Behavioral Code: Legible, Auditable Loops for Autonomous Agents

Abutalib Namazov [1] **Tae Wook Kim** [1] **Eagon Meng** [1] **Daniel Jackson** [1]

Abstract

Autonomous research agents can now run unattended for hours, producing code changes, logs, and metrics, yet leave no readable account of what situations they recognized, why they acted, or what they expected to happen next. In many current systems, the closest thing to a behavioral specification is a prompt plus a tool list: reviewers can inspect the scaffold and the execution trace, but not a structured account of the agent's intended behavior.

We introduce *behavioral code*: a reviewable program of named *concepts* and *reactions* that makes an agent loop's behavior explicit. Concepts describe human-meaningful situations in the loop, while reactions specify how the agent should respond. Each reaction cites the events that triggered it and records predictions before outcomes are observed. This lets later successes and failures be compared against explicit behavioral commitments, rather than explained after the fact. We pair this representation with a lightweight harness that enforces declared commitments against the execution trace. We demonstrate the system in two case studies: a language-model training loop and an exploratory C++ performance loop. Across both settings, the concepts make the loop legible; the harness makes its execution auditable.

1. Introduction

A researcher leaves a coding agent running and comes back to a long sequence of commits and an updated metrics file. The metric improved, but the record does not say whether any individual change worked for an anticipated reason, whether two changes repeated the same idea, or whether the apparent gain sat inside run-to-run noise. Across ML training (Karpathy, 2025; Lu et al., 2024), software repair (Jimenez et al., 2024; Yang et al., 2024), and multi-agent orchestration (Hong et al., 2024), traces of *what happened* are voluminous, while the artifact governing the run is usually a prompt plus a tool list (Yao et al., 2023; Wang et al., 2024a).

That artifact has two structural limitations: one about control and one about explanation. A reviewer who wants to forbid an action—editing production configuration, committing without a measurement, or acting without a stated hypothesis—has nowhere to write a scoped rule that the runtime checks at the moment of action. The prompt may express such preferences, but the behavior it induces is mediated by model weights no reviewer can inspect; Chan et al. (2024) treat this opacity as a central governance problem for deployed agents. External guardrails can block some actions, but they usually live beside the agent loop rather than serving as the behavioral program whose executions appear in the trace (Rebedea et al., 2023; Debenedetti et al., 2025). Auditability also suffers: rules written in a prompt establishing allowed behaviors are distinct from the guardrails enforced at runtime.

The trace is detailed but semantically thin. Metric files and tool-call traces capture results and intermediate interaction, but not the prediction that preceded a run, the rule that licensed it, or the earlier event that made it relevant. Without predictions, a reviewer cannot distinguish a correct intervention from a stumbled-into improvement. Without provenance links, the reviewer cannot reconstruct why a given action was taken now rather than later, or in response to which earlier event. The trace is therefore hard to read as a research process: it says what happened, but not what hypothesis was being tested or why one action followed from another.

Our goal is automated research whose governing program, predictions, and event trace can all be inspected after the fact. Our mechanism is *behavioral code*: a small, reviewable program that makes the load-bearing behavior of an agent loop explicit. The program declares *concepts*, which

[1]MIT Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology. Correspondence to: Abutalib Namazov <abutalib@mit.edu>, Tae Wook Kim <terrytwk@mit.edu>, Eagon Meng <eagon@mit.edu>, Daniel Jackson <dnj@mit.edu>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

name the kinds of events the loop can produce, and *reactions*, which specify when one event authorizes another. Predictions, provenance links, and permissions are recorded as events rather than reconstructed from prose, so the trace records executions of the same behavioral program.

A harness enforces this contract at write time. Events are recorded only when authorized by the declared behavioral program, claimed metrics and diffs are grounded in execution, and actions outside the interface are rejected. The trace therefore records not just what happened, but which rule authorized it, what evidence it cited, and what prediction preceded the outcome.

Contributions. **(C1)** We introduce behavioral code: concepts and reactions over an append-only event log, with predictions and provenance recorded before outcomes, making an agent loop legible as a small human-readable program (§2.1). **(C2)** We describe a harness that enforces the contract at write time, making permissions, provenance, and outcomes auditable invariants rather than conventions (§2.2). **(C3)** We instantiate the same interface in two research loops—language-model training and exploratory C++ performance optimization—showing how changing only the behavioral program adapts the trace structure to distinct workflows (§3–4).

2. Behavioral code and its harness

The behavioral code program (§2.1) declares the vocabulary of the agent loop the reactions that relate its trace events. The harness (§2.2) executes the loop against this program: it accepts only events licensed by declared reactions, grounds empirical claims in execution, and records the provenance needed for audit.

2.1. Legibility: concepts and reactions

We build on the concepts-and-synchronizations pattern of Meng & Jackson (2025), applying legible software ideas to the structure of an autonomous research loop. In our setting, concepts name research activities, while synchronizations specify the rules by which one activity authorizes another. Each accepted reaction execution is then written into an auditable log with provenance, predictions, and grounded outcomes.

A *concept* names a human-meaningful research state or activity, such as a hypothesis, code modification, experiment, result, or message, together with the events that can occur within it. A *reaction* connects concepts by specifying when events in one activity authorize events in another.

This split matches the structure of autoresearch. The artifacts a reviewer wants to inspect are local: hypotheses, code changes, experiments, measurements, and log entries. The audit relationships span them: a code change should point to the hypothesis that motivated it, an experiment should identify the prediction it tests, and a logged result should compare the observed outcome against a prior prediction.

