Closed Languages for Agent Authorship: Design Principles and Production Evidence from Topaz

Author copy. Submitted to LMPL 2026, October 2026.

Matt Park
Studio Haze Inc.
cto@studiohaze.co.kr

Abstract

Agent authorship makes a programming language more than an expressiveness device: it becomes the boundary between intended production actions and tempting off-surface text. This extended abstract presents Topaz, a small closed intent language for application intent, as a design case for making that boundary executable for both humans and agents. The mechanism is deliberately narrow. Topaz gives both authors the same fixed grammar and refusal surface, keeps the public normative surface small enough to fit in a single model context, treats documentation examples as parser-checked, runnable, or transcript-pinned artifacts, and compares observed outcomes across interpreter and emission paths over pinned fixture sets. The evidence is release-harness rows reproduced at a pinned snapshot commit: checked-documentation counts, backend fixture counts, and an ATLAS three-way outcome-agreement count. The claim is bounded: Topaz does not show that an agent chose the right intent or that emitted programs are correct, only how a closed language boundary makes agent output executable, rejectable, and differentially checked.

CCS Concepts: • **Software and its engineering** → Domain specific languages; Software testing and debugging.

Keywords: closed intent language, agent authorship boundary, executable documentation, outcome-level differential agreement

1 Problem: Agent Authorship Needs a Closed Surface

For agent authorship, a language is not only an expressiveness device but a control surface. When a human asks an agent to change production intent, the immediate artifact is program text, and the review problem is not whether that text looks plausible but whether it stayed on a surface whose constructs, refusals, documentation, and release checks are explicit enough for the toolchain to accept or reject.

Large language models make program text a plausible medium for indirect authoring, and code-oriented models made that setting especially visible [2, 3]. Grammar-constrained decoding restricts what a model can emit at generation time [6, 7]; Topaz places that restriction at the toolchain boundary instead, so any author on any path meets the same closed surface. Traditional language-design work gives a complementary lesson: domain-specific languages and modular interfaces are useful not only because they shorten programs, but because they decide what distinctions can be expressed and where responsibility is localized [4, 5, 9]. The agent-authorship setting makes this old lesson operational: an author who can emit any host-language idiom makes every review a judgment about an open language, while an author held to a closed intent language turns many tempting inventions into parser or checker events.

Topaz is the public system used here as the design case [8]. It is a small language for application intent, written by people and agents on the same closed surface. This abstract introduces no new benchmark or experiment; it organizes the Topaz design around a narrower claim: the language boundary itself can be part of the harness. A closed surface does not make agents correct. It makes off-surface output rejectable, keeps examples and normative material close to the toolchain, and lets multiple execution and emission paths be compared by observed outcomes over pinned fixture sets.

2 Design Principles

Topaz reads as five design principles for agent authorship, ordinary PL mechanisms whose purpose is authoring control rather than expressiveness alone.

Closed grammar and refusal surface. Topaz constrains agent authorship by giving humans and agents the same closed intent surface: fixed grammar, reserved refusal points, and no per-file macro or reflection escape hatch. This does not prove agent intent correctness or eliminate hallucination. It narrows the authorable surface and makes off-surface output rejectable. The refusal surface is not merely a missing-feature list. It records the constructs that the language deliberately leaves off the authorable path: open template systems, local grammar extension, unchecked host-language imports, string indexing and slicing, call-site generics, spread-style structural copying, assertions as an authoring idiom, and anonymous function forms outside the approved profile. A closed language turns off-surface agent output into a parser or checker event rather than a reviewer judgment. Module paths reject NFC/NFD and case-fold collisions at the boundary.

Normative spec fits in context. The public normative surface is small enough to fit in a single model context: the SPEC is about 18,530 tokens and the public profiles material about 3,214 tokens, with 22 reserved words. This does not mean the model understands or follows the spec, only that the normative surface can be supplied whole rather than retrieved piecemeal.

