026b), not against a general theorem about pipeline decomposition. A reader who cares about a different cost (e.g., latency at the last stage, or peak context-window usage mid-stage) will reach different conclusions and should re-run the analysis of Section 4 with that cost in mind.

Documentation-vs-code drift. A design-rationale paper is an artefact separate from the source code, and drifts from it over time. The paper corresponds to version `0.13.0`; reliance on the present description for versions beyond that is at the reader's risk, and for definitive behaviour the source and changelog are canonical (rlabs, 2026b).

6. Conclusion

We documented the AgentGuard generation pipeline at version `0.13.0`: four externally observable stages, two cross-cutting services, three boundary conditions for pause, resume, and replay, and four explicit design decisions with the alternatives we rejected. The architecture sits between the simplicity of monolithic prompting and the expressivity of agent graphs, and the boundary at which it diverges from each is named rather than left implicit.

The intended use of this paper is to make the claims of Labbé (2026c), Labbé (2026b), and Labbé (2026a) evaluable against a specific, documented system. The first describes a mechanism that requires distinct stage boundaries; we now have the stages. The second measures the mechanism across models; the measurement is legible because the thing being measured is described. The third argues for the enterprise consequences of measuring form alongside function; the artefact through which that form is measured is this pipeline.

Architectures evolve. The sharding-by-predicate variant, the multi-model routing variant, and the cross-session cache variant are three directions we are considering; each will move the boundaries we have named. When those moves happen, the companion paper will document them; the principle — make the stage boundaries explicit, make the contract at each boundary explicit, make the boundary conditions explicit — is intended to outlive the specific decomposition we describe here.

References

Anthropic (2024). *Model Context Protocol*. https://modelcontextprotocol.io/specification. Open protocol for LLM-tool integration.

Chase, Harrison (2024). *LangGraph: Stateful, Multi-Actor Applications with LLMs*. https://langchain-ai.github.io/langgraph/. Project documentation.

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

Gao, Yunfan, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang (2023). "Retrieval-Augmented Generation for Large Language Models: A Survey". In: *arXiv preprint arXiv:2312.10997*.

Labbé, Ramón (2026a). *Archetypes as Enterprise Primitives: Why Form Beats Function for AI-Assisted Development*. https://papers.rlabs.cl/004-archetypes-enterprise-primitives.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.

Lewis, Patrick, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Madaan, Aman, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark (2023). "Self-Refine: Iterative Refinement with Self-Feedback". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

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

Miao, Ning, Yee Whye Teh, and Tom Rainforth (2023). "SelfCheck: Using LLMs to Zero-Shot Check Their Own Step-by-Step Reasoning". In: *arXiv preprint arXiv:2308.00436*.

Moura, João (2024). *CrewAI: Framework for Orchestrating Role-Playing, Autonomous AI Agents*. https://github.com/crewAIInc/crewAI. Open-source project.

rlabs (2026a). *AgentGuard archetype schema (source)*. https://github.com/rlabs-cl/agentguard-lib/blob/main/agentguard/archetypes/schema.py. Canonical schema definition in the public `rlabs-agentguard` Python package.

— (2026b). *rlabs-agentguard (public source mirror, v0.13.0)*. https://github.com/rlabs-cl/agentguard-lib. Source of the architecture described in this paper.

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

Shinn, Noah, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao (2023). "Reflexion: Language Agents with Verbal Reinforcement Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

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

Wang, Xuezhi, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou (2023). "Self-Consistency Improves Chain of Thought Reasoning in Language Models". In: *International Conference on Learning Representations (ICLR)*.

Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Wu, Qingyun, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed H. Awadallah, Ryen W. White, Doug Burger, and Chi Wang (2023). "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework". In: *arXiv preprint arXiv:2308.08155*.