Beyond Pass Rate: A Hierarchy of Behavioral Consistency for LLM Agents

Tian Lu*
Northeastern University
lu.tian1@northeastern.edu

Zikai Wang*
Northeastern University
wang.zikai1@northeastern.edu

Cheng Tan
Northeastern University
c.tan@northeastern.edu

Abstract

LLM agents now act on real systems, and running the same task more than once is increasingly routine. Traditional metrics such as pass rate score how often the agent succeeds but say nothing about whether those runs agreed on how they got there. We propose a complementary axis, behavioral *consistency*: a property of the trajectory distribution from which an agent draws when invoked on the same task repeatedly. We formalize it as a totally ordered hierarchy of five levels and a metric $\textsc{consistency}^N_{lv}$ for the probability that N independent runs agree at level lv. Across 17 models on 10 system-administration tasks, pass rates routinely exceed 80%, yet two random runs agree on the sequence of state-modifying actions (i.e., $\textsc{consistency}^2_{lv3}$) only about 39% of the time on average. Even the frontier model reaches only $\textsc{consistency}^2_{lv3} \approx 57\%$. Case studies on certificate rotation and zero-downtime upgrades further show that the same pass rate can hide trajectory distributions with very different deployment-risk profiles.

1 Introduction

LLM agents now act on systems where actions have consequences. Unlike chatbots, these tasks produce *trajectories*: sequences of tool calls that read and modify system state. Running the same task with the same agent more than once is now routine in regression testing, maintenance, and audits [10, 14, 17]. This raises a question that chatbots have never had to answer: *do we get the same behavior every time?*

The gap. None of today's evaluation metrics answers this question. The dominant one, *pass rate*, scores how often tasks succeed and is silent on whether the runs that produced it agreed on how they got there. What we want instead is behavioral *consistency*: a property of the *distribution over trajectories* from which an agent draws when invoked on the same task repeatedly. Pass rate and behavioral consistency can diverge sharply. In our experiments across 17 models and 10 system-administration tasks (§4), pass rates routinely exceed 80%. Yet when we sample two runs from the same task, the probability that they agree on the resolution (that is, that they issue the same sequence of state-modifying actions) is only about 39%.

*Equal contribution.

One metric will not work. How much consistency an application needs is not universal. A single metric cannot serve all regimes; what is needed is a framework that lets a developer specify which level of consistency the application requires and measure whether a given model achieves it.

Our empirical focus is system administration, as such tasks are well suited for checking consistency: they are inherently multi-step, state-mutating, and admit several plausible strategies of differing risk. The framework itself, however, applies to any tool-using agent.

Our solution: a consistency hierarchy. We define five levels of behavioral consistency for tool-using LLM agents, ordered from strict to relaxed: token-identical, action-equivalent, effect-equivalent, state-equivalent, and outcome-equivalent (§3, Table 1). Each level implies all weaker levels, so the levels form a total order on behavioral guarantees. We implement an automated checker that assigns trajectories to per-level equivalence classes via tool-argument canonicalization, read/write classification, and state-projection comparison.

2 Motivation and Related Work

The question of whether LLM agents behave consistently across repeated runs has received growing attention, but existing work fragments into approaches that each capture only part of the picture.

2.1 Measuring Agent Consistency and Reliability

The most widely adopted consistency measure is pass^k [1, 17]; CAR-bench [9] similarly quantifies the gap between occasional and consistent success. These metrics establish the *severity* but treat consistency as binary, offering no insight into *where* divergence occurs. Richer frameworks decompose reliability into multiple axes [5, 8, 14], and other studies analyze trajectory variance directly [2, 11, 12], but the former define *independent* metrics rather than an *ordered hierarchy* and the latter use ad hoc statistics rather than formal equivalence classes.

2.2 Tool-Using Agent Benchmarks

The closest benchmarks to our domain are Terminal-Bench [13] and ITBench [7], covering containerized CLI tasks and IT-automation scenarios across SRE, security/compliance, and FinOps; both use execution-based

evaluation but report single-run results. We adopt Terminal-Bench's agent framework and task format, extending it with the consistency dimension neither measures.

2.3 Techniques for Achieving Consistency

Prior work treats consistency as an engineering problem to be enforced, with techniques targeting different levels. Batch-invariant GPU kernels [6] and FP32-throughout computation [18] eliminate token-level non-determinism (Level 1); we study the behavioral non-determinism that remains. Structural testing [10] mocks LLM outputs for regression (Level 1–2), while AgentRR [3] records and replays trajectories at multiple abstraction levels, from action sequences (Level 2) to higher-level procedures (Level 3). None specify which level they guarantee for unconstrained agents, nor measure it.

Positioning. Our work unifies these threads with a totally ordered hierarchy and an automated checker that classifies *where in the execution* divergence occurs, supplying both the theoretical structure the frameworks lack and the granularity the empirical studies lack.

3 Consistency Hierarchy

We formalize behavioral consistency as a five-level hierarchy, where each level captures a progressively weaker notion of consistency.

3.1 Setup

Interaction model. We model an agent as a stochastic policy $\pi_\theta : \mathcal{U} \times (\mathcal{A} \times \mathcal{O})^* \to \Delta(\mathcal{A})$ that takes a task instruction and an interaction history and returns a distribution over next actions (tool calls or text emissions). The environment responds deterministically: given action a_t in state s_t, the next state is $s_{t+1} = T(s_t, a_t)$ and the observation is $o_t = \Omega(s_t, a_t)$. This ensures that all variation across repeated runs is attributable solely to the stochasticity of π_θ, not to environmental randomness.

Trajectory and group. A *trajectory* $\tau = \langle (a_1, o_1), \ldots, (a_n, o_n), r \rangle$ records the full sequence of actions, observations, and the final response from one run. We write $\text{actions}(\tau) = (a_1, \ldots, a_n)$ for the action subsequence, $\text{tokens}(\tau)$ for the full sequence of agent-emitted tokens (tool-call names, tool-call arguments, and final response), and $s_f(\tau)$ for the final state. A *group* $G = \{\tau_1, \ldots, \tau_k\}$ collects k trajectories produced by the same policy, instruction, and environment (reset to initial state s_0 before each run). Consistency is always measured over a group.

Read and write actions. Each tool in the environment is annotated as either *read* (a^R: does not mutate tracked state, e.g., file read, search) or *write* (a^W: mutates state, e.g., file write, config change, service restart). This classification is central to Level 3 and must be specified in the evaluation setup.

Correctness, state projection, and outcome projection. A *correctness predicate* $C(\tau)$ is an execution-based check that returns TRUE when the task specification is satisfied. A *state projection* $\rho : \mathcal{S} \to \mathcal{S}'$ extracts the task-relevant portion of environment state for comparison (e.g., a directory-tree hash or normalized config diff), since comparing full system state is generally intractable. We additionally define an *outcome projection* $\sigma : \mathcal{S} \to \mathcal{S}''$ that extracts only the subset of state on which C depends, so that C factors as $C(\tau) = \tilde{C}(\sigma(s_f(\tau)))$ for some predicate $\tilde{C} : \mathcal{S}'' \to \{0, 1\}$. We require σ to be a coarsening of ρ: any two states equal under ρ are also equal under σ.

3.2 Consistency Levels

We define five levels, ordered from strictest to most relaxed. Level i implies all levels $j > i$: a group that is token-identical (Level 1) is necessarily also action-equivalent (Level 2), effect-equivalent (Level 3), state-equivalent (Level 4), and outcome-equivalent (Level 5), but not vice versa. Table 1 summarizes the hierarchy; Appendix A gives the full version with sub-levels, extension points, and a proof that the chain is strictly ordered.

Level 1: Token-identical. The strongest level requires that every externally emitted token is identical across runs. Achieving Level 1 requires deterministic inference (greedy decoding with batch-invariant kernels).

Level 2: Action-equivalent. Relaxing from tokens to semantics, Level 2 requires that the same tool calls appear in the same order with equivalent arguments, after canonicalization (JSON key-sorting, whitespace normalization, type coercion). Surrounding natural-language text and token segmentation may differ.

Level 3: Effect-equivalent. The key insight behind Level 3 is the separation of *diagnosis* from *treatment*. In many tasks, an agent first investigates the environment (reads) and then modifies it (writes). Different diagnostic strategies may be equally valid, but the corrective actions should be predictable. Level 3 requires that only the write actions match in sequence:

$$\forall\, \tau_i, \tau_j \in G : \quad \text{filter}_W\big(\text{actions}(\tau_i)\big) = \text{filter}_W\big(\text{actions}(\tau_j)\big),$$

where filter_W retains write actions with normalized arguments and preserves their relative order. Read actions, their ordering, and the total trajectory length may all vary.

Level 3 strictly implies Level 4 under deterministic dynamics, but not vice versa: a trajectory writing x=1; x=2 and one writing x=2 have identical final state but distinct write sequences. Appendix A.3 states this formally and gives the proof.

Level 4: State-equivalent. Relaxing further, Level 4 drops the requirement on write sequences and asks only that the final environment state matches under projection. This is the

Table 1. The consistency hierarchy. Each level implies all levels below it.

Level	Name	What must match	What may vary
1	Token-identical	All emitted tokens	–
2	Action-equivalent	Tool calls + args (normalized)	Surrounding text, tokenization
3	Effect-equivalent	Write actions (ordered)	Read actions, text, path length
4	State-equivalent	Final state under ρ	Actions, text
5	Outcome-equivalent	Outcome under σ	Final state outside σ, actions, text

projection-restricted analog of state equivalence under bisimulation [4]: trajectories producing ρ-identical final states are behaviorally indistinguishable from any consumer that observes only $\rho(s)$.

Level 5: Outcome-equivalent. The most relaxed level requires only that all runs produce the same outcome under the coarser projection σ. Because σ is a coarsening of ρ, Level 4 implies Level 5. Since C factors through σ, two trajectories satisfying Level 5 share the same correctness verdict; when that verdict is "pass," this corresponds exactly to τ-bench's pass$\hat{}k = 1.0$ on the group [17].