A reaction specifies a permitted transition in the research process: when a new event is proposed, the reaction identifies the earlier events that must already exist and the follow-on events it authorizes. We write these rules with a compact `when`/`where`/`then` syntax (see Appendix C).

The autoresearch loop uses six concepts: `Hypothesizing`, `Modifying`, `Experimenting`, `Evaluating`, `Logging`, and `Communicating`. A representative reaction is:

```
when:  Modifying.applied(?change)
where: Hypothesizing.formed(?hypothesis,
                            supports: ?change)
       Experimenting.planned(?experiment,
                              hypothesis: ?hypothesis)
then:  authorize Modifying.commit(?change)
       authorize Experimenting.run(?experiment)
```

Here, `?change` is bound by the applied-change event, and the `where` clauses require earlier log events that bind a supporting `?hypothesis` and planned `?experiment`. The rule says that an applied change may be committed and evaluated only when those premises already exist in the log.

Provenance links and predictions. When the harness accepts an event under a reaction execution, it records which earlier events licensed that execution. This provenance link is separate from the agent's free-form `rationale`: the former is checked structure, the latter is model-written explanation. Keeping them separate prevents a common failure of prompt-and-tool traces, in which a fluent reason is treated as if it were provenance.

Predictions are also structural rather than retrospective. Each `Hypothesizing.formed` event must include a prediction before the corresponding experiment runs—direction, magnitude, mechanism, and side effects. A later `Logging.recorded` event is accepted only if it compares the observed outcome against that prediction. Because the log is append-only, weakening a prediction after seeing the result is itself visible. A metric improvement with the wrong mechanism becomes an explicit disagreement between prediction and outcome, not a silent success.

2.2. Auditability: the harness

The harness turns the behavioral code program from a convention into an invariant. It is the runtime for the loop and the sole writer to the event log: it surfaces the reactions whose premises are currently satisfied, mediates the agent's proposed events against the program, and records both accepted events and refusals. The agent reads the trace and proposes actions, but whether a proposal becomes part of the

accepted trace is determined by the current log, the declared reactions, and the checks attached to them. Acceptance is a checked transition rather than a model choice.

What the harness does at each step. The harness inspects the log, identifies which reactions in the program are now eligible, and presents the agent with that set. When the agent proposes an event under an eligible reaction, the harness binds the event to earlier log entries via the reaction's premises, runs the checks attached to the event and reaction, and either appends the event with its provenance links or records a refusal with the reason. Refusals are first-class entries in the trace.

Extension points. The harness exposes three places where a user shapes the contract without modifying the runtime. *Validators* decide whether an event satisfies the declared contract: required fields are present, predictions are well formed, and requested actions stay within the permitted interface. *Grounders* own empirical facts; when an accepted event names a metric, diff, exit code, or runtime, the harness obtains that value from execution and writes the grounded value into the trace, so the agent's claim cannot substitute for a measurement. *Hooks* run on accepted events or reactions for side effects a workflow needs—scheduling a job, updating a dashboard, dispatching a notification—without becoming part of the authorization chain. The same harness serves different workflows by swapping these extensions; the program file and the extensions together form the auditable surface.

Who owns which fact. Validators own admissibility and grounders own measurements. The agent owns narrative and judgmental fields: hypotheses, mechanisms, rationales, requested actions, and magnitude bands. What remains the agent's is exactly what the trace foregrounds for review: what it believed, what it expected, what it asked to do, and how it explained the result after observing it.

Permission and audit are one path. A guardrail typically decides upstream whether an action is allowed, while a separate trace records downstream what happened. Behavioral code ties those objects together. A `Modifying.committed` event exists only because a reaction authorized it, and its provenance links reach the events the reaction checked. To audit why a commit was allowed, a reviewer walks backward through the same event chain the harness used to authorize it.

Trust boundary. The trusted base is the harness, the schemas in the program file, and the user-supplied validators and grounders. Their correctness is a review obligation, not something supplied by the agent. We assume the agent cannot bypass the harness, but we do not assume the agent is benign: a prompt-injected agent can still issue syntactically valid but semantically deceptive requests. Within that boundary, deception is observable in the trace—a missing provenance event, a prediction that does not match the grounded outcome, a refusal repeated under different framings—rather than hidden in model behavior.

Relation to guardrails. Our harness is a guardrail in the standard sense: a runtime that gates proposed agent actions against a declared policy. What differs is the policy language and its relationship to the trace. Existing systems answer *is this action permitted?*—NeMo Guardrails routes turns through Colang flows (Rebedea et al., 2023), GuardAgent uses an LLM monitor (Xiang et al., 2025), Progent enforces tool-call policies (Shi et al., 2025), Formal-LLM constrains plans against a grammar (Li et al., 2024), and CaMeL tracks capabilities through dataflow (Debenedetti et al., 2025). In each, the policy artifact and the trace are distinct, and the policy governs individual calls.

Behavioral code answers *what research event is this, what authorized it, what was predicted, and how did execution adjudicate the prediction?* The policy describes research structure rather than action permission; each accepted trace entry is a recorded execution of a declared reaction, with matched events as provenance; and predictions are required before experiments and checked against outcomes after. Table 2 summarizes these dimensions. The approaches are complementary: a deployment can use a permission-style guardrail to gate raw side effects and behavioral code to make research structure auditable.

3. Two experiments

We exercise behavioral code on two agentic research loops. Both use the same harness; only the behavioral program changes. The loops were chosen to surface different failure patterns that ordinary research artifacts—commit histories, metric files, chat logs—tend to flatten.

