Show the Rubric Before You Ask for the Work: Quality Contracts as a Prevention Layer in LLM Generation Pipelines

Ramón Labbé

April 2026 • Paper 001

Multi-stage pipelines for large language model (LLM) generation of structured artefacts — code, documents, specifications — commonly layer a terminal validation step on top of a linear chain of generation prompts. In practice, the generating model encounters the evaluation rubric only at that terminal step, after the artefact is already written. We observed, during internal dogfooding of the AgentGuard pipeline, that long or multi-section artefacts drift away from the rubric in later sections because the criteria are out of attention while the model writes, producing a corrective rewrite loop that final validation surfaces too late to be cheap. We propose a `quality_contract` block emitted at the top of every generation stage, carrying the same self-challenge criteria, expected structure, and validation checks the terminal step will use. The generating model reads the contract before producing output at each stage, so final validation becomes reinforcement of a decision already made rather than the first encounter with the rubric. Measured over the ten built-in archetypes shipped with AgentGuard, the contract adds an average of 1,498 bytes (approximately 374 tokens at a four-byte-per-token heuristic) per stage response, a 29% share of total response size. We argue the token overhead is favourable compared to a single correction cycle on a long artefact, and that the principle — show the rubric at generation time, not only at grading time — generalises to other LLM pipelines with declared evaluation criteria. We report one observed dogfood case (a quarterly strategy memo) in which the pre-contract pipeline produced three violations at terminal validation while the contract design was hypothesised to have caught them during writing, though we explicitly flag this as observational not experimental evidence.

Keywords: LLM pipelines, code generation, quality assurance, prompt engineering, agent tooling

1. Introduction

Systems that use large language models (LLMs) to generate production-grade artefacts rarely rely on a single prompt. The mature pattern, whether the output is code (Cursor, 2024; Anthropic, 2024b), specifications (OpenAI, 2024b), or longer reasoning-heavy documents (Anthropic, 2024a), is a pipeline of narrower prompts where each stage produces intermediate state the next stage consumes. This is good engineering: it keeps individual prompts short enough for the model to attend to every instruction, avoids the compounding error of asking for everything at once, and lets the caller retry stages independently.

A common design choice within this pattern is to place evaluation against declared quality criteria at the terminal step. A final validator scores the finished artefact, reports failures, and optionally triggers a rewrite. This mirrors how unit tests are typically structured: write code, then run tests. It has the conceptual virtue of separation of concerns: the generator writes, the validator judges.

The problem we observed is that this separation is too clean. Under realistic conditions — a long artefact with several sections, a model whose attention window is finite and whose attention across that window is not uniform (Liu et al., 2024) — the rubric that will judge the artefact at the end is absent from the model's context while it writes later sections. Earlier sections written with a partial mental model of quality influence later sections; later sections drift further because the reference point is no longer in view. Terminal validation catches the drift, but each caught violation implies a rewrite, and the rewrite cost scales with how late in the artefact the drift appeared.

The specific trigger for this paper was an internal dogfood: we used our own pipeline, AgentGuard (rlabs, 2026c), to write a quarterly strategy memo via the `strategy_memo` archetype. The artefact was ten markdown files totalling roughly seven hundred lines of prose. Terminal validation reported three violations — two involving bullet-point formatting in middle sections, one involving a field label — that the model could plausibly have avoided given foreknowledge of the criteria.

We propose a change small enough to be described in one sentence: surface the evaluation criteria at the start of each generation stage, not only at the end. The rest of this paper formalises that change, describes its implementation in AgentGuard v0.13.0, measures its overhead on the ten built-in archetypes, and argues the principle generalises beyond AgentGuard to any LLM pipeline with declared criteria.