4 Experimental Evaluation

Setup. We evaluate 17 models on our benchmark consisting of 10 tasks. For each (model, task, parameter) combination, we run 25+ independent trials, resulting in 8,000+ total trajectories. Our tasks are inspired by Terminal-Bench [13], focusing on system-administration scenarios in a CLI environment. Task descriptions, projector definitions, the model list, and environment setup are in Appendices G and C.

Metrics. Given multiple repeated trials, we classify their trajectories into clusters: two trajectories belong to the same cluster if and only if they are considered consistent at that level. We summarize this distribution with $\textsc{consistency}^N$, the probability that N independently drawn trajectories from the same group all fall in the same cluster. When a consistency level lv is fixed, we write $\textsc{consistency}^N_{lv}$; for example, $\textsc{consistency}^2_{lv3}$ is the pairwise collision probability at Level 3 (effect-equivalent). Formally, let $p_1, \ldots, p_m$ be the probabilities that the agent produces a trajectory in each equivalence class at level lv; then

$$\textsc{consistency}^N_{lv} = \sum_i p_i^N.$$

In practice we approximate the probability from sampled trials using a without-replacement unbiased estimator (see Appendix E.2). We use N=2 throughout. We focus on Levels 3, 4, and 5 because they capture the practically relevant question: does the agent produce the same effect (or reach the same final state)?

Figure 1 reports $\textsc{consistency}^2$ at Levels 3, 4, and 5 alongside pass rate for 16 models and 10 tasks. Levels 1 and 2 are omitted: they are near-zero for all models and tasks under normal setups (Appendix F.1). Pass rates are tightly clustered: per-model averages span only 0.76 to 0.96, leaving little room to tell most models apart. Our key findings:

- **How consistent are current LLMs?** At Level 5, per-model averages span only 0.75 to 0.92, and several tasks pin every model at $\textsc{consistency}^2_{lv5}$=1.0; at this level a cell is mostly set by the task. At Level 4 the spread widens. At Level 3 the spread is widest, from 0.18 to 0.57, with the across-model median near 0.39: two random attempts from a typical model have less than a two-in-five chance of producing the same effect. As the equivalence tightens, the source of variation shifts from task to model: Level 5 discriminates by task more than by model, Level 3 does the opposite.
- **Within a family, larger models are more consistent.** The order is monotonic in size at Level 3 and Level 5 across all three closed-source families, and at Level 4 in OpenAI and Qwen; the only break is Anthropic at Level 4 (Sonnet 0.80 > Opus 0.76). The Qwen-3.5 dense series shows the gap most cleanly at Level 3: 0.45 (Plus) > 0.35 (27B) > 0.18 (4B). Pass rate is decoupled from scale in all three families (Sonnet beats Opus, Nano matches GPT-5.4, the two largest Qwen-dense models tie), so larger models settle on a tighter set of action sequences: a property pass rate alone does not surface.
- **MoEs are less consistent than dense models at the same parameter budget.** Within Qwen-3.5, the 32B-A3B MoE matches the 27B dense model on pass rate (0.84 vs. 0.83), but its $\textsc{consistency}^2_{lv3}$ (0.20) is much closer to the 4B dense model (0.18) than to the 27B dense (0.35). The other open MoEs in our study show the same pattern at Level 3 (DeepSeek-V3.2: 0.26; Kimi K2.5: 0.28; MiniMax M2.5: 0.46; GLM-5: 0.42), all sitting in the lower half of the Level 3 spread despite competitive pass rates. A plausible explanation is routing variance: a different expert mix on the next attempt produces a different action distribution, even when the final state matches.
- **Two cluster shapes recur across tasks.** When consistency is low, the per-cell distribution takes one of two recognizable shapes. *Mode A (dispersed):* on `cert-rotation` with Gemini 3.1 Pro, 12 of 25 attempts overwrite the certificate files with `cp -f` (no easy rollback) and 13 of 25 use `sed -i` to repoint the service configuration. Both

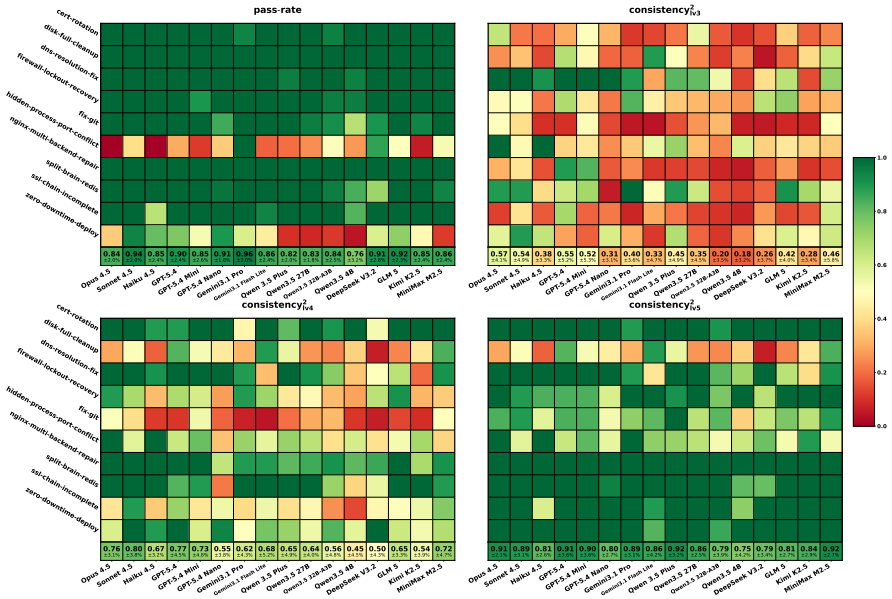

Figure 1. CONSISTENCY^2 at Levels 3 (effect-equivalent), 4 (state-equivalent), and 5 (outcome-equivalent), and pass rate across models and 10 tasks at default parameters. Each cell reports the point estimate with bootstrap 95% CIs (B=1,000). Per-cell breakdowns appear in Appendix F.1.

paths pass, but the rollback risk is opposite. *Mode B (dominant cluster with a tail):* on `zero-downtime-deploy` with Sonnet 4.5, 24 of 25 attempts follow the correct start-then-reload-then-kill order and one attempt kills v1 first, introducing a downtime window that a 25-trial evaluation can easily miss. Pass rate cannot discriminate Mode A from a healthy Mode B. Reporting the cluster distribution alongside scalar metrics is what makes the difference visible.

- **Higher temperature reduces consistency.** We locally host Qwen 3.5-27B and Gemma 4-31B and vary the temperature from 0 to 1.0. Across all tasks, CONSISTENCY^2 decreases as temperature increases (Figure 2). Higher temperature flattens the softmax distribution, and because agentic trajectories involve many sequential decisions, per-token entropy compounds across turns.
- **Patterns are task-dependent for reasoning.** We vary reasoning effort from `low` to `xhigh` at temperature 1.0 on GPT-5.4 Nano. Unlike temperature, reasoning effort produces *task-dependent* effects (Figure 3), governed by each task's *solution topology*: a fixed procedure, open-ended optional work, or a hidden optimal strategy that only deep reasoning discovers (Appendix F.4).
- **Consistency and correctness are orthogonal; structural features predict consistency.** At every level, per-model task reward correlates only weakly with CONSISTENCY^2 (r=+0.34, 0.28, 0.20 at Levels 3, 4, 5; all p>0.2), so pass rate says little about how concentrated the trajectory distribution is. Per task, the read/write ratio (Level 3, r=+0.697) and reasoning-token share (Levels 4–5, r=−0.852 at Level 5) are the strongest predictors (Appendix F.5).

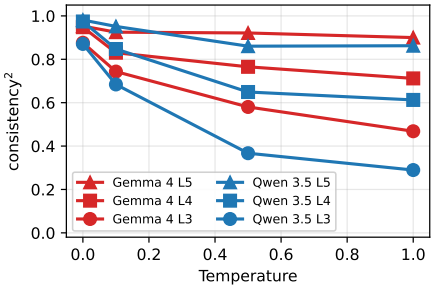

Figure 2. CONSISTENCY^2 at Levels 3 and 4 for Qwen 3.5-27B and Gemma 4-31B as sampling temperature varies from 0 to 1.0. Per-task breakdowns are reported in Appendix F.3.

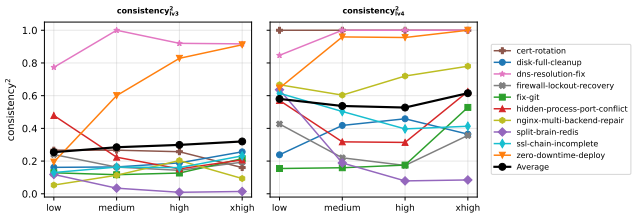

Figure 3. CONSISTENCY^2 at Levels 3 (effect-equivalent) and 4 (state-equivalent) for GPT-5.4 Nano as reasoning effort varies from `low` to `xhigh` at temperature 1.0.

References

[1] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. *arXiv preprint arXiv:2506.07982* (2025).

[2] Hongliu Cao, Ilias Driouich, and Eoin Thomas. 2026. Beyond Task Completion: Revealing Corrupt Success in LLM Agents through Procedure-Aware Evaluation. *arXiv preprint arXiv:2603.03116* (2026).

[3] Erhu Feng, Wenbo Zhou, Zibin Liu, Le Chen, Yunpeng Dong, Cheng Zhang, Yisheng Zhao, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. 2025. Get Experience from Practice: LLM Agents with Record & Replay. (2025). arXiv:2505.17716 [cs.LG] https://arxiv.org/abs/2505.17716

[4] Norm Ferns, Prakash Panangaden, and Doina Precup. 2004. Metrics for Finite Markov Decision Processes. In *Conference on Uncertainty in Artificial Intelligence*.

[5] Aayush Gupta. 2026. ReliabilityBench: Evaluating LLM Agent Reliability Under Production-Like Stress Conditions. *arXiv preprint arXiv:2601.06112* (2026).