Model training. A small language-model training loop in the style of recent autoresearch demos (Karpathy, 2025): the agent edits `train.py`, runs training, measures `val_bpb`, and decides whether to keep each change. Across 54 hypotheses, 4 were kept; two early structural changes account for nearly all of the gain.

The trace makes the other 50 legible. Reading `Hypothesizing.formed` events in sequence, the agent revisited the same hyperparameters repeatedly—the same learning rate from above, then below, then from a different angle—with no event recording that it noticed. In a transcript this reads as persistence; in the trace it is search without memory, most likely an artifact of the agent operating beyond its effective context. The fix lives inside the existing

interface: validators the harness supports need not be static. A model-based validator on `Hypothesizing.formed` could compare each proposed hypothesis against prior ones in the log and refuse near-duplicates, the refusal entering the trace beside the entries it repeats (§5).

Performance engineering. An exploratory optimization loop for an unoptimized $O(N^2)$ C++ N-body simulator. Its program adds `Discovering`, which records a baseline noise floor, and `Profiling`, which requires each hypothesis to cite a recent measurement before proposing an optimization. Across twelve hypotheses, six changes are kept, four regress, and two fall below the noise floor—a separation a commit history would collapse to a single number.

One event illustrates the contribution directly. An OpenMP parallelization produced a 3.0× speedup against the agent's committed 5–10× prediction. The grounded measurement justified keeping the change; the comparison recorded `outcome_vs_prediction: violated`, because the stated mechanism was wrong. A commit history would log only the kept speedup. The trace preserves both the empirical success and the failed explanation as the distinct facts they are.

Two failure modes, one representation. The training loop shows an agent fooling itself by repetition; the performance loop shows it being right for the wrong reason. Both are erased by ordinary artifacts and visible the same way: predictions written before experiments, accepted changes bound to the events that authorized them. The harness gives validators a clean place to act on either pattern.

4. Legible traces and visible failures

The log lets a reviewer read the run as a research chain. Because accepted events carry structure, provenance, and grounded values, a reviewer can recover which hypothesis authorized a change and which measurement adjudicated it. The same structure makes visible failure modes a metric file or a chat history would hide.

Some failures are refused at write time. The harness rejects the proposed event and records the refusal with a reason: a result that cannot be traced back to a prior `Hypothesizing.formed` event is refused as an outcome without a recorded prediction. A malformed prediction, a missing measurement, or a request outside the declared interface fails the same way.

Other failures are admitted but marked, so a reviewer can search the trace for them after the run. A diff mismatch attaches to its event as a warning rather than a silent pass. A change whose measured gain falls below the noise floor records that the evidence is inconclusive. A metric that moves in the desired direction while contradicting the agent's stated mechanism is recorded as `outcome_vs_prediction: violated`, as in the OpenMP case. These are not necessarily reasons to discard the action. They are reasons the trace should not collapse "the metric improved" into "the agent understood why." A reviewer auditing the run filters for these markers and revisits the events they attach to.

Limitations. Behavioral code makes a run easier to inspect, but it does not make the run correct. The harness can require predictions, provenance, and outcome comparisons; it cannot decide whether a mechanism is true or whether a syntactically valid request is strategically deceptive. A sandbagging agent, or one with sleeper-agent behavior, can preserve an internally consistent log while still producing bad work. The trace is also only as expressive as the program: if a concept or validator is missing, the log cannot reveal what was never required. Grounded measurements can themselves be noisy or mis-specified. Our case studies show feasibility and trace quality, not scalability, reviewer workload, or robustness against a determined agent.

5. Discussion

This paper contributes the behavioral program and the harness that executes it, not yet a study of their effects. The next step is to measure what the trace changes. For reviewers, a within-subjects study could compare audits of the same runs as raw transcripts and as behavioral-code traces, measuring failure detection and time-to-detect on planted errors: missing measurements, repeated hypotheses, post-hoc explanations, and schema-valid but misleading behavior. For agents, the same log can be tested as structured memory. Surfacing prior hypotheses, failed predictions, open obligations, and near-duplicates at the relevant reaction could reduce repeated attempts and improve prediction calibration.

The interface should also be tested beyond the two loops here. Software repair, personal agents, and multi-agent systems all need permissions, provenance, and grounded outcomes, but expose different review-relevant structure. A personal agent, for example, may require sessions with different authority: one session that reads user state and proposes actions, another that performs bounded external actions, and another that revises the behavioral program itself. The point is not to put more trust in prompt text, but to make the scope of each role an inspectable part of the behavioral interface.

The broader question is how much of this structure survives a change of domain. If a new setting can be handled by replacing the behavioral program and validators while keeping the harness and log discipline fixed, behavioral code becomes more than a notation for autoresearch. It becomes

a way to make agent behavior reviewable wherever long-running agents act under delegated authority.

Acknowledgements

We thank the reviewers for their thoughtful feedback. We also thank Alcino Cunha, Abbas Mammadov, Arda Ozdemir, and Murad Aghazada for helpful discussions and feedback.

References

Abdelnabi, S., Greshake, K., Mishra, S., Endres, C., Holz, T., and Fritz, M. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec@CCS 2023)*, pp. 79–90. ACM, 2023. doi: 10.1145/3605764.3623985.

Baek, J., Jauhar, S. K., Cucerzan, S., and Hwang, S. J. ResearchAgent: Iterative research idea generation over scientific literature with large language models. In *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2025. arXiv:2404.07738.