Contributions

The specific, verifiable contributions of this paper are:

1. A formal description of the `quality_contract` block — a compact, stage-independent payload carrying the criteria, expected structural shape, and validation checks the terminal validator will apply. The block is machine-readable, appended to every stage response, and subject to the same confidentiality contract that protects other archetype internals.
2. An implementation in the `rlabs-agentguard` Python package (version 0.13.0, publicly available on PyPI) that adds the block to the `skeleton`, `contracts_and_wiring`, and `logic` pipeline endpoints without changing any existing API shape, and a regression test suite that pins the block's presence, shape, and cross-stage consistency.
3. A measurement of the overhead across the ten built-in archetypes, reporting criteria count, contract payload size, and response-size share, with full methodology and raw data reproducible from the released package.
4. An argument, grounded in one observational dogfood case and in the broader literature on LLM attention and prompt design, that the principle generalises to other pipeline patterns — retrieval-augmented generation, agentic loops with tool calls, multi-pass content generation — wherever the generator is operating against a declared rubric.

We are explicit about what this paper does not claim. We do not present a controlled experiment with multiple replications across models showing a statistically significant reduction in validation failures; we flag that work as future. We present a mechanism, its implementation, its cost, and the motivating observation.

2. Related Work

We organise the relevant literature around three topic areas: type systems as a conceptual precedent, alignment techniques that declare specifications to the model, and prompt-engineering work on embedded rubrics.

2.1 Type systems and declared contracts as prevention

The argument that declaring a contract at generation time is cheaper than discovering a violation after the fact is not original to LLM systems. It is foundational to static typing. Modern type systems — Hindley-Milner in the ML tradition (Milner, 1978), gradual typing in TypeScript (Microsoft, 2024), strict mode in Python (Rossum et al., 2014) — make the contract (function signatures, invariants) visible at every call site, and verify the contract both locally (within a function) and globally (at build time). The empirical finding from this literature, relevant to our argument, is that earlier surfacing of a violation correlates with smaller fix cost and improved maintainability over the life of a codebase (Hanenberg et al., 2014). Our proposal can be read as applying this well-known pattern to LLM-generated artefacts: the `self_challenge_criteria` are the types, the pipeline stages are call sites, and the terminal validator is the global type check. Prior to this change, we had only the global check.

Contract programming in the Eiffel tradition (Meyer, 1992) and its descendants in modern languages (Racket (Findler and Felleisen, 2002), Dafny (Leino, 2010)) make the same argument in a runtime setting: assertions declared close to where behaviour is defined catch more bugs per token of specification than assertions placed only at system boundaries.

2.2 Alignment and explicit specifications for LLM behaviour

A second relevant thread is the use of declared specifications to shape model behaviour during generation rather than after. Constitutional AI (Bai et al., 2022) provides the model with a written constitution during training, and during inference the model checks its own outputs against those principles before emitting them. Deliberative alignment (Guan et al., 2024) extends this to having the model reason explicitly about a specification before acting on a request. Both approaches share our core assumption: behaviour is more tractable when the model has the evaluative frame in context while producing output, rather than being evaluated against a frame it never saw.

Our contribution differs in scope. We are not training on the rubric; we are surfacing an existing, caller-provided rubric at inference time. The rubric is not moral or safety-oriented; it is task-structural (file layout, style rules, domain-specific criteria). The implementation is a single field in a tool response, not a training regime.

2.3 Rubric-in-prompt and self-critique

The third thread is the recent prompt-engineering literature on embedding evaluation criteria directly in the generation prompt. Self-Refine (Madaan et al., 2023) shows that an LLM asked to critique and revise its own output improves quality on reasoning benchmarks, with the largest gains when the critique prompt is specific rather than generic. Reflexion (Shinn et al., 2023) extends this with episodic memory of past failures against a rubric. Rubric-augmented evaluation work in assistant benchmarking (Zheng et al., 2023) shows that the quality of LLM-as-judge scoring rises when the judge model receives explicit scoring criteria rather than generic quality prompts.

The closest precedent is LATS (Zhou et al., 2024), which introduces a search tree where each node is evaluated against criteria shared across siblings. In LATS the criteria live in the search loop; in our design they live in the pipeline stage payload. The key distinction is that LATS assumes a single end-to-end generation problem, while we address the case where the artefact is assembled from multiple pipeline stages that may run minutes or days apart and may be executed by different model instances.