[6] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. doi:10.64434/tml.20250910 https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[7] Saurabh Jha, Rohan R. Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Jae wook Ahn, Debanjana Kar, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. 2025. ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=jP59rz1bZk

[8] Raffi Khatchadourian. 2026. Replayable Financial Agents: A Determinism-Faithfulness Assurance Harness for Tool-Using LLM Agents. *arXiv preprint arXiv:2601.15322* (2026).

[9] Johannes Kirmayr, Lukas Stappen, and Elisabeth André. 2026. CAR-bench: Evaluating the Consistency and Limit-Awareness of LLM Agents under Real-World Uncertainty. *arXiv preprint arXiv:2601.22027* (2026).

[10] Jens Kohl, Otto Kruse, Youssef Mostafa, Andre Luckow, Karsten Schroer, Thomas Riedl, Ryan French, David Katz, Manuel P. Luitz, Tanrajbir Takher, Ken E. Friedl, and Céline Laurent-Winter. 2025. Automated Structural Testing of LLM-Based Agents: Methods, Framework, and Case Studies. In *2025 IEEE International Conference on Big Data (BigData)*. 1847–1856. doi:10.1109/BigData66926.2025.11401679

[11] Aman Mehta. 2026. Consistency Amplifies: How Behavioral Variance Shapes Agent Accuracy. *arXiv preprint arXiv:2603.25764* (2026).

[12] Aman Mehta. 2026. When Agents Disagree With Themselves: Measuring Behavioral Consistency in LLM-Based Agents. *arXiv preprint arXiv:2602.11619* (2026).

[13] Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. 2026. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=a7Qa4CcHak

[14] Stephan Rabanser, Sayash Kapoor, Peter Kirgis, Kangheng Liu, Saiteja Utpala, and Arvind Narayanan. 2026. Towards a Science of AI Agent Reliability. *arXiv preprint arXiv:2602.16666* (2026).

[15] The SGLang Team. 2025. Towards Deterministic Inference in SGLang and Reproducible RL Training. https://www.lmsys.org/blog/2025-09-22-sglang-deterministic/.

[16] The vLLM and TorchTitan Teams. 2025. No More Train-Inference Mismatch: Bitwise Consistent On-Policy Reinforcement Learning with vLLM and TorchTitan. https://vllm.ai/blog/bitwise-consistent-train-inference.

[17] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. In *International Conference on Learning Representations*. arXiv:2406.12045.

[18] Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. 2026. Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=Q3qAsZAEZw

Appendix

The appendix collects supporting material referenced from the main text. §A gives the full version of the consistency hierarchy: detailed preliminaries (interaction model, environment, action classification, projections), all five levels with sub-levels and extension points, and a proof that the levels form a strictly ordered chain. §B walks a single trial end-to-end to illustrate what each consistency level extracts from a trajectory. §C lists model snapshots, API and vLLM parameters, serve commands, and compute resources. §D reports licenses for third-party assets. §E gives formal definitions for CONSISTENCY^N, the bootstrap procedure, and the trajectory-level metrics used in the correlation study. §F contains per-(model, task) heatmaps at every consistency level, results under deterministic inference, full temperature and reasoning-effort sweeps, and per-metric correlation tables. §G describes each task's setup, success criterion, and projector.

A Full Consistency Hierarchy and Strict Ordering

This appendix gives the detailed version of the consistency hierarchy referenced from Section 3. §A.1 fixes the interaction model, environment definition, action classification, and projections. §A.2 restates the five levels with sub-levels, normalization assumptions, and extension points. §A.3 states explicit assumptions and proves that the levels form a strictly ordered chain (forward implications hold; converses fail).

A.1 Preliminaries

Interaction tuple. We use the tuple $(\mathcal{U}, \mathcal{S}, \mathcal{A}, \mathcal{O}, T, \Omega, s_0)$, where $\mathcal{U}$ is the instruction space (task prompts), $\mathcal{S}$ the environment state space, $\mathcal{A}$ the action space (tool calls or text emissions), $\mathcal{O}$ the observation space (tool outputs, errors, feedback), $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ the transition function, $\Omega : \mathcal{S} \times \mathcal{A} \to \mathcal{O}$ the observation function, and $s_0 \in \mathcal{S}$ the initial state. The agent does not directly access s; it acts on the basis of its own action and observation history.

Agent policy. The agent is a stochastic policy

$$\pi_\theta : \mathcal{U} \times (\mathcal{A} \times \mathcal{O})^* \to \Delta(\mathcal{A}),$$

where $\Delta(\mathcal{A})$ denotes the set of probability distributions over $\mathcal{A}$. The policy represents the full agent system (model, prompting, scaffolding, retrieval, orchestration), not only the underlying language model. Two agents that share an LLM but differ in scaffolding are different policies.

Determinism assumption (main setting). To isolate behavioral variability of the agent, we assume deterministic environment dynamics:

$$s_{t+1} = T(s_t, a_t), \qquad o_t = \Omega(s_t, a_t).$$

Under this assumption, repeated runs with the same instruction and reset state differ only through stochasticity in π_θ.

Definition 1 (Environment)**.** *An* environment $\mathcal{E} = (s_0, \mathcal{T}, \delta)$ *consists of the initial state s_0, a tool registry $\mathcal{T}$ (available tools, schemas, and access modes), and a stabilization policy $\delta \in \{\textsc{live}, \textsc{cached}, \textsc{simulated}\}$ specifying whether tool responses are recomputed live, replayed from a record, or produced by a simulator. The stabilization policy operationalizes the determinism assumption: cached or simulated responses make repeated runs deterministic; live responses are used only when stochasticity is intentionally introduced (extension point E.2 below).*

Definition 2 (Action classification)**.** *Each tool in $\mathcal{T}$ is annotated with an access mode:*

- Read *(a^R): does not mutate tracked environment state (e.g., search, file read, GET requests).*
- Write *(a^W): mutates tracked environment state (e.g., file write, database update, POST/PUT/DELETE, shell commands with side effects).*

The classification is application- and environment-dependent: the same tool may be a read in one context and a write in another (a logging call is a side-effect-free read for the primary task but a write from the perspective of an audit system). The classification must be specified as part of the evaluation setup.

Definition 3 (Trajectory)**.** *Given an instruction $u \in \mathcal{U}$, agent policy π_θ, and environment $\mathcal{E}$, a* trajectory *is the logged event sequence*

$$\tau = \big\langle (s_0, a_1, o_1),\ (s_1, a_2, o_2),\ \ldots,\ (s_{n-1}, a_n, o_n),\ r \big\rangle,$$

where $a_i \in \mathcal{A}$ is an agent action, $o_i \in \mathcal{O}$ the resulting observation, and r the final response. Under the determinism assumption, the state subsequence $s_0, s_1, \ldots, s_n$ is uniquely determined by the action subsequence: $s_i = T(s_{i-1}, a_i)$ and $o_i = \Omega(s_{i-1}, a_i)$. We write $\text{actions}(\tau) = (a_1, \ldots, a_n)$ *for the action subsequence,* $\text{tokens}(\tau)$ *for the full sequence of agent-emitted tokens (tool-call arguments and final response), and $s_f(\tau) = s_n$ for the final state.*

Definition 4 (Group)**.** *A group $G = \{\tau_1, \ldots, \tau_k\}$ is the set of k trajectories produced by the same agent policy π_θ, the same task instruction u, and the same environment $\mathcal{E}$ reset to s_0 before each run. By construction, the only source of variation within G is the stochasticity of π_θ.*

Definition 5 (Correctness predicate)**.** *A* task correctness predicate $C(\tau) \in \{0, 1\}$ *returns 1 iff τ satisfies the task's correctness specification (tests pass, required state invariants hold, expected output produced). We require execution-based predicates throughout.*

Definition 6 (State projection)**.** *A* state projection $\rho : \mathcal{S} \to \mathcal{S}'$ *extracts the task-relevant slice of environment state. Examples include directory-tree hashes for file-system tasks, working-tree diffs for Git tasks, and normalized resource manifests for Kubernetes tasks. We compare $\rho(s_f)$ rather than full state, since full comparison is generally intractable and*

would include irrelevant artifacts (logs, timestamps, scratch files).

Definition 7 (Outcome projection)**.** *Any execution-based correctness predicate C inspects only a subset of state to decide pass/fail: the files, service states, configuration values, or exit codes the test script actually reads. The* outcome projection *$\sigma : \mathcal{S} \to \mathcal{S}''$ extracts exactly this subset, so that C factors through σ as $C = \tilde{C} \circ \sigma$ for some predicate $\tilde{C} : \mathcal{S}'' \to \{0, 1\}$. We require σ to be a coarsening of ρ: any two states equal under ρ are also equal under σ. Equivalently, ρ extracts at least as much information as σ, so whatever correctness depends on is part of what we consider task-relevant.*

Sourcing σ in practice. The outcome projection is implicit in any execution-based evaluator: it is the set of state features the test script actually reads to decide pass/fail. In our setup, σ is recovered by inspecting which files, service properties, and exit codes the per-task evaluator queries; we do not require task authors to supply σ separately from C.

Extension points (orthogonal axes). We name two relaxations of the main setting for future work. They change *what is being measured* (robustness to input variation, tolerance to environment noise) rather than behavioral consistency itself, and compose naturally with the levels defined below: one can ask, for example, whether an agent maintains effect-equivalent consistency under paraphrased instructions or intermittent tool failures.

- **E.1** *Robustness*: vary the instruction while preserving task intent (different surface forms, same semantic content).
- **E.2** *Fault tolerance*: allow stochastic transitions or observations (transient failures, timeouts, noisy outputs).

A.2 Consistency Levels

We define five consistency levels, ordered from strictest to most relaxed. Each level captures a distinct practical guarantee; extension points are indexed for future refinement. §A.3 states the assumptions under which the levels form a strictly ordered chain.

A.2.1 Level 1: Token-identical consistency. All externally observable tokens produced by the agent are identical across runs.