Birgisson, A., Politz, J. G., Erlingsson, Ú., Taly, A., Vrable, M., and Lentczner, M. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *21st Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

Bowman, S. R., Hyun, J., Perez, E., Chen, E., Pettit, C., Heiner, S., Lukošiūtē, K., Askell, A., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., Olah, C., Amodei, D., Amodei, D., et al. Measuring progress on scalable oversight for large language models. *CoRR*, abs/2211.03540, 2022. doi: 10.48550/arXiv.2211.03540.

Chan, A., Ezell, C., Kaufmann, M., Wei, K., Hammond, L., Bradley, H., Bluemke, E., Rajkumar, N., Krueger, D., Kolt, N., Heim, L., and Anderljung, M. Visibility into AI agents. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT)*. ACM, 2024. arXiv:2401.13138.

Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., and Tramèr, F. Defeating prompt injections by design. *CoRR*, abs/2503.18813, 2025. doi: 10.48550/arXiv.2503.18813.

Fowler, M. Event sourcing. https://martinfowler.com/eaaDev/EventSourcing.html, 2005. Enterprise Application Architecture pattern.

Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., Wallach, H. M., Daumé III, H., and Crawford, K. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021. doi: 10.1145/3458723. arXiv:1803.09010.

Gottweis, J., Weng, W.-H., Daryin, A., Tu, T., Palepu, A., Sirkovic, P., Myaskovsky, A., Weissenberger, F., Rong, K., Tanno, R., Saab, K., Popovici, D., Blum, J., Zhang, F., Chou, K., Hassidim, A., Gokturk, B., Vahdat, A., Kohli, P., Matias, Y., Carroll, A., Kulkarni, K., Tomasev, N., Guan, Y., Karthikesalingam, A., Natarajan, V., et al. Towards an AI co-scientist. *CoRR*, abs/2502.18864, 2025. doi: 10.48550/arXiv.2502.18864.

Greenblatt, R., Shlegeris, B., Sachan, K., and Roger, F. AI control: Improving safety despite intentional subversion. *CoRR*, abs/2312.06942, 2023. doi: 10.48550/arXiv.2312.06942.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024. arXiv:2308.00352.

Huang, Q., Vora, J., Liang, P., and Leskovec, J. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024. arXiv:2310.03302.

Hubinger, E., Denison, C., Mu, J., Lambert, M., Tong, M., MacDiarmid, M., et al. Sleeper agents: Training deceptive LLMs that persist through safety training. *CoRR*, abs/2401.05566, 2024. doi: 10.48550/arXiv.2401.05566.

Irving, G., Christiano, P., and Amodei, D. AI safety via debate. *CoRR*, abs/1805.00899, 2018. doi: 10.48550/arXiv.1805.00899.

Jackson, D. Towards a theory of conceptual design for software. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pp. 282–296. ACM, 2015. doi: 10.1145/2814228.2814248.

Jackson, D. *The Essence of Software: Why Concepts Matter for Great Design*. Princeton University Press, 2021. ISBN 9780691225388.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

Karpathy, A. autoresearch: Give an AI agent a small but real LLM training setup and let it experiment autonomously overnight. https://github.com/karpathy/autoresearch, 2025. GitHub repository.

Kim, B., Wattenberg, M., Gilmer, J., Cai, C., Wexler, J., Viégas, F., and Sayres, R. Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (TCAV). In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pp. 2668–2677, 2018. arXiv:1711.11279.

Koh, P. W., Nguyen, T., Tang, Y. S., Mussmann, S., Pierson, E., Kim, B., and Liang, P. Concept bottleneck models. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, pp. 5338–5348, 2020. arXiv:2007.04612.

Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7): 558–565, 1978. doi: 10.1145/359545.359563.

Laurie, B., Langley, A., and Kasper, E. Certificate transparency. Request for Comments 6962, Internet Engineering Task Force, 2013. https://www.rfc-editor.org/rfc/rfc6962.

Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., Corsar, D., Garijo, D., Soiland-Reyes, S., Zednik, S., and Zhao, J. PROV-O: The PROV ontology. W3C recommendation, World Wide Web Consortium (W3C), 2013. URL https://www.w3.org/TR/prov-o/.

Li, Z., Xu, W., Shi, H., Zhang, Y., and Yan, Y. Formal-LLM: Integrating formal language and natural language for controllable LLM-based agents. *CoRR*, abs/2402.00798, 2024. doi: 10.48550/arXiv.2402.00798.

Lu, C., Lu, C., Lange, R. T., Foerster, J., Clune, J., and Ha, D. The AI scientist: Towards fully automated open-ended scientific discovery. *CoRR*, abs/2408.06292, 2024. doi: 10.48550/arXiv.2408.06292.

Meng, E. and Jackson, D. What you see is what it does: A structural pattern for legible software. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2025. ACM, 2025. doi: 10.1145/3759429.3762628. arXiv:2508.14511.

Miller, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, 2006. URL https://erights.org/talks/thesis/.

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D., and Gebru, T. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency (FAT*)*, pp. 220–229. ACM, 2019. doi: 10.1145/3287560.3287596. arXiv:1810.03993.

Nathani, D., Madaan, L., Roberts, N., Bashlykov, N., Menon, A., Moens, V., Budhiraja, A., Magka, D., Vorotilov, V., Chaurasia, G., Hupkes, D., Cabral, R. S., Shavrina, T., Foerster, J., Bachrach, Y., Wang, W. Y., and Raileanu, R. Meta MLGym and MLGym-Bench: A new framework and benchmark for LLM agents on AI research tasks. *CoRR*, abs/2502.14499, 2025. doi: 10.48550/arXiv.2502.14499.