A practical observation: none of the rubric-in-prompt work addresses the confidentiality concern that arises when the rubric itself is intellectual property of a third party — for instance, an archetype author in a marketplace. Our design includes a confidentiality directive that instructs the generating model to use the criteria but not expose them back to the end user, which makes the upfront-rubric pattern compatible with paid or licensed rubrics. We are not aware of prior published work on this combination.

3. Methodology

This section describes the design of the `quality_contract` block, its integration into AgentGuard's three pipeline stages, and the measurement experiment we ran over the ten built-in archetypes.

3.1 Pipeline under study

AgentGuard (rlabs, 2026c) exposes a code-and-artefact generation pipeline via the Model Context Protocol (MCP) (Anthropic, 2024c). The pipeline has three user-facing generation stages (plus an orthogonal validator):

- `skeleton`: given a specification and an archetype name, returns instructions for the caller LLM to produce a file tree with responsibilities per file.

- `contracts_and_wiring`: given the skeleton, returns per-file instructions for typed stubs and import wiring.
- `logic`: given a file body containing `NotImplementedError` stubs, returns instructions for implementing a single function.

A fourth endpoint, `validate`, runs after all generation is done. It returns the declared evaluation criteria (the archetype's `self_challenge_criteria`), expected directory/file structure, and validation checks, and instructs the caller LLM to self-score the finished artefact. Prior to version 0.13.0, the criteria were reachable only via this terminal call.

Each archetype is a declarative YAML specification (rlabs, 2026a) carrying a `tech_stack`, a `structure` block, a `self_challenge` block containing a list of prose criteria, and a `validation` block with a list of check types. All four pipeline stages load the archetype by name; the terminal `validate` call embeds the criteria and structure into its response. Our change surfaces the same payload at the three earlier stages.

3.2 The `quality_contract` block

We define the `quality_contract` block as a compact JSON object with the following schema:

```
{
  "stage": "L1_skeleton",
  "must_satisfy_criteria": ["<prose criterion 1>", "..."],
  "expected_structure": {
    "dirs":  ["<glob>", "..."],
    "files": ["<path>", "..."]
  },
  "validation_checks": ["<check-name>", "..."],
  "enforcement": "<prose note instructing the generator>"
}
```

The `stage` field identifies which pipeline step emitted the block, so the generating model can vary its emphasis by stage (structural fidelity during `skeleton`, contract fidelity during `contracts_and_wiring`, logic-level fidelity during `logic`). The criteria list is emitted verbatim from the archetype, with no filtering or summarisation. The `expected_structure` dictionary carries the dirs and files the terminal validator will check. The `validation_checks` list enumerates the named checks (e.g. `structure`, `syntax`) the terminal step will apply. The `enforcement` field is a one-paragraph prose note directing the generator to treat the contract as binding during generation — not as advisory.

The block is covered by the same confidentiality directive that already protects archetype internals in the pipeline responses. The generating model is instructed to use the contract contents to guide output but not to reproduce or paraphrase them in artefacts the end user sees. This preserves the intellectual property boundary around paid or licensed archetypes.

3.3 Implementation

Implementation is a single helper function and three call sites. The helper, `_build_quality_contract(arch,stage)`, reads the archetype's `self_challenge.criteria`, `structure`, and `validation.checks` attributes and returns the dictionary above. The three call sites are the return payloads of `agentguard_skeleton`, `agentguard_contracts_and_wiring`, and `agentguard_logic`. In each case the new block is added to the existing JSON response under the key `quality_contract`. No existing fields change. Callers that ignore the new key observe no behavioural difference.

The full diff is available in commit `c405ad3` of the public mirror (rlabs, 2026b). The test suite (`tests/test_quality_contract.py`) pins four invariants: the block is present in all three stages; it contains the five required top-level keys; the criteria list is non-empty for archetypes that ship with criteria; and the criteria are consistent across the three stages for a given archetype.

3.4 Measurement protocol

To measure the overhead of embedding the contract we wrote a deterministic script (`scripts/measure_quality_contract.py`, shipped with the package) that iterates over every built-in archetype in the local registry, invokes `agentguard_skeleton` with a fixed synthetic specification, and records three values per archetype:

- `criteria_count`: number of criteria in the archetype's `self_challenge.criteria` list, as surfaced in the contract.
- `contract_bytes`: UTF-8 byte length of the JSON-serialised `quality_contract` dictionary.
- `response_bytes_total`: UTF-8 byte length of the entire skeleton response including the contract.