Definition 8 (Level 1)**.** *A group G satisfies* token-identical consistency *if*

$$\forall\, \tau_i, \tau_j \in G : \quad \text{tokens}(\tau_i) = \text{tokens}(\tau_j),$$

where $\text{tokens}(\tau)$ *denotes the full sequence of tokens emitted by the agent during trajectory τ, including tool-call arguments and the final response r.*

Sub-levels.

- **1a.** *Full-trace identical*: both reasoning (chain-of-thought) tokens and action tokens are identical. Testable only when the platform exposes internal traces (open-weight or research settings). In multi-agent systems, internal dialogue between sub-agents may also be observable from the orchestrator's perspective and falls under this sub-level.
- **1b.** *Action-token identical*: all externally visible tokens (tool calls, arguments, final response) are identical; internal reasoning tokens may differ.

When required. Regression testing of agent systems; reproducibility for scientific experiments; debugging infrastructure-level nondeterminism.

Achievability. Requires deterministic inference: batch-invariant kernels, fixed numerical precision, and greedy decoding. Achievable with current infrastructure for open-weight models.

A.2.2 Level 2: Action-equivalent consistency. The same tool calls are made in the same order with equivalent arguments, but the specific tokens used to express them may differ.

Token versus string identity. The distinction between Level 1 and Level 2 is not only about surrounding text. Due to tokenizer vocabulary, the same string can be produced by different token sequences (the string `"filename"` may be tokenized as `["file","name"]` or `["filen","ame"]`). Level 1 requires identical *token sequences*; Level 2 requires only that the decoded action strings, after canonicalization, are equivalent.

Definition 9 (Level 2)**.** *A group G satisfies* action-equivalent consistency *if*

$$\forall\, \tau_i, \tau_j \in G : \quad \text{norm}\big(\text{actions}(\tau_i)\big) = \text{norm}\big(\text{actions}(\tau_j)\big),$$

where norm *applies (i) string canonicalization of tool arguments (whitespace normalization, JSON key-sorting, type coercion), and (ii) tool-identity resolution under the assumption that the tool registry $\mathcal{T}$ is non-redundant: no two distinct tools produce the same state effect for the same inputs.*[1]

Extension point.

- **E.3** *Action-equivalence mapping.* An equivalence function $\sim_a$ over actions can be supplied to relax strict syntactic matching:

  $$\forall\, \tau_i, \tau_j \in G : \quad \text{actions}(\tau_i) \sim_a \text{actions}(\tau_j).$$

  This captures functionally equivalent but syntactically distinct actions: (a) semantically equivalent commands (e.g., `mv a b` vs. `cp a b && rm a`), and (b) API arguments treated identically by the downstream service (optional fields with default values, alternative

[1] Formally, for any two tools $t_1 \neq t_2$ in $\mathcal{T}$ and any argument x, $t_1(x) \neq t_2(x)$ with respect to their effect on tracked state. Relaxing this assumption is extension point E.3 below.

date formats, reordered list elements where order is ignored by the receiver).

Note on observations. Under deterministic dynamics, identical action sequences imply identical observation sequences, since $o_i = \Omega(s_{i-1}, a_i)$ and the state subsequence is uniquely determined by the actions. The action-only versus action-plus-observation distinction becomes meaningful only when E.2 is invoked.

When required. Workflow auditing and compliance scenarios requiring deterministic audit trails; cost estimation (identical API-call patterns imply identical cost).

A.2.3 Level 3: Effect-equivalent consistency. All state-modifying (write) actions are identical and in the same order; read actions may vary freely. Which actions are reads versus writes is application-dependent (see Action Classification in §A.1); the consistency guarantee at this level is relative to the chosen classification.

Definition 10 (Level 3)**.** *A group G satisfies* effect-equivalent consistency *if*

$$\forall\, \tau_i, \tau_j \in G: \quad \text{filter}_W\big(\text{actions}(\tau_i)\big) = \text{filter}_W\big(\text{actions}(\tau_j)\big),$$

where filter_W *retains only write actions* a^W *(with normalized arguments) and preserves their relative order.*

When required. System administration where the sequence of state mutations matters but diagnostic (read) strategies may vary; software engineering where edit sequences must be predictable but information-gathering paths need not be.

A.2.4 Level 4: State-equivalent consistency. All trajectories produce the same final environment state under projection; the action sequences (including writes) may differ.

Definition 11 (Level 4)**.** *A group G satisfies* state-equivalent consistency *if*

$$\forall\, \tau_i, \tau_j \in G: \quad \rho\big(s_f(\tau_i)\big) = \rho\big(s_f(\tau_j)\big).$$

This is the projection-restricted analog of state equivalence under bisimulation [4]: trajectories producing ρ-identical final states are behaviorally indistinguishable from any consumer that observes only $\rho(s)$.

Extension points. The gap between write-order equivalence (Level 3) and state equivalence can be narrowed by exploiting structural properties of write operations:

- **E.4** *Commutativity*: if two writes act on independent resources (e.g., disjoint files), their order may be permuted without affecting final state.
- **E.5** *Idempotency*: if a write w satisfies $w \circ w = w$, duplicate occurrences can be collapsed.
- **E.6** *Overwrite absorption*: if a later write to the same resource fully supersedes an earlier one, the earlier write can be elided from the comparison.
- **E.7** *Structural decomposition*: the state may be partitioned into independent substructures (directories in a file system, namespaces in Kubernetes, tenants in a multi-tenant database). State equivalence can then be evaluated per substructure: given $\rho = (\rho_1, \ldots, \rho_m)$, per-partition equivalence holds if

$$\forall\, j \in [m],\ \forall\, \tau_i, \tau_k \in G: \quad \rho_j(s_f(\tau_i)) = \rho_j(s_f(\tau_k)).$$

Formalizing these rewrite rules and integrating them into the checker is left to future work.

When required. Operator-facing scenarios where what matters is the system's final state, not the path taken: configuration management, database migrations, and any deployment whose contract is expressed in terms of the post-condition rather than the procedure.

A.2.5 Level 5: Outcome-equivalent consistency. All trajectories produce the same outcome under the coarser projection σ (the subset of state on which the correctness predicate depends); fuller final states under ρ, action paths, and intermediate observations may all differ.

Definition 12 (Level 5)**.** *A group G satisfies* outcome-equivalent consistency *if*

$$\forall\, \tau_i, \tau_j \in G: \quad \sigma\big(s_f(\tau_i)\big) = \sigma\big(s_f(\tau_j)\big).$$

Relation to correctness. Because C factors as $C = \tilde{C} \circ \sigma$, two trajectories satisfying Level 5 share the same correctness verdict: $C(\tau_i) = C(\tau_j) = \tilde{C}(\sigma(s_f(\tau_i)))$. The shared verdict need not be "pass". The relationship to τ-bench's pass$\hat{}k$ metric [17] is therefore: pass$\hat{}k = 1.0$ on a group G holds iff G satisfies Level 5 *and* the common σ value satisfies $\tilde{C}$.

Extension point.

- **E.8** *Parameterized acceptance.* The binary post-projection predicate $\tilde{C} : \mathcal{S}'' \to \{0, 1\}$ can be generalized to a parameterized acceptance function $\tilde{C}_\phi$ that encodes domain-specific notions beyond pass/fail. Examples: semantic equivalence of generated code (via diff or AST tools); database states matching under a normalized schema; satisfying domain-specific invariants. The choice of ϕ adjusts what "acceptable outcome" means downstream of Level 5, without altering the projection σ used for the level itself.

When required. Deployment scenarios where only the correctness-checked subset of state determines acceptance: software-engineering benchmarks where structurally different patches passing the same tests are interchangeable; database migrations where only the final schema and data values are checked; any contract where "it produces the right answer" is the binding requirement and other state differences are immaterial.

A.3 Strict Ordering of the Hierarchy

We now show that, under explicit assumptions, the five levels form a chain in which each adjacent implication is strict (the converse fails).

Assumptions.

- **A1** *Deterministic dynamics*: T and Ω are functions, not distributions.
- **A2** *Common reset*: every $\tau \in G$ starts from the same s_0.
- **A3** *Faithful read/write classifier*: any action labeled a^R leaves ρ invariant, i.e., $\rho(T(s, a^R)) = \rho(s)$ for all $s \in \mathcal{S}$.
- **A4** *Coarsening*: σ is a coarsening of ρ, i.e., there exists $\pi : \mathcal{S}' \to \mathcal{S}''$ with $\sigma = \pi \circ \rho$.
- **A5** *Deterministic action extraction*: actions(τ) is a deterministic function of tokens(τ) (the parser that turns emitted tokens into structured tool calls is itself a function).
- **A6** *Non-redundant tool registry* (E.3 not invoked): distinct tools have distinct effects on tracked state, and norm is a deterministic function of action sequences.

Theorem (Strict ordering of consistency levels). *Under A1–A6, for any group G and any i, j with $1 \le i < j \le 5$, if G satisfies Level i then G satisfies Level j. Moreover, for every adjacent pair $(i, i{+}1)$ the converse fails: there exists a group satisfying Level $i{+}1$ but not Level i.*

Proof. We prove the four forward implications, then exhibit a witness against each converse.

(1) Level 1 implies Level 2. Assume $\text{tokens}(\tau_i) = \text{tokens}(\tau_j)$. By A5, $\text{actions}(\tau_i) = \text{actions}(\tau_j)$. By A6, norm is a function of action sequences, so equal inputs produce equal outputs: $\text{norm}(\text{actions}(\tau_i)) = \text{norm}(\text{actions}(\tau_j))$, which is Level 2.

(2) Level 2 implies Level 3. Assume $\text{norm}(\text{actions}(\tau_i)) = \text{norm}(\text{actions}(\tau_j))$. The map filter_W retains only write actions (with normalized arguments) and preserves their relative order; the read/write annotation of each action is fixed by the registry, and argument normalization is what norm already applies. Hence filter_W factors through norm: $\text{filter}_W(\text{actions}(\tau)) = \text{filter}'_W(\text{norm}(\text{actions}(\tau)))$ for some function filter'_W. Equal inputs yield equal outputs, giving $\text{filter}_W(\text{actions}(\tau_i)) = \text{filter}_W(\text{actions}(\tau_j))$, which is Level 3.