Nosek, B. A., Ebersole, C. R., DeHaven, A. C., and Mellor, D. T. The preregistration revolution. *Proceedings of the National Academy of Sciences*, 115(11):2600–2606, 2018. doi: 10.1073/pnas.1708274114.

pi.dev contributors. pi.dev: a minimal, extensible harness for LLM coding agents. https://pi.dev, 2025. Anonymised reference for review.

Räuker, T., Ho, A., Casper, S., and Hadfield-Menell, D. Toward transparent AI: A survey on interpreting the inner structures of deep neural networks. In *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pp. 464–483, 2023. doi: 10.1109/SaTML54575.2023.00039. arXiv:2207.13243.

Rebedea, T., Dinu, R., Sreedhar, M. N., Parisien, C., and Cohen, J. NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In *EMNLP System Demonstrations*, 2023.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Sandall, T. Open policy agent: Policy-based control for cloud native environments. https://www.openpolicyagent.org/, 2019. CNCF graduated project; Rego policy language.

Schmidgall, S., Su, Y., Wang, Z., Sun, X., Wu, J., Yu, X., Liu, J., Liu, Z., and Barsoum, E. Agent laboratory: Using LLM agents as research assistants. *CoRR*, abs/2501.04227, 2025. doi: 10.48550/arXiv.2501.04227.

Shi, T., Chen, J., Yan, Z., Zhang, J., Zhang, W., Lou, Q., Wang, S., and Song, D. Progent: Programmable privilege control for LLM agents. *CoRR*, abs/2504.11703, 2025. doi: 10.48550/arXiv.2504.11703.

Si, C., Yang, D., and Hashimoto, T. Can LLMs generate novel research ideas? A large-scale human study with 100+ NLP researchers. *CoRR*, abs/2409.04109, 2024. doi: 10.48550/arXiv.2409.04109.

Starace, G., Jaffe, O., Sherburn, D., Aung, J., Chan, J. S., Maksin, L., Dias, R., Mays, E., Kinsella, B., Thompson, W., Heidecke, J., Glaese, A., and Patwardhan, T. PaperBench: Evaluating AI's ability to replicate AI research. *CoRR*, abs/2504.01848, 2025. doi: 10.48550/arXiv.2504.01848.

van der Weij, T., Hofstätter, F., Jaffe, O., Brown, S. F., and Ward, F. R. AI sandbagging: Language models can strategically underperform on evaluations. *CoRR*, abs/2406.07358, 2024. doi: 10.48550/arXiv.2406.07358.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., and Wen, J.-R. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024a. doi: 10.1007/s11704-024-40231-1. arXiv:2308.11432.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. OpenHands: An open platform for AI software developers as generalist agents. *CoRR*, abs/2407.16741, 2024b. doi: 10.48550/arXiv.2407.16741.

Wijk, H., Lin, T., Becker, J., Jawhar, S., Parikh, N., Broadley, T., Chan, L., Chen, M., Clymer, J., Dhyani, J., Ericheva, E., Garcia, K., Goodrich, B., Jurkovic, N., Karnofsky, H., Kinniment, M., Lajko, A., Nix, S., Sato, L., Saunders, W., Taran, M., West, B., and Barnes, E. RE-Bench: Evaluating frontier AI r&D capabilities of language model agents against human experts. *CoRR*, abs/2411.15114, 2024. doi: 10.48550/arXiv.2411.15114.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *CoRR*, abs/2308.08155, 2023. doi: 10.48550/arXiv.2308.08155.

Wu, Y., Nie, A., Dong, K., Velagapudi, P., Duchenne, O., Ben-Nasr, M., Bahl, S., Ermon, S., Ma, T., Weinberger, K. Q., et al. AIDE: AI-driven exploration of machine learning engineering. *CoRR*, abs/2502.13138, 2025. doi: 10.48550/arXiv.2502.13138.

Xiang, Z., Zheng, L., Li, Y., Hong, J., Li, Q., Xie, H., Zhang, J., Xiong, Z., Xie, C., Yang, C., Song, D., and Li, B. GuardAgent: Safeguard LLM agents by a guard agent via knowledge-enabled reasoning. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. arXiv:2405.15793.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Appendix

A. Demo, source, and concrete artifacts

A working prototype is online at https://interpretable-autoresearch-icml.pages.dev/, with source at https://anonymous.4open.science/r/interpretable-autoresearch-icml/. Both URLs are anonymized for review. The demo contains the two runs used in the paper: a small language-model training loop and an N-body performance-engineering loop. A reader can inspect the event stream, expand typed events, and follow provenance links backward from a decision to the hypothesis, prediction, measurement, and reaction that shaped it.

The prototype separates three roles. `program.md` is the behavioral program: it defines the concepts, events, and reactions that make up the loop. This is the file a reviewer edits when the agent's behavior should change. `events.jsonl` is the official append-only trace accepted by the harness. Each event records an action, its arguments, its emitter, and links to earlier events. In the implementation these links are stored as `caused_by`; in the paper we call them provenance links, because they record checked dependency rather than proving causal truth. `hooks.py` is the trusted adapter between the behavioral program and the outside world. It is ordinary human-owned Python code that decides which claims are admissible, which facts must be grounded by execution, and which inconsistencies should remain visible as warnings.

`program.md` says what kind of research process is allowed. `events.jsonl` records the process that was actually accepted. `hooks.py` defines where the agent's prose stops and trusted checks begin.

Concretely, the prototype is built on pi.dev's Pi coding-

agent infrastructure, which supplies the underlying agent loop and extension surface. Our prototype layers the paper's behavioral program, append-only accepted trace, and trusted domain hooks on top of that substrate (pi.dev contributors, 2025).