From these we derive the overhead percentage as

$$100 \cdot \frac{\texttt{contract\_bytes}}{\texttt{response\_bytes\_total} - \texttt{contract\_bytes}},$$

i.e. the contract size relative to what the response would have been without it. We approximate the token cost as `contract_bytes`/4, consistent with the common four-bytes-per-token heuristic for English text (OpenAI, 2023). We do not make the specification vary across archetypes because the contract is a function of the archetype, not of the specification.

The script is fully reproducible. It performs no network calls, uses no randomness, and depends only on locally installed archetypes. A reader with `rlabs-agentguard` 0.13.0 installed can run `python-mscripts.measure_quality_contract` and obtain the same numbers we report in Section 4, plus any additional marketplace archetypes their API key grants access to.

3.5 Observational corpus

Beyond the measurement of overhead, we report one observational case: the quarterly strategy memo generated via the `strategy_memo` archetype immediately before the v0.13.0 change (rlabs, 2026d). The memo was ten markdown files, approximately seven hundred lines of prose, generated in a single session by one agent instance. Terminal validation surfaced three rubric violations — two involving formatting rules in sections written after the sixty-percent mark of the artefact, one involving an author-label convention applied throughout. The agent's rewrite cost, measured by `git diff` lines changed in the correction commit, was approximately twenty percent of the original artefact size. We use this case to motivate the change and to frame the future-work question; we do not present it as a controlled experimental result.

4. Results

This section reports the output of the measurement protocol described in Section 3 without interpretation. Interpretation is deferred to Section 5.

4.1 Per-archetype overhead

Table 1 shows the measurement output for each of the ten built-in archetypes available in `rlabs-agentguard` 0.13.0.

4.2 Aggregate statistics

Over the ten archetypes:

- Mean criteria count per archetype: 13.3 (stdev 9.0, range 5–30).
- Mean contract bytes: 1,498 (stdev 749, range 752–3,030).
- Mean total skeleton response bytes: 5,092 (stdev 1,026, range 3,996–6,917).
- Mean overhead percentage: 40.6 (stdev 16.6, range 23.2–78.0).
- Approximate mean token cost at four-bytes-per-token: 374 tokens (range 188–759).

Table 1: Quality-contract overhead per archetype, measured on the skeleton response with a fixed synthetic specification. `criteria` is the count of items in the archetype's `self_challenge.criteria` list. `contract_B` is the UTF-8 byte length of the serialised contract block. `total_B` is the byte length of the full skeleton response. `overhead_%` is the contract share relative to the rest of the response.

archetype	criteria	contract_B	total_B	overhead_%
api_backend	15	1387	4982	38.6
cli_tool	7	1015	4510	29.0
debug_backend	6	924	4161	28.5
debug_frontend	7	1125	4386	34.5
library	9	1030	4487	29.8
react_spa	30	2435	6884	54.7
script	5	752	3996	23.2
software_architecture	26	3030	6917	78.0
strategy_memo	8	1488	5066	41.6
web_app	20	1794	5532	48.0

4.3 Correlation between criteria count and overhead

Within the ten archetypes, contract size is dominated by the criteria list length. A rank-order inspection of Table 1 shows the two archetypes with the largest criteria counts (`react_spa` at 30 and `software_architecture` at 26) also have the two largest contract byte sizes (2,435 and 3,030 respectively) and the two largest overhead percentages (54.7 and 78.0 respectively). The archetype with the smallest criteria count (`script` at 5) has the smallest contract and the smallest overhead. We do not fit a regression given $n = 10$.

4.4 Observational case reference

The dogfood case described in Section 3 is referenced as a single data point. Of the three violations raised at terminal validation on the seven-hundred-line strategy memo:

- Two concerned a formatting rule (no bullet points in body prose) that was violated in sections written at roughly the sixty-percent and eighty-percent marks of the artefact. Sections written earlier (before the sixty-percent mark) did not violate the rule.
- One concerned a field-label convention (owner should be a role, not a personal name) applied throughout the artefact.