(3) Level 3 implies Level 4. We use the following lemma.

Lemma 1 (read-purity for ρ). *Under A1 and A3, for any sequence of actions $a_1, \ldots, a_n$ executed from initial state s_0 with resulting final state s_n, $\rho(s_n)$ depends only on the subsequence of write actions and their relative order.*

Proof of Lemma 1. By induction on n. *Base case* ($n = 0$): trivial; $s_n = s_0$ and the empty write subsequence yields $\rho(s_0)$. *Inductive step*: assume the claim holds for length-$(n-1)$ sequences. Given a length-n sequence with intermediate state s_{n-1} and final action a_n:

- If $a_n = a_n^R$ is a read, then by A3, $\rho(T(s_{n-1}, a_n^R)) = \rho(s_{n-1})$, so $\rho(s_n) = \rho(s_{n-1})$. The length-$(n-1)$ prefix has the same write subsequence, so by inductive hypothesis $\rho(s_{n-1})$ is determined by the write subsequence of the prefix, which equals the full sequence's write subsequence.
- If $a_n = a_n^W$ is a write, then by A1, $s_n = T(s_{n-1}, a_n^W)$ and hence $\rho(s_n)$ is a function of s_{n-1} and a_n^W. By inductive hypothesis, the inputs to that function are determined by the prefix's write subsequence, and appending a_n^W extends the write subsequence by one. Therefore $\rho(s_n)$ is determined by the full write subsequence and its order.

Now suppose Level 3 holds: $\text{filter}_W(\text{actions}(\tau_i)) = \text{filter}_W(\text{actions}(\tau_j))$. By A2 both runs start from s_0. By Lemma 1, $\rho(s_f(\tau_i))$ and $\rho(s_f(\tau_j))$ are each determined by their respective (equal) write subsequences executed from the same s_0. Therefore $\rho(s_f(\tau_i)) = \rho(s_f(\tau_j))$, which is Level 4.

(4) Level 4 implies Level 5. By A4, $\sigma = \pi \circ \rho$ for some π. If $\rho(s_f(\tau_i)) = \rho(s_f(\tau_j))$ then

$$\sigma(s_f(\tau_i)) = \pi(\rho(s_f(\tau_i))) = \pi(\rho(s_f(\tau_j))) = \sigma(s_f(\tau_j)),$$

which is Level 5.

Strictness witnesses. We now show each converse fails by exhibiting a 2-trajectory group $G = \{\tau_a, \tau_b\}$ that satisfies the weaker level but not the stronger.

Level 2 does not imply Level 1. The fundamental gap is that a single tool call, even when byte-identical in its serialized form, can be emitted by multiple distinct token sequences. The tokenizer vocabulary contains tokens at varying granularities, so the same byte string `read_file("a.txt")` may be emitted as the four-token sequence

$$\big[\texttt{read\_file},\ \texttt{(},\ \texttt{"a.txt"},\ \texttt{)}\big]$$

or, when a coarser merge exists in the vocabulary, as a single token whose decoded value is the entire expression,

$$\big[\texttt{read\_file("a.txt")}\big].$$

The model generates token IDs directly and under sampling may select either path. The decoded string, the parsed action, and norm(actions(τ)) all coincide across the two runs (Level 2 holds), yet the emitted token sequences differ (Level 1 fails). A second source is formatting variation that norm absorbs (whitespace, JSON key ordering, number formats such as `1` versus `1.0`, equivalent string escapes): here the byte sequences themselves differ across runs but collapse to the same action after norm. Either mechanism alone witnesses Level 2 $\not\Rightarrow$ Level 1.

Level 3 does not imply Level 2. Let τ_a first `ls` and then `cat README.md`, ending with `write_file("out.txt", v)`. Let τ_b first `cat README.md`, then `ls`, then perform the same `write_file("out.txt", v)`. Both have the same write subsequence [`write_file("out.txt", v)`] (Level 3 holds), but their action sequences differ in the read prefix (Level 2 fails).

Level 4 does not imply Level 3. Let τ_a execute writes x:=1 then x:=2; let τ_b execute the single write x:=2. Both produce the same final state $\rho(s_f) = \{x = 2\}$ (Level 4 holds) but distinct write subsequences (Level 3 fails).

Level 5 does not imply Level 4. Let ρ track the pair $(x, \log)$ and σ project to x only (a valid coarsening since the test script reads only x). Let τ_a produce $\rho(s_f)$ = (x=1, log="A") and τ_b produce $\rho(s_f)$ = (x=1, log="B"). Then σ-projections agree (Level 5 holds) but ρ-projections differ (Level 4 fails). This pattern arises whenever task-relevant state captured by ρ (timestamps, log contents, scratch files retained for downstream analysis) is intentionally omitted from σ.

This completes the proof.

B Worked Example: What Each Level Extracts

To make the consistency hierarchy concrete, we walk through one trajectory from our experiments and show what each level retains for the equivalence comparison. The trajectory is one run of task `cert-rotation` by Qwen 3.5-4B. The task is to rotate TLS certificates for two services running in a container: an Nginx server on port 443 (configured in `/etc/nginx/conf.d/app.conf`) and a Python API server on port 8443 (configured in `/app/api_server.py`). The new certificate and key live at `/etc/ssl/certs/server-new.crt` and `/etc/ssl/private/server-new.key`; both services must end up running and serving the new certificate.

B.1 Initial and Per-Turn States

Before the agent acts, and after every turn, the projector records the states of the container. Throughout this section, **red** marks the three `critical_keys` retained at Level 5. The initial state is:

```
{
    nginx_running:              true,
    api_running:                true,
    nginx_cert_cn:              "expiring.example.com",
    api_cert_cn:                "expiring.example.com",
    nginx_conf_cert:            "/etc/ssl/certs/server.crt",
    nginx_conf_key:             "/etc/ssl/private/server.key",
    api_conf_cert:              "/etc/ssl/certs/server.crt",
    api_conf_key:               "/etc/ssl/private/server.key",
    old_cert_is_symlink:        false,
    old_cert_replaced:          false,
    old_key_is_symlink:         false,
    old_key_replaced:           false,
    cert_key_matched:           true,
}
```

B.2 Trajectory

Turn 1. *State unchanged* (read-only).

```
$ cat /etc/nginx/conf.d/app.conf
$ head -50 /app/api_server.py
```

Turn 2. *State change:* `nginx_conf_cert/key` now point at the new files.

```
$ sed -i 's|/etc/ssl/certs/server.crt|/etc/ssl/certs/server-new.crt|g' /etc/nginx/conf.d/app.conf
$ sed -i 's|/etc/ssl/private/server.key|/etc/ssl/private/server-new.key|g' /etc/nginx/conf.d/app.conf
```

Turn 3. *State unchanged* (read-only).

```
$ sed -i 's|"CERT_FILE = "/etc/ssl/certs/server.crt"|...|' /app/api_server.py
$ sed -i 's|"KEY_FILE = "/etc/ssl/private/server.key"|...|' /app/api_server.py
```

Turn 4. *State unchanged* (read-only).

```
$ cat /etc/nginx/conf.d/app.conf
$ head -20 /app/api_server.py
$ pgrep -a nginx
$ pgrep -a python
```

Turn 5. *State change:* `api_conf_cert/key` now point at the new files.

```
$ sed -i 's|CERT_FILE = "/etc/ssl/certs/server.crt"|...new.crt"|' /app/api_server.py
$ sed -i 's|KEY_FILE  = "/etc/ssl/private/server.key"|...new.key"|' /app/api_server.py
$ head -15 /app/api_server.py
```

Turn 6. **Critical state change:** the API-server python process is killed, and nginx is reloaded against the new cert config.

```
$ kill 19
$ nginx -s reload
```

⇓

Turn 6 (zoomed in)

State before turn 6:

```
{
    nginx_running:                  true,
    api_running:                    true,
    nginx_cert_cn:                  "expiring.example.com",
    api_cert_cn:                    "expiring.example.com",
    nginx_conf_cert:                "/etc/ssl/certs/server-new.crt",
    nginx_conf_key:                 "/etc/ssl/private/server-new.key",
    api_conf_cert:                  "/etc/ssl/certs/server-new.crt",
    api_conf_key:                   "/etc/ssl/private/server-new.key",
    old_cert_is_symlink:            false,
    old_cert_replaced:              false,
    old_key_is_symlink:             false,
    old_key_replaced:               false,
    cert_key_matched:               true,
}
```

Output token IDs:

```
[6820, 2107, 6343, 3425, 599, 978, 5860, 13, 4543, 353, 1144, 310...]
```

Output message:

```
{
  "reasoning": "Now both configuration files have been updated.
               Now I need to restart both services:\n
               1. nginx - reload it to apply the new cert config\n
               2. The API server - kill the current process
                  and start it again",
  "content":   "Both config files are updated. Now I need to
               restart both services with the new certificates:",
  "tool_calls": [
    {"name": "bash", "arguments": {"command": "kill 19"}},
    {"name": "bash", "arguments": {"command": "nginx -s reload"}}
  ]
}
```

State after turn 6 (keys that changed in this turn marked •):

```
{
      nginx_running:                true,
      api_running:                  true,
    • nginx_cert_cn:                "expiring.example.com" → "server.example.com",
    • api_cert_cn:                  "expiring.example.com" → null,
      nginx_conf_cert:              "/etc/ssl/certs/server-new.crt",
      nginx_conf_key:               "/etc/ssl/private/server-new.key",
      api_conf_cert:                "/etc/ssl/certs/server-new.crt",
      api_conf_key:                 "/etc/ssl/private/server-new.key",
      old_cert_is_symlink:          false,
      old_cert_replaced:            false,
      old_key_is_symlink:           false,
      old_key_replaced:             false,
      cert_key_matched:             true,
}
```

Turn 7. *State change (critical):* **api_cert_cn** becomes "server.example.com".

```
$ python3 /app/api_server.py &
```

Turn 8. *State unchanged* (verification probes).

```
$ pgrep -a nginx && pgrep -a python
$ echo | openssl s_client -connect localhost:443  2>/dev/null | openssl x509 -noout -subject
$ echo | openssl s_client -connect localhost:8443 2>/dev/null | openssl x509 -noout -subject
```

Turn 9. *State unchanged* (issuer probes).

```
$ echo | openssl s_client -connect localhost:443  2>/dev/null | openssl x509 -noout -issuer
$ echo | openssl s_client -connect localhost:8443 2>/dev/null | openssl x509 -noout -issuer
```

Turn 10. *State unchanged.*

```
$ ls -la /etc/ssl/certs/server-new.crt /etc/ssl/private/server-new.key
```

Turn 11. Calls `finish()` to end the trajectory.