Executable documentation. Public examples are part of the authoring surface. Topaz treats documentation as executable documentation by classifying samples as parser-checked samples, runnable examples, or transcript-pinned outputs. This makes documentation drift visible to the release harness and gives agent prompts a checked source of idioms rather than a prose-only source of conventions.

Outcome-level differential agreement. Topaz separates claims about intent from claims about observed outcomes. Its harness compares interpreter results and emitted-path results over pinned fixture sets. The comparison is outcome-level: it asks whether the relevant paths agree on what is observed for the selected fixtures, not whether all possible programs are covered.

Closed tagged templates as intent forms. Topaz also moves common string-construction surfaces into closed tagged templates such as `sql`, `sh`, and `path`; for example, the SQL form lowers to parameterized query objects rather than raw string concatenation.

Together, these principles make the language boundary a release artifact. The grammar decides what can be authored, the profile decides which idioms are visible to agents, the documentation gives checked examples of those idioms, and the release harness checks whether execution and emission paths agree on observed behavior. The important move is not to ask the agent for discipline in an open language, but to make the authorable surface small, closed, and checked enough that off-surface text is not a matter of taste.

3 Production and Release Evidence

The evidence is structured as a snapshot table. Numeric claims are reproduced from producer outputs at snapshot commit `6d0be1e9d7cc`, with the working tree clean. The snapshot repository is private, so measured rows are author-reproduced producer outputs, not reviewer-runnable artifacts; documented design facts remain checkable on the public pages. Public Topaz and ATLAS pages are used as third-person public identity and design citations only [1, 8]; measured counts come from those producer outputs.

Topaz treats documentation as a checked artifact surface: in the audited snapshot the release harness checks 594 documentation samples, 530 parse samples, 40 executable examples, and 24 transcript-pinned output blocks. These checks do not prove the prose complete or every example faithful to intended semantics; they show public examples staying syntactically and operationally synchronized with the toolchain.

Topaz's run≡build discipline compares observed outcomes across execution and emission paths: the Rust path covers 1,386 pinned single-file fixtures and 34 module fixtures, the native backend reports 220 eligible cases with 0 refusals and 1 fallback (a refusal here is a declined lowering, not the language's refusal surface), and the ATLAS proof suite records 5,939 three-way agreements among interpreter, emitted Rust, and generated Python outcomes. This is outcome-level agreement over pinned suites, not byte-for-byte binary identity, formal compiler verification, or whole-program correctness.

Topaz has been exercised under production pressure, most concretely through the ATLAS proof suite's 5,939 three-way outcome agreements over a paragraph line-breaking kernel authored in Topaz, pure fixed-point integer code replayed across the interpreter and both emission paths. We use this as evidence that the workflow is not only a toy harness, while avoiding claims about global typography or product correctness.

Table 1 separates design facts from measured release evidence. The distinction is part of the claim. A documented refusal list can support the statement that Topaz has a closed authoring surface; it cannot support a frequency claim about agent failures. A producer-backed fixture row can support an observed outcome-agreement claim; it cannot support a claim about all programs. ATLAS contributes production pressure because it exercises the workflow through a public production case and a proof suite, not because the paper exposes private product data.

4 Lessons, Limits, and SLE Roadmap

The main lesson is that agent authorship changes what a language boundary is for. A conventional DSL boundary improves notation, hides host detail, or localizes domain concepts; the Topaz boundary also decides what an agent may author at all, which makes the parser, checker, documentation verifier, interpreter, emitters, and release gates parts of the authoring interface rather than downstream quality tools.

A second lesson is that executable documentation is a prompt-surface discipline. Agents learn a project's local idiom largely from examples, and examples checked only by human review let the prompt surface drift from the toolchain. Parser-checked, runnable, or transcript-pinned examples make the public documentation a release-gated artifact, narrowing the gap between what agents see and what the toolchain accepts.