The rewrite commit modifying the violating content was approximately 140 lines changed against an original artefact size of approximately 700 lines, i.e. a twenty-percent surface area.

We report these counts as observations from a single generation session. We do not quantify what the same session would have produced under the v0.13.0 contract because that would require a counter-factual replay we did not perform.

5. Discussion

We interpret the measurements from Section 4 and the observational case from Section 3 under the framing of Section 2, then discuss limits and generalisation.

5.1 Interpretation of the overhead

The mean per-stage overhead of approximately 374 tokens (mean 1,498 bytes, stdev 749) is small in absolute terms. For context, GPT-4 class models at the time of writing have context windows measured in tens of thousands of tokens (OpenAI, 2024a); 374 additional tokens per stage response is a fraction of a percent of the available context. Against that budget, the open question is whether the token expenditure prevents enough correction work to justify itself.

The observational dogfood case is consistent with, but does not prove, the hypothesis that it does. The strategy memo rewrite touched roughly 140 lines of markdown to correct three

violations. If the generating model had encountered the criteria during the writing of the violating sections — sections we know from the spatial distribution of violations were the later ones, where rubric drift is most likely — the model had a plausible opportunity to avoid the violations at the cost of the contract token overhead at each earlier stage response it consumed. We cannot quantify that counter-factual without a controlled re-run.

5.2 Mechanism, not empirical claim

The contribution we feel confident defending is the mechanism: a compact, stage-independent payload that carries the terminal validator's rubric to every earlier generation stage, is implemented without breaking existing callers, and is cheap to produce. The measurements establish that the mechanism is lightweight. What they do not establish is that the mechanism reduces final-validation failures at any particular rate.

We flag this explicitly because the difference matters for readers evaluating whether to adopt a similar pattern in their own pipelines. The argument for adoption that this paper supports is: the mechanism is cheap, compatible, and theoretically motivated by analogous findings in static typing and alignment work. The argument for adoption that this paper does not support is: the mechanism reduces failure rate by $X\%$ in a statistically significant way. The latter requires the controlled experiment we flag in Section 5.5.

5.3 Generalisation beyond AgentGuard

The design assumes three conditions hold in the target pipeline:

1. There is a declared, machine-readable rubric at evaluation time.
2. The generation is assembled in multiple stages whose outputs are visible to the same generating model.
3. The rubric can be serialised compactly enough to sit in every stage response without materially crowding out the rest of the context.

These conditions are common. Retrieval-augmented generation (RAG) pipelines typically have declared quality criteria for citation accuracy and factual grounding (Gao et al., 2024); the criteria are checked at terminal evaluation, not embedded during retrieval or composition. Agentic tool-calling loops have declared criteria for tool-call validity and termination conditions that are usually encoded only in the final scoring prompt, not in the per-step prompts that the agent sees while acting. Multi-pass content generation (outline → draft → revision) has editorial criteria that typically appear only at the revision prompt, not at the draft prompt.

In each of these settings the same transformation applies: lift the rubric into every stage response. The implementation cost is similar to what we describe in Section 3 — a helper function plus call-site inclusion. We have not implemented the transformation in those domains and we make no claim about effect sizes there. We argue only that the design criterion — declared rubric, multi-stage pipeline, bounded serialisation cost — is not unique to AgentGuard.

5.4 Threats to Validity

Internal validity. The measurement protocol measures overhead of the mechanism, not reduction in failures. The observational case (Section 3) is a single generation session by a single model instance against a single archetype. We do not claim causation between contract presence and failure rate; the dogfood motivated the design but does not empirically test it.

External validity. The ten archetypes we measured are all built-in archetypes authored by the project maintainers. Marketplace archetypes may have systematically different criteria counts or structural complexity; we did not measure them because the measurement script requires local access to archetype YAML and marketplace archetypes download lazily. A reader with broader archetype access can rerun the script and extend Table 1.