B.3 What Each Level Extracts

Level 1 (Token-identical). The full sequence of emitted token IDs across all turns, before any decoding (the per-turn token IDs concatenated end to end). Two runs match only if every emitted token ID is identical, position by position.

Level 2 (Action-equivalent). The ordered sequence of tool calls with their arguments (reasoning text and tool responses discarded). For this trajectory:

```
[bash("cat /etc/nginx/conf.d/app.conf"), bash("head -50 /app/api_server.py"), bash("sed -i 's|.../server.crt|.../server-new.crt|g'
/etc/nginx/conf.d/app.conf"), . . . , finish()]
```

Level 3 (Effect-equivalent). Since read-only commands do not change any state variable, Level 3 collapses to the sequence of post-turn projected states with adjacent duplicates removed—one entry per turn that actually moved the state. For this trajectory (states after turns 1, 3, 4, 8–10 are omitted during comparison because they were read operations):

Initial state: `{nginx_running:true, api_running:true, nginx_cert_cn:"expiring.example.com"...}`
After turn 2: `{nginx_running:true, api_running:true, nginx_cert_cn:"expiring.example.com"...}`
After turn 5: `{nginx_running:true, api_running:true, nginx_cert_cn:"expiring.example.com"...}`
After turn 6: `{nginx_running:true, api_running:true, nginx_cert_cn:"server.example.com"...}`
After turn 7: `{nginx_running:true, api_running:true, nginx_cert_cn:"server.example.com"...}`

Level 4 (State-equivalent). The final container state under the projection π defined in `cert_rotation.py`. For this trajectory π returns:

```
{
  nginx_running:              true,
  api_running:                true,
  nginx_cert_cn:              "server.example.com",
  api_cert_cn:                "server.example.com",
  nginx_conf_cert:            ".../server-new.crt",
  nginx_conf_key:             ".../server-new.key",
  api_conf_cert:              ".../server-new.crt",
  api_conf_key:               ".../server-new.key",
  old_cert_is_symlink:        false,
  old_cert_replaced:          false,
  old_key_is_symlink:         false,
  old_key_replaced:           false,
  cert_key_matched:           true,
}
```

A different run that solves the task by overwriting `server.crt` in place with `cp -f server-new.crt server.crt` reaches a different projected state at this level (`old_cert_replaced: true`, `nginx_conf_cert: unchanged`) and so would *not* cluster with this run, even though both end up serving the new certificate.

Level 5 (Outcome-equivalent). Only the subset of π tagged as `critical_keys` in the projector—for `cert-rotation` these are **`api_running`**, **`nginx_cert_cn`**, and **`api_cert_cn`**:

```
{
  api_running:                true,
  nginx_cert_cn:              "server.example.com",
  api_cert_cn:                "server.example.com",
}
```

At this level the sed-based run above and the `cp -f`-based run cluster together: both leave the api running and both serve `server.example.com` on ports 443 and 8443.

B.4 Summary

Moving from Level 1 to Level 5, the equivalence relation gets coarser, so the corresponding consistency notion gets weaker—two runs that match at a higher level always match at every lower level, but the converse does not hold. Concretely, this hierarchy progressively discards information: Level 1 keeps every token; Level 2 drops reasoning and tool responses; Level 3 drops

reads and diagnostics; Level 4 drops the action sequence and keeps only its effect on the projected state; Level 5 drops all non-critical state keys. Two runs that differ only in diagnostic commands cluster together at Level 3 and below; two runs that reach the same projected state via different action sequences cluster together at Level 4 and below; two runs that satisfy the same outcome predicate cluster together at Level 5 even when their full projected states differ.

C Detailed Experiment Settings

C.1 Model Versions

Table 2. Model identifiers and snapshot versions used in all experiments.

Model	Deployment Mode	ID / Snapshot
GPT-5.4	OpenRouter API	`openai/gpt-5.4-20260305`
GPT-5.4 Mini	OpenRouter API	`openai/gpt-5.4-mini-20260317`
GPT-5.4 Nano	OpenRouter API	`openai/gpt-5.4-nano-20260317`
Gemini 3.1 Pro	OpenRouter API	`google/gemini-3.1-pro-preview-20260219`
Gemini 3.1 Flash Lite	OpenRouter API	`google/gemini-3.1-flash-lite-preview-20260303`
Claude Opus 4.5	OpenRouter API	`anthropic/claude-4.5-opus-20251124`
Claude Sonnet 4.5	OpenRouter API	`anthropic/claude-4.5-sonnet-20250929`
Claude Haiku 4.5	OpenRouter API	`anthropic/claude-4.5-haiku-20251001`
DeepSeek V3.2	OpenRouter API	`deepseek/deepseek-v3.2-20251201`
GLM 5	OpenRouter API	`z-ai/glm-5-20260211`
Kimi K2.5	OpenRouter API	`moonshotai/kimi-k2.5-0127`
MiniMax M2.5	OpenRouter API	`minimax/minimax-m2.5-20260211`
Qwen 3.5 Plus	OpenRouter API	`qwen/qwen3.5-plus-20260216`
Qwen 3.5 35B-A3B	Local (vLLM)	`Qwen/Qwen3.5-35B-A3B@59d61f3`
Qwen 3.5 27B	Local (vLLM)	`Qwen/Qwen3.5-27B@fc05dae`
Qwen 3.5 4B	Local (vLLM)	`Qwen/Qwen3.5-4B@851bf6e`
Gemma 4 31B	Local (vLLM)	`google/gemma-4-31B-it@439edf5`

C.2 OpenRouter Parameters

Table 3. OpenRouter API settings used for all remotely accessed models.

Setting	Value
`temperature`	1.0
`top_p`	1.0
`top_k`	0 (consider all tokens)
`tool_choice`	`auto`
`reasoning.enabled`	`true`
`reasoning.effort`	`medium` (if supported)

For all other optional parameters (see OpenRouter API reference), we use default values without explicit specification. For the temperature sweep experiment, reasoning is disabled via the request parameter.

C.3 Local Deployment Settings

All local models were served using vLLM. To produce the per-token trajectories used by our projectors, vLLM must be served with `--return-tokens-as-token-ids`; otherwise token-id sequences are not returned in the response. The serve commands for each model family are listed below.

Locally hosted Gemma 4 31B..

```
vllm serve google/gemma-4-31B-it \
  --port 8000 \
  --max-model-len 15000 \
  --tensor-parallel-size 1 \
  --enable-auto-tool-choice \
  --tool-call-parser gemma4 \
  --reasoning-parser gemma4 \
  --chat-template ./tool_chat_template_gemma4.jinja
```

Locally hosted Qwen 3.5.

```
# <CONTEXT_LEN>: 64000 for 27B and 35B-A3B; flag omitted (uncapped) for 4B
vllm serve Qwen/MODEL_NAME \
  --port 8000 \
  --max-model-len <CONTEXT_LEN> \
  --tensor-parallel-size 1 \
  --enable-auto-tool-choice \
  --tool-call-parser qwen3_coder \
  --reasoning-parser qwen3 \
  --language-model-only
```

C.4 Compute Resources

Local inference. The temperature sweep (§4) was run with vLLM `v0.19.1` on 4× NVIDIA H100 PCIe (80 GB), CUDA `13.0` (driver `580.126.20`).

Remote inference. All other experiments accessed models via the OpenRouter API. Total API spend across all reported remote runs was approximately $190.

Container host. Tasks ran in Docker `28.4.0` on an Apple M4 Mac with 10 CPU cores and 16 GB RAM.

Wall-clock. Summed across all reported attempts, the total runtime is approximately 165 hours: 35 hours for local (vLLM) runs and 130 hours for cloud (OpenRouter) runs. These numbers are the sum of per-attempt durations. In practice we ran multiple attempts concurrently, with the concurrency level chosen per setting based on provider rate limits and host resources, so the elapsed wall-clock time was substantially shorter.

D Licenses for Existing Assets

We use the following third-party assets. All are used in accordance with their licenses and terms of service. Snapshot identifiers and links for each model are listed in Appendix C.

Benchmarks and tooling.

- **Terminal-Bench** [13]: Apache-2.0. A subset of the tasks in our benchmark are adapted from Terminal-Bench scenarios, and our container interaction code and agent prompt are adapted from the Terminal-Bench agent harness; adapted versions are released under the same license.
- **vLLM**: Apache-2.0. Used to serve open-weight models locally.
- **Docker Engine**: Apache-2.0. Used to create isolated container environments for agent interaction.
- **Docker Desktop**: governed by the Docker Subscription Service Agreement; our use falls under the agreement's free-tier eligibility.

Open-weight models (local vLLM deployment).

- **Qwen 3.5** (35B-A3B, 27B, 4B): Apache-2.0, per the respective Hugging Face model cards. Qwen 3.5 Plus is accessed only via API and is listed below.
- **Gemma 4 31B**: Gemma Terms of Use, per the Hugging Face model card.

Closed API models. GPT-5.4 (and Mini, Nano), Claude Opus / Sonnet / Haiku 4.5, Gemini 3.1 Pro / Flash Lite, DeepSeek V3.2, GLM 5, Kimi K2.5, MiniMax M2.5, and Qwen 3.5 Plus were accessed via the OpenRouter API under the respective providers' terms of service.