B. Related work

Autonomous research systems and oversight mechanisms already support long-running agentic work. Their traces usually record actions, outputs, and scores, while leaving the governing rule, prior prediction, and execution-based adjudication implicit. Behavioral code makes those relations part of the research artifact. A behavioral program declares research concepts and reactions; a harness enforces those declarations at write time, so the accepted log records the authorization chain and the predictions carried by each event.

Autonomous research and software agents. Recent systems show that language-model agents can carry out extended research and engineering work. Autoresearch demos, The AI Scientist, Agent Laboratory, and AIDE run loops over ideation, experimentation, and reporting (Karpathy, 2025; Lu et al., 2024; Schmidgall et al., 2025; Wu et al., 2025). AI Co-Scientist, ResearchAgent, and large-scale studies of LLM-generated ideas emphasize ideation, critique, and refinement (Gottweis et al., 2025; Baek et al., 2025; Si et al., 2024). Software-engineering agents and orchestration systems such as SWE-agent, OpenHands, MetaGPT, AutoGen, and ReAct provide interfaces, roles, tools, and multi-step control (Yang et al., 2024; Wang et al., 2024b; Hong et al., 2024; Wu et al., 2023; Yao et al., 2023). Benchmarks including MLAgentBench, RE-Bench, PaperBench, MLGym, and SWE-bench measure progress on machine-learning engineering, paper replication, and real software tasks (Huang et al., 2024; Wijk et al., 2024; Starace et al., 2025; Nathani et al., 2025; Jimenez et al., 2024). These systems motivate our setting. They usually evaluate the agent or final artifact rather than the fit between a run and a declared behavioral program. Behavioral code can sit beneath a research agent or benchmark harness, exposing which `Hypothesizing`, `Modifying`, `Experimenting`, `Evaluating`, and `Logging` events were permitted, which reaction authorized each transition, and which prediction was adjudicated.

Concept design and our use of concepts. Agent research traces often merge research events, permissions, explanations, and tool calls into one stream. That makes it difficult to recover which rule authorized a change or what prediction was being tested. We use Jackson's theory of concepts to separate the named activities a user recognizes from the synchronizations that relate them (Jackson, 2015; 2021). In Meng and Jackson's legible software pattern, a concept should be small enough to read locally, while synchronizations describe dependencies among concepts (Meng & Jackson, 2025). Behavioral code adapts that pattern to agent loops. `Hypothesizing` owns formed hypotheses and predictions; `Modifying` owns proposed and applied changes; `Experimenting` owns planned and run experiments; `Evaluating` owns grounded measurements; `Logging` owns comparisons between outcomes and prior predictions; and `Communicating` owns reports to external stakeholders. Together, these concepts give reviewers a vocabulary for predictions before runs, changes and experiments that test them, grounded measurements after execution, and reports drawn from the accepted record. Reactions carry the cross-concept commitments: a committed change must be bound to a prior hypothesis and a planned experiment by a checked transition. The concepts are review terms for admissible research states, not implementation modules.

Guardrails, policy languages, and authorization. Guardrail systems make language-model applications safer by routing or rejecting proposed behavior. NeMo Guardrails provides programmable Colang flows; GuardAgent uses a separate guard agent; Progent enforces privilege-control policies over tools; Formal-LLM constrains plans with a formal language; and CaMeL uses dataflow capabilities to resist prompt injection (Rebedea et al., 2023; Xiang et al., 2025; Shi et al., 2025; Li et al., 2024; Debenedetti et al., 2025). Authorization systems such as capabilities, macaroons, and policy languages likewise show how explicit rules can mediate access (Miller, 2006; Birgisson et al., 2014; Sandall, 2019). The harness gates behavior, with a research event as its unit of control: a `Hypothesizing`, `Experimenting`, or `Logging` action, rather than a raw tool call such as running a script or committing a file. The policy artifact and the trace are joined. When an event is accepted, the trace records which reaction authorized it and which earlier events satisfied that reaction's premises. To audit why a commit was allowed, a reviewer walks backward through the same provenance chain the harness checked.

Provenance, append-only logs, and scientific commitments. The event log borrows from established audit patterns while leaving semantic correctness to review. Lamport ordering, certificate transparency, PROV-O, and event sourcing treat append-only records and explicit dependency edges as ways to make distributed or temporal behavior inspectable (Lamport, 1978; Laurie et al., 2013; Lebo et al., 2013; Fowler, 2005). Behavioral code applies that discipline to agent loops: the log is a sequence of typed concept events accepted under declared reactions. The required prediction before a measurement mirrors preregistration in empirical science, where hypotheses and analysis plans are committed before data are observed (Nosek et al., 2018). The analogy

Table 1. Behavioral code compared with agent and research systems: what they make visible and how behavioral code structures the trace differently for auditability.