Construct validity. "Drift" is operationalised as "violations present at terminal validation," which is not equivalent to "violations the generator would have avoided with the rubric in context." A violation may exist because the rubric is unclear, because the model cannot satisfy it in principle, or because the generator made a local mistake unrelated to attention. Our design addresses only the third case. Empirically separating the three requires per-violation analysis we did not perform.

Confidentiality constraint. The contract payload is subject to a confidentiality directive instructing the generating model not to expose criteria verbatim to the end user. We have not stress-tested the model's compliance with this directive at scale. A safety-critical deployment should include an independent check that the criteria do not leak into user-visible output.

Token-cost estimate. The four-bytes-per-token approximation is accurate for English prose with modern BPE tokenisers but can diverge by ten to twenty percent for criteria written in other languages or heavy with code identifiers (Karpathy, 2024). The per-archetype numbers in Table 1 should be re-measured with an exact tokeniser for precision pricing.

5.5 Future Work

We identify four specific follow-ups:

1. **Controlled multi-model study.** Generate the same corpus — ideally fifty or more artefacts spanning both code and content archetypes — with three pipeline variants: (a) no contract at any stage, (b) contract at stages but no terminal validate, (c) contract plus terminal validate (the current v0.13.0 behaviour). Measure violations found at terminal validation and rewrite cost. Replicate across at least three distinct generator models.
2. **Cost-benefit modelling.** Express the trade-off as an expected-value calculation: the probability of a violation in the naive pipeline times the cost of a rewrite, versus the deterministic cost of the contract tokens across all stages. Fit to empirical data from (1) to establish when the contract is net-positive.
3. **Hybrid `output_kind` archetypes.** AgentGuard's schema supports a `hybrid` output kind that mixes code files and content files in the same artefact. The contract design assumes one rubric applies throughout; we have not verified whether hybrid artefacts benefit from segment-specific contracts. A small extension to the helper could emit different contract fields per stage or per file tier.
4. **Community measurement.** Invite archetype authors in the public marketplace to run the measurement script against their own archetypes and contribute the results to an open table. This would broaden the sample substantially without requiring centralised access to proprietary rubrics.

6. Conclusion

We described a small, backwards-compatible change to the AgentGuard generation pipeline that surfaces the evaluation rubric at every generation stage instead of only at the terminal validator. The change was motivated by an observed dogfood case in which a long multi-section artefact drifted from its declared criteria in later sections. We formalised the `quality_contract` block, implemented it across three pipeline stages in version 0.13.0 of the public Python package, and measured its overhead across the ten built-in archetypes: a mean of approximately 374 tokens per stage response, with a maximum of 759 tokens for the most criteria-heavy archetype.

We stop short of claiming the change reduces terminal-validation failures by any specific rate. We present the mechanism, its cost, and the rationale; we flag controlled experimental evaluation as the natural next step. The principle — that declared rubrics are more useful to a generating model when they arrive with the task than when they arrive as a verdict — is a straightforward application of prevention-over-detection from static typing and contract programming to the LLM

pipeline setting, adapted to account for the confidentiality constraints that arise when rubrics are intellectual property.

The change is available today as `rlabs-agentguard` 0.13.0 on PyPI. The measurement script and regression tests are part of the same release, so a reader who wants to extend or replicate the analysis can do so without access to any proprietary data. We invite empirical follow-ups.

Related work by the author. Labbé (2026c) measures the mechanism reported here across four LLMs — three frontier and one local — on three archetypes with a reproducible protocol. Labbé (2026b) documents the four-stage pipeline architecture that implements the mechanism. Labbé (2026a) argues that the declarative archetype is the minimum-viable governance primitive for AI-assisted development at enterprise scale — the sharpest form of the thesis this work builds toward: *function is solved; form is the bottleneck.*

References

Anthropic (2024a). *Building Agents with the Claude Agent SDK*. https://www.anthropic.com/engineering/building-effective-agents. Engineering blog post.

— (2024b). *Claude Code — Anthropic's Agentic Coding Tool*. https://www.claude.com/claude-code. Anthropic product documentation, retrieved 2026.

— (2024c). *Model Context Protocol specification*. https://modelcontextprotocol.io/specification. Open protocol for tool and resource exposure to LLM clients.