E Metrics

E.1 Trajectory Metrics

E.2 Formal Definition of CONSISTENCY^N

Definition. Let $p_1, \ldots, p_m$ be the true probabilities of each equivalence cluster under a fixed (model, task, parameter) distribution. CONSISTENCY^N is the probability that N independently drawn trajectories all fall into the same cluster:

$$\text{CONSISTENCY}^N = \sum_{i=1}^{m} p_i^N. \tag{1}$$

CONSISTENCY^N $= 1$ when all trajectories are identical; CONSISTENCY^N $\to 0$ as behavior becomes maximally dispersed. Given m clusters with observed sizes $|C_1|, \ldots, |C_m|$ across k trials, we estimate CONSISTENCY^N with two variants. The *with-replacement plug-in estimator* substitutes $\hat{p}_i = |C_i|/k$:

$$\hat{C}^N_{\text{plugin}} = \sum_{i=1}^{m} \left(\frac{|C_i|}{k} \right)^N. \tag{2}$$

The *without-replacement unbiased estimator* uses falling factorials to correct for finite-sample bias:

$$\hat{C}^N_{\text{unbiased}} = \frac{\sum_{i=1}^{m} \binom{|C_i|}{N}}{\binom{k}{N}} = \frac{\sum_{i=1}^{m} |C_i|^{\underline{N}}}{k^{\underline{N}}}, \tag{3}$$

where $k^{\underline{N}} = k(k-1)\cdots(k-N+1)$.

We use the without-replacement unbiased estimator (3) in the main text and the appendix tables. A consequence is that any singleton cluster ($|C_i| = 1$) contributes 0 to $\hat{C}^N_{\text{unbiased}}$ for $N \geq 2$; the plug-in instead assigns it the small but nonzero mass $1/k^N$. As $k \to \infty$ both estimators converge to the $\sum_i p_i^N$.

Detection probability. With k trials, the probability of observing a behavior with true rate p at least once is:

$$P(\text{detect}) = 1 - (1-p)^k. \tag{4}$$

For $k = 25$, behaviors occurring $\geq 10\%$ of the time are detected with $> 92\%$ probability, while behaviors with $p < 5\%$ are detected only 72% of the time. Achieving 95% detection at $p = 1\%$ would require $k \geq 300$.

Table 4. Trajectory-level metrics collected for each trial.

Metric	Explanation
initial_prompt_tokens	Prompt token count of the first API call.
total_completion_tokens	Sum of completion tokens across all API calls.
total_reasoning_tokens	Sum of reasoning tokens across all API calls.
total_non_reasoning_tokens	total_completion_tokens − total_reasoning_tokens.
final_context_length	Total token count (total_tokens) of the last API call.
tool_result_tokens	final_context_length − total_completion_tokens − initial_prompt_tokens.
reasoning_ratio	total_reasoning_tokens / total_completion_tokens.
trajectory_length	Number of API calls (rounds) in the trial.
tokens_per_round	total_completion_tokens / trajectory_length.
write_rounds	Number of rounds where the environment state changed.
read_rounds	trajectory_length − write_rounds.
rw_ratio	read_rounds / write_rounds.
first_write_round	1-indexed round number of the first state change.
first_write_ratio	first_write_round / trajectory_length.
write_burst_ratio	(last_write_round − first_write_round) / trajectory_length.
reward	Binary correctness score (0 or 1) from the evaluator.

F Detailed Results

Bootstrap confidence intervals. All CIs are computed via bootstrap with $B = 1{,}000$ resamples; we report the 2.5th and 97.5th percentiles as the 95% CI.

F.1 Per-(Model, Task) Heatmaps

Figures 4–7 report the per-(model, task) consistency CONSISTENCY^2 at Levels 2–5 with bootstrap 95% CIs ($B = 1{,}000$ resamples).

F.2 Deterministic Inference

Setup. To isolate the floor on CONSISTENCY$^2_{lv1}$ and CONSISTENCY$^2_{lv2}$, we re-ran a subset of the benchmark with deterministic inference enabled, which fixes bit-level model outputs for a given input regardless of batch composition or kernel scheduling. We use vLLM's batch-invariant kernels via `VLLM_BATCH_INVARIANT=1` with a fixed sampling seed and greedy decoding (temperature 0); the batch-invariant operators replace the standard matmul, attention, and RMSNorm kernels (and for MoE models, the routing and expert-dispatch path) with versions whose reduction order does not depend on batch size, eliminating the floating-point non-associativity that otherwise lets vLLM outputs vary with batch composition [6, 16]. The MoE coverage has been validated upstream on `Qwen3-30B-A3B` and `Qwen3-Next-80B-A3B`; our single-GPU configuration (no expert parallelism) further avoids the all-to-all non-determinism that distributed MoE serving could introduce. Equivalent functionality is available in SGLang via `–enable-deterministic-inference` [15] and in Hugging Face Transformers via `torch.use_deterministic_algorithms(True)` together with `CUBLAS_WORKSPACE_CONFIG=:16:8`. Three Qwen-3.5 models (4B dense, 27B dense, 35B-A3B MoE) were served via vLLM with the same serve commands as in §4, and we ran 25 attempts on each of four tasks: `cert-rotation`, `firewall-lockout-recovery`, `hidden-process-port-conflict`, `zero-downtime-deploy`.

An empirical baseline for the contribution of model non-determinism. Because deterministic inference fixes the model's output as a function of input, all variation in CONSISTENCY^2 in Figure 8 is attributable to environmental noise. The deterministic numbers therefore serve as a baseline against which the main-paper results (which include both model and environmental noise) can be compared: the gap between them at each level is an upper bound on the model-stochasticity contribution to inconsistency at that level. In our setup, CONSISTENCY$^2_{lv4}$ and CONSISTENCY$^2_{lv5}$ reach 1.0 on most cells, indicating that environmental noise alone rarely changes the final environment state or task outcome; CONSISTENCY$^2_{lv3}$ stays high but is occasionally pushed below 1 when run-varying tool output flips a downstream write argument; CONSISTENCY$^2_{lv1}$ and CONSISTENCY$^2_{lv2}$ are dominated by the environmental floor (see below). The pass-rate panel is included for reference: pass rate is unchanged from the main-paper setting on most cells, since the deterministic configuration affects *which* trajectory the agent produces, not whether the produced trajectory satisfies the task's correctness predicate.

Why CONSISTENCY$^2_{lv1}$ and CONSISTENCY$^2_{lv2}$ stay below 1. The residual gap is environmental, not stack-side. Comparing two attempts on `cert-rotation` with Qwen-3.5-4B turn by turn, the model's reasoning text and tool calls match exactly for the first eight turns, confirming that the inference stack itself is bit-stable. The first divergence does not originate in the model. It appears in a tool result: `ps aux` reports different RSS values for the same process across runs (19812 vs. 19996 bytes), and from that point the model's input differs, so greedy decoding produces different (still deterministic) tokens on every subsequent turn. Tool outputs in this benchmark commonly contain such run-varying numeric content: `ls -la` and `stat` expose mtime, `journalctl` and `dmesg` expose wall-clock timestamps, `df` reports byte counts that drift with kernel-internal allocations, `git log` reports commit hashes that incorporate the committer timestamp, and any command printing PIDs depends on container start-up ordering. A single such drift compounds across a multi-turn trajectory, collapsing CONSISTENCY$^2_{lv1}$; CONSISTENCY$^2_{lv2}$ collapses for the same reason once the model's downstream action choices diverge.

Existence proof. `firewall-lockout-recovery` reaches CONSISTENCY$^2_{lv1}$=…=CONSISTENCY$^2_{lv5}$=1.0 across all three models (Figure 8). The trajectory is short (six to seven turns) and the commands the agent issues do not expose run-varying state, demonstrating that deterministic inference combined with a clean tool environment is sufficient for full reproducibility, including at the token level. Achieving CONSISTENCY$^2_{lv1}$=1.0 on the other tasks would require either canonicalizing tool outputs to strip such fields before they enter the model's context, or restricting evaluation to tasks engineered not to expose them; neither is a property of typical system-administration workloads.

F.3 Temperature Sweep

Table 5 reports per-temperature CONSISTENCY^2 at Levels 3–5 for the two locally hosted models (Qwen 3.5-27B and Gemma 4-31B), averaged across all tasks. Values are CI midpoints with bootstrap 95% half-widths ($B = 1{,}000$).

F.4 Reasoning Sweep

Table 6 reports per-task CONSISTENCY^2 at Levels 3–5 for GPT-5.4 Nano at $T = 1.0$ across reasoning efforts. Values are CI midpoints with bootstrap 95% half-widths ($B = 1{,}000$); the bottom *average* row is the cross-task mean.