Line of work	Main artifact	What is visible	Relation to behavioral code
Research agents (Karpathy, 2025; Lu et al., 2024; Schmidgall et al., 2025; Wu et al., 2025)	Prompt and scaffolding code for ideation, experimentation, and reporting	Runs, reports, commits, and metrics	Behavioral code declares which research events may occur in a reaction-based program, and the accepted trace records which reaction authorized each event and what provenance it carried.
Software agents (Yang et al., 2024; Wang et al., 2024b; Hong et al., 2024; Wu et al., 2023)	Agent-computer interfaces, roles, tools, and conversations	Tool calls, messages, patches, and task outcomes	The harness is orthogonal to the agent framework: it gates what enters the official research trace rather than replacing the model loop.
Agent benchmarks (Huang et al., 2024; Wijk et al., 2024; Starace et al., 2025; Nathani et al., 2025; Jimenez et al., 2024)	Task environments and scoring procedures	Final scores, artifacts, and sometimes transcripts	Behavioral code adds audit questions beyond final score: which reaction authorized each research event, what prediction preceded it, and how execution adjudicated that prediction.
Structured discovery (Romera-Paredes et al., 2024)	A generate–evaluate search loop	Candidate programs and evaluator scores	Concepts and reactions generalize the loop structure from a hard-coded cycle to a reviewable behavioral program.

has limits. The harness cannot verify that a mechanism is true or that an analysis plan is scientifically sufficient. It can still rule out a purely post-hoc trace. If a researcher changes the prediction after seeing the result, the accepted log shows a divergence between the prior prediction and grounded outcome. A metric improvement with the wrong mechanism becomes an explicit disagreement rather than a silent success.

Transparency, interpretability, and oversight. Here, a *concept* belongs to the agent loop rather than to concept-based model interpretability. TCAV and concept bottleneck models use concepts as human-interpretable directions or variables inside a model (Kim et al., 2018; Koh et al., 2020). Mechanistic interpretability studies internal neural-network structure (Räuker et al., 2023). Model cards and datasheets document models or datasets as artifacts (Mitchell et al., 2019; Gebru et al., 2021). Behavioral code targets the layer where an agent uses models and tools over time. At that layer the agent forms a hypothesis, designs an experiment, runs it, and interprets the result. Behavioral code makes that sequence auditable, while the truth of each individual claim remains a harder problem. This loop-level focus connects to work on visibility into AI agents and scalable oversight (Chan et al., 2024; Bowman et al., 2022; Irving et al., 2018). It also shows the approach's limits. Prompt injection, sleeper-agent behavior, sandbagging, or intentional subversion can still produce syntactically valid traces (Abdelnabi et al., 2023; Hubinger et al., 2024; van der Weij et al., 2024; Greenblatt et al., 2023). Behavioral code gives reviewers a structural object to audit: the behavioral program, the accepted log, provenance links, and grounded outcome comparisons.

C. Reaction syntax and one event chain

Reaction syntax is a small pattern language over the accepted event log. The `when` clause is matched against the event currently being considered. Variables beginning with `?` are binders: their first occurrence binds to a value, and later occurrences must refer to that same value. The `where` clauses query earlier accepted events using those bindings and may introduce additional ones. The `then` clauses do not directly mutate the log; they name the requests that become authorized once the `when` and all `where` clauses match.

The following reaction gates a code change on an earlier hypothesis and a planned experiment:

```
when:  Modifying.applied(?change)
where: Hypothesizing.formed(?hypothesis,
                            supports: ?change)
       Experimenting.planned(?experiment,
                              hypothesis: ?hypothesis)
then:  authorize Modifying.commit(?change)
       authorize Experimenting.run(?experiment)
```

In this example, `Modifying.applied(?change)` binds the concrete change under consideration. The first `where` clause finds a recorded hypothesis that supports that same change and binds it as `?hypothesis`; the second finds a planned experiment for that hypothesis and binds `?experiment`. Only then may the harness authorize the commit and experiment run. The authorization is therefore a recorded rule execution whose premises name the hypothesis and experiment that must already exist, instead of appearing only as nearby prose in a transcript.

Table 2. Guardrails gate proposed actions. Behavioral code gates research events into an auditable trace in which each event carries the reaction that authorized it and the predictions it adjudicates.

Approach	Governed unit	Policy-trace relation	What behavioral code adds
NeMo Guardrails (Rebedea et al., 2023)	Conversation turns and flows	Colang flow is separate from the downstream trace	Research event types, provenance, and prior predictions are recorded in one accepted log.
GuardAgent (Xiang et al., 2025)	Proposed agent actions	Monitor reasoning is external to the agent's work trace	A declared program whose reaction executions become the trace being audited.
Progent (Shi et al., 2025)	Tool privileges	Privilege checks mediate tool access	Cross-event commitments such as "this experiment tests that hypothesis" and "this outcome adjudicates that prediction."
Formal-LLM (Li et al., 2024)	Generated plans	Plan syntax is constrained by a formal grammar	Domain structure—research concepts tied by reactions—and grounded empirical facts beyond syntactic admissibility.
CaMeL (Debenedetti et al., 2025)	Capabilities through dataflow	Security labels control information and authority flow	Complementary loop-level provenance: why this research event was authorized now.
Behavioral code	Research events	The policy artifact is the program whose executions are recorded in the log	A research event's acceptance, authorizing reaction, prediction, and measured outcome all appear in the trace.

A representative pair of events from the training loop is:

```
{"event_id":"evt-00041",
 "by":"autoresearch-agent",
 "action":"Hypothesizing.formed",
 "args":{"hypothesis_id":"h-12",
   "description":"WARMDOWN_RATIO 0.10 -> 0.0",
   "prediction":{"direction":"val_bpb_down",
     "magnitude":"0.0005-0.004",
     "mechanism":"short runs are high-LR-update
     limited;
                  cooldown is wasted budget",
     "side_effects":"possible late-step instability"}},

 "caused_by":["evt-00038"]}

{"event_id":"evt-00045",
 "by":"iar-harness",
 "action":"Logging.recorded",
 "args":{"experiment_id":"exp-003",
   "val_bpb":2.061059,
   "outcome_vs_prediction":
     "matched: 0.003302 drop, inside band;
      no late-step instability observed"},
 "caused_by":["evt-00044","evt-00041"]}
```

The logged result links to both the measurement that produced the metric and the hypothesis whose prediction it adjudicates. A reviewer can therefore recover the prediction being tested without trusting the agent's later prose explanation.