Bai, Yuntao, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, et al. (2022). "Constitutional AI: Harmlessness from AI Feedback". In: *arXiv preprint arXiv:2212.08073*. DOI: 10.48550/arXiv.2212.08073.

Cursor (2024). *Cursor — The AI Code Editor*. https://www.cursor.com. Commercial LLM-integrated IDE, online documentation.

Findler, Robert Bruce and Matthias Felleisen (2002). "Contracts for higher-order functions". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, pp. 48–59. DOI: 10.1145/581478.581484.

Gao, Yunfan, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang (2024). "Retrieval-Augmented Generation for Large Language Models: A Survey". In: *arXiv preprint arXiv:2312.10997*. DOI: 10.48550/arXiv.2312.10997.

Guan, Melody Y., Manas Joglekar, Eric Wallace, Saachi Jain, Boaz Barak, Alec Heylar, Rachel Dias, Andrea Vallone, Hongyu Ren, Jason Wei, Hyung Won Chung, Sam Toyer, Johannes Heidecke, Alex Beutel, and Amelia Glaese (2024). "Deliberative Alignment: Reasoning Enables Safer Language Models". In: *arXiv preprint arXiv:2412.16339*. DOI: 10.48550/arXiv.2412.16339.

Hanenberg, Stefan, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik (2014). "An empirical study on the impact of static typing on software maintainability". In: *Empirical Software Engineering* 19.5, pp. 1335–1382. DOI: 10.1007/s10664-013-9289-1.

Karpathy, Andrej (2024). *Let's build the GPT Tokenizer*. https://www.youtube.com/watch?v=zduSFxRajkE. Technical video essay on tokeniser behaviour across languages and code.

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

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

Leino, K. Rustan M. (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

Liu, Nelson F., Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang (2024). "Lost in the Middle: How Language Models Use Long Contexts". In: *Transactions of the Association for Computational Linguistics* 12, pp. 157–173. DOI: 10.1162/tacl_a_00638.

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* 36. DOI: 10.48550/arXiv.2303.17651.

Meyer, Bertrand (1992). "Applying "Design by Contract"". In: *IEEE Computer* 25.10, pp. 40–51. DOI: 10.1109/2.161279.

Microsoft (2024). *TypeScript Language Specification*. https://www.typescriptlang.org/docs/handbook/2/basic-types.html. Official language handbook.

Milner, Robin (1978). "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

OpenAI (2023). *What are tokens and how to count them?* https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them. OpenAI Help Center; source of the 4 bytes/token English rule of thumb.

— (2024a). *Models and their context windows*. https://platform.openai.com/docs/models. Official model reference, retrieved 2026.

— (2024b). *Structured Outputs in the OpenAI API*. https://openai.com/index/introducing-structured-outputs-in-the-api/. OpenAI Developer Blog.

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). *agentguard-lib public GitHub mirror of the Python package*. https://github.com/rlabs-cl/agentguard-lib. Source repository; commit c405ad3 introduces the quality_contract block.

— (2026c). *AgentGuard: A Quality-Assurance Engine for LLM-Generated Artefacts*. https://agentguard.rlabs.cl. Software package, version 0.13.0.

— (2026d). *Portfolio Strategy Memo: Q2 2026*. Internal document, infra/strategy/portfolio-2026-04. Observational case referenced for the dogfood example; generated via the strategy_memo archetype prior to the v0.13.0 change.

Rossum, Guido van, Jukka Lehtosalo, and Lukasz Langa (2014). *PEP 484 – Type Hints*. https://peps.python.org/pep-0484/. Python Enhancement Proposal, accepted.

Shinn, Noah, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao (2023). "Reflexion: Language Agents with Verbal Reinforcement Learning". In: *Advances in Neural Information Processing Systems* 36. DOI: 10.48550/arXiv.2303.11366.

Zheng, Lianmin, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica (2023). "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena". In: *Advances in Neural Information Processing Systems* 36. DOI: 10.48550/arXiv.2306.05685.

Zhou, Andy, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang (2024). *Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models*. Proceedings of the International Conference on Machine Learning (ICML 2024). DOI: 10.48550/arXiv.2310.04406.