A third lesson is that differential evidence should be phrased at the level actually measured, observed outcomes on pinned fixture sets and backend paths, strong enough for a release discipline yet narrow enough not to overstate. The table therefore treats measured counts as producer-backed rows, documented design facts as public facts, and sanitized failure modes as motivation only.

Table 1. Evidence map for the final snapshot. Measured counts are producer outputs reproduced at snapshot commit `6d0be1e9d7cc` (working tree clean, `cargo 1.96.0`, `gpt-tokenizer@3.4.0`, `o200k_base`); documented design facts are pinned to the public docs at the same commit.

Claim	Tag	Producer/source	Citation path	Counts	Allowed wording	Excludes
A: closed surface as authorship control	DOCUMENTED; MEASURED for surface size	Public SPEC, public profiles, refusal list, parser/checker behavior	Public fact for documented surface; sanitized author assertion only for motivating failure modes	22; optionally 11,402, 18,530, 1,912, 1,765, 3,214	Closed intent language; agent authorship boundary; off-surface output becomes a parser or checker event	No agent-intent correctness claim; no elimination claim; no private frequency claim
B: executable documentation	MEASURED	`npm run verify-samples`	Producer output; public examples as public facts	594 total, decomposed as 530, 40, 24	Documentation as checked artifact surface; parser-checked samples; runnable examples; transcript-pinned outputs	Does not prove prose completeness or full intended semantics
C: outcome-level differential agreement	MEASURED	Rust run≡build fixture producer; native backend producer; ATLAS proof-suite producer	Producer output for counts; ATLAS public site for public case identity	1,386, 34, 220, 0, 1, 5,939	Outcome-level differential agreement; observed outcomes; three-way backend agreement over pinned fixture sets	No claim beyond pinned suites; no whole-program correctness claim
D: ATLAS production evidence	MEASURED plus DOCUMENTED identity	ATLAS public site; ATLAS proof-suite producer	Public fact for ATLAS identity; producer output for proof-suite count	5,939	Production pressure on the workflow; not only a toy harness	No private operational data; no global typography or product-correctness claim
E: closed tagged templates as intent forms	DOCUMENTED	Public docs and SPEC template registry	Public fact	None (design fact)	Closed tagged templates; intent forms	No general guarantee; no universal guarantee; no prevention claims
F: spec-in-context	MEASURED	SPEC and PROFILES word, token, and line counters; reserved-word producer	Producer output; public documents as public facts	11,402, 18,530, 1,912, 1,765, 3,214, 22	Normative spec fits in a single model context; supplied whole rather than piecemeal	Does not mean the model understands or follows the spec

Topaz does not prove that an agent chose the right intent, that emitted code is correct for all programs or formally verified, that hallucination is eliminated, that tagged templates provide any general security or injection guarantee, or that ATLAS output is always typographically correct. This paper is not an LLM benchmark study, reports no agent-authorship percentage, and uses no product, client, or private operational data. Its claim is narrower: a closed authoring surface and release harness make agent output executable, rejectable, and differentially checked at the toolchain boundary.

Future work will turn this design case into a comparative PL study over an agent-authored change corpus, measuring parse-error rate, off-surface invention frequency, convergence iterations, and repair burden across Topaz and mainstream languages. That study belongs outside this abstract; what Topaz already suggests is that agent-authored software

becomes less open-ended not by asking agents for discipline, but by giving them a smaller, executable language boundary whose whole toolchain agrees on what counts.

References

[1] ATLAS. 2026. ATLAS Cloud Typesetting Engine. https://www.atlasbind.com/. Public production site.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[3] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] doi:10.48550/arXiv.2107.03374

[4] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *Comput. Surveys* 28, 4es (1996), 196–es. doi:10.1145/242224.242477

[5] David L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (1972), 1053–1058. doi:10.1145/361598.361623

[6] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations (ICLR)*.

[7] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 9895–9901.

[8] Topaz. 2026. Topaz: One way to say it. https://topaz.ooo/. Public system site.

[9] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35, 6 (2000), 26–36. doi:10.1145/352029.352035