D. What `hooks.py` owns

The main paper describes the harness abstractly. In the prototype, the practical trust boundary is `hooks.py`. The behavioral program can require a prediction before an experiment; `hooks.py` decides what counts as a complete prediction. The program can require a measurement; `hooks.py` runs the training or benchmark command and extracts the measured value. The program can permit a keep or revert decision; `hooks.py` checks that decision against the grounded measurement and domain-specific rules.

The hooks have three roles.

First, validators reject events before they enter the accepted log. For example, the training loop rejects incomplete hypotheses and logs that fail to compare the observed outcome against the prior prediction. The performance loop adds domain checks for discovery, noise floors, profile attribution, and measurement status. These checks are trusted code rather than model-written advice.

Second, grounders produce empirical facts. In the training loop, grounders run the training command, extract `val_bpb` from the run output, and ground reverts through the repository state. In the performance loop, grounders run the benchmark and return timing and correctness information. The agent may ask for a measurement; the accepted metric is the value produced by the grounder.

Third, observers surface reviewable inconsistencies that need not be fatal. A claimed file path can disagree with the observed diff. A declared performance significance can disagree with the measured noise floor. Such cases can be recorded as warnings rather than silently erased. A stricter deployment could promote some warnings to validators.

The following schematic fragment illustrates the division:

```
@validate("Hypothesizing.formed")
def complete_prediction(event, log):
    pred = event.args["prediction"]
    require(pred["direction"])
    require(pred["magnitude"])
    require(pred["mechanism"])
    require(pred["side_effects"])

@ground("Evaluating.measured")
def measure_training(request, workspace):
    run("uv run train.py", cwd=workspace)
    return {
        "val_bpb": parse_val_bpb("run.log"),
        "exit_code": last_exit_code()
    }
```

```
@validate("Logging.recorded")
def compare_against_prediction(event, log):
    prediction = lookup(event.caused_by, "
     Hypothesizing.formed")
    require(event.args["outcome_vs_prediction"])
    require_mentions_prediction(event.args["
     outcome_vs_prediction"],
                                prediction)

@on("Modifying.applied")
def warn_if_claimed_diff_mismatches(event, workspace):
    if not claimed_paths_match_git_diff(event,
     workspace):
        warn("claimed files do not match observed diff
     ")
```

This schematic fragment is illustrative rather than the exact shipped code. It shows the boundary: the agent owns the hypothesis and explanation; the validator owns admissibility; the grounder owns measured facts; the observer makes non-fatal inconsistencies visible.

E. What the accepted log guarantees

The harness makes acceptance into the official trace a checked transition, while leaving the agent's choices nondeterministic. Given the current log, the behavioral program, and the trusted hooks, a proposed event is either accepted, refused, grounded, or surfaced with warnings. The LLM may choose among eligible actions and may write free-form fields. An off-contract event enters the accepted trace only by passing the harness boundary.

In the shipped loops, this means the accepted trace has several structural properties. Hypotheses contain predictions before experiments run. Logged outcomes compare observations against those predictions after the run. Measurements come from execution rather than self-report. Changes are tied to the hypothesis and experiment that authorized them. Rejections and warnings remain in the trace rather than disappearing into an invisible runtime failure.

These are deliberately weaker than correctness guarantees. The harness cannot determine whether a natural-language mechanism is true, whether a hypothesis is scientifically useful, or whether a well-formed request is strategically deceptive. It moves several important questions from the agent's private behavior into a record a reviewer can inspect: What did the agent predict? Which event authorized the action? What did execution measure? Did the outcome agree with the prediction? Was the decision to keep, discard, or warn grounded in the declared interface?

F. How the two hooks files differ

The two case studies share the same harness but use different trusted checks.

In the model-training loop, `hooks.py` is centered on a narrow experimental cycle. It checks that hypotheses include direction, magnitude, mechanism, and side effects; runs training; extracts `val_bpb`; and grounds keep-or-revert behavior against the measured result and repository state. This makes the training log readable as a sequence of predictions tested against measurements.

In the performance-engineering loop, `hooks.py` adds checks that are specific to optimization work. The agent must establish a baseline and noise floor, cite profiling evidence before proposing optimizations, preserve benchmark correctness, and mark measured changes relative to run-to-run variance. This makes the performance log readable as claims about bottlenecks, mechanisms, and measurement significance, with speedups or regressions attached to those claims.

Human-owned hooks provide the domain-specific trust boundary. The behavioral program supplies the legible structure of the loop. Changing domains should preserve the idea of behavioral code while changing what the trusted checks know how to validate and ground.

G. Residual risks

The architecture protects the official trace rather than every possible side effect of a coding agent. If the agent has raw filesystem or shell permissions outside the harness boundary, it may still create unlogged effects. If `hooks.py` is weak, the accepted log will inherit those weaknesses. If a model writes a syntactically valid but misleading mechanism, the harness can preserve that mechanism and compare it to outcomes. It cannot decide that the prose is truthful.

The guarantee is structural rather than semantic. The accepted log can show that predictions preceded outcomes, that empirical fields were grounded, that events carried provenance links, and that outcomes were compared against prior predictions. That is weaker than saying the agent did good science. It is stronger than a transcript, a commit history, or a metrics file.