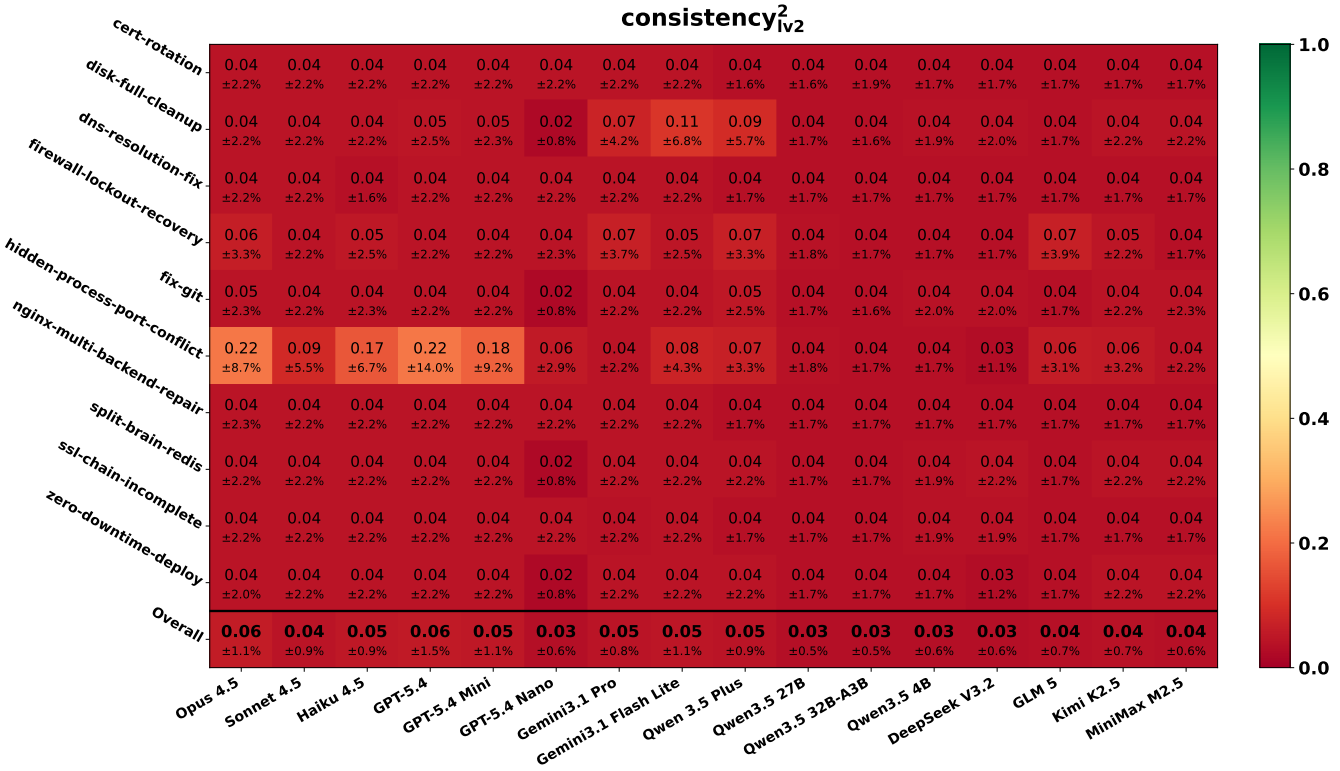

Figure 4. CONSISTENCY$^2_{lv2}$

Table 5. Temperature sweep: CONSISTENCY^2 at Levels 3–5 averaged across tasks, with bootstrap 95% CI half-widths.

	$T = 0.0$	$T = 0.1$	$T = 0.5$	$T = 1.0$
Qwen 3.5-27B				
CONSISTENCY$^2_{lv3}$	0.875 ± 0.030	0.694 ± 0.053	0.390 ± 0.040	0.315 ± 0.039
CONSISTENCY$^2_{lv4}$	0.973 ± 0.021	0.853 ± 0.043	0.663 ± 0.033	0.627 ± 0.037
CONSISTENCY$^2_{lv5}$	0.981 ± 0.019	0.953 ± 0.027	0.866 ± 0.033	0.867 ± 0.034
Gemma 4-31B				
CONSISTENCY$^2_{lv3}$	0.882 ± 0.034	0.752 ± 0.045	0.596 ± 0.042	0.487 ± 0.034
CONSISTENCY$^2_{lv4}$	0.953 ± 0.014	0.837 ± 0.031	0.775 ± 0.033	0.722 ± 0.029
CONSISTENCY$^2_{lv5}$	0.960 ± 0.012	0.929 ± 0.022	0.926 ± 0.020	0.905 ± 0.018

Curve-shape analysis. The effect of reasoning effort on consistency is governed by two competing mechanisms: (1) **convergence**—deeper reasoning helps identify the best approach and avoid omissions, concentrating trajectories into fewer clusters; (2) **exploration**—deeper reasoning discovers alternative strategies or optional improvements, diversifying trajectories across runs. Unlike temperature—where convergence dominates and the trend is mostly monotonic—the balance between these mechanisms for reasoning effort is task-dependent, producing qualitatively different patterns across tasks:

- *Monotonically increasing:* On `zero-downtime-deploy`, CONSISTENCY$^2_{lv4}$ rises from 0.65 at `low` to 1.00 at `xhigh`. Convergence dominates: deeper reasoning helps the model reliably identify the fixed procedure.
- *Monotonically decreasing:* On `split-brain-redis`, CONSISTENCY$^2_{lv4}$ drops from 0.64 at `low` to 0.09 at `xhigh`. Exploration dominates: deeper reasoning leads the model to pursue diverse repair strategies.
- *U-shape (high at extremes):* On `hidden-process-port-conflict`, consistency is high at `low` (0.58) and `xhigh` (0.63) but dips at `medium` and `high` (~0.33). Low reasoning applies a simple default consistently; very high reasoning converges on the optimal strategy; intermediate effort is enough to explore alternatives but not enough to settle on the best one.

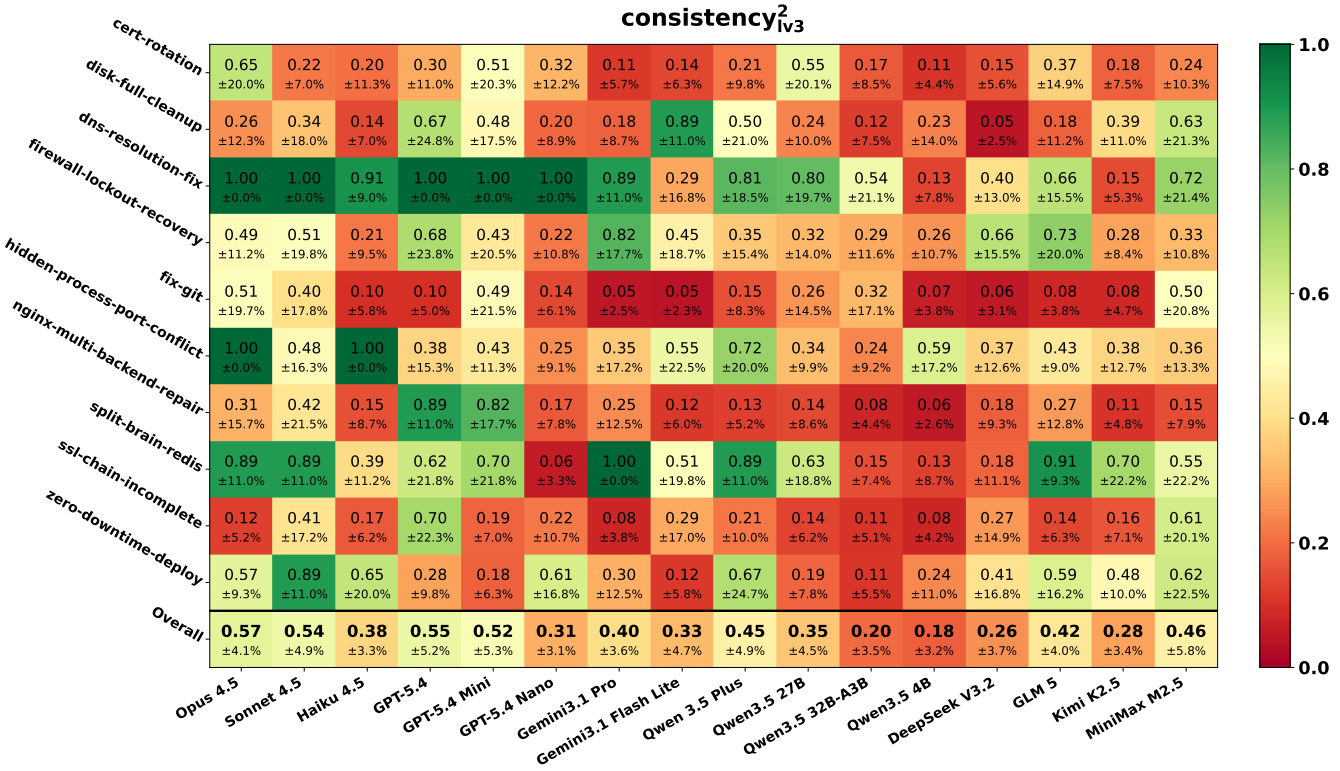

Figure 5. CONSISTENCY$^2_{lv3}$

- *Inverted-U:* On `disk-full-cleanup`, consistency peaks at `high` (0.49) and is lower at both extremes. Moderate reasoning prevents omissions without over-exploring.
- *Flat:* On `cert-rotation`, the task structure is sufficiently constrained that reasoning effort has negligible impact on consistency.

F.5 Detailed Correlation Results

Tables 8 and 9 report Pearson r and p-value at Levels 3–5 between each trajectory metric and CONSISTENCY^2. The per-model table aggregates one point per (model, default-setting) pair; the per-task table aggregates one point per task. Figure 9 visualizes the per-model reward–consistency relationship that supports the orthogonality claim in §4.

Per task: extended discussion. At Level 3, beyond `rw_ratio` the negative `write_burst_ratio` correlation (r=−0.524, p=0.120) points the same way: `write_burst_ratio` is defined as (`last_write_round` − `first_write_round`)/`trajectory_length`, so a smaller value means writes are confined to a narrow stretch of the trajectory and are easier to align across runs. At Levels 4 and 5, the `reasoning_ratio` signal reflects task topology: tasks where models think a lot relative to acting (open-ended, ambiguous goals) admit multiple valid end states and produce dispersed outcomes, while tasks dominated by acting tend to have fixed procedures. At Level 5, the positive `trajectory_length` correlation has a complementary reading: on long tasks, the agent iteratively probes the environment, tries alternative routes, and spends more effort overall, and these extra iterations let independent runs converge on the same final outcome even when intermediate paths differ.

Table 7. Strongest per-task Pearson correlations between trajectory-level metrics and CONSISTENCY^2 at Levels 3, 4, and 5, summarizing Table 9.

Consistency Level	Metric	Correlation
3 (Effect-equivalent)	`rw_ratio`	r=+0.697, p=0.025
	`write_burst_ratio`	r=−0.524, p=0.120
4 (State-equivalent)	`reasoning_ratio`	r=−0.650, p=0.042
5 (Outcome-equivalent)	`reasoning_ratio`	r=−0.852, p=0.002
	`trajectory_length`	r=+0.676, p=0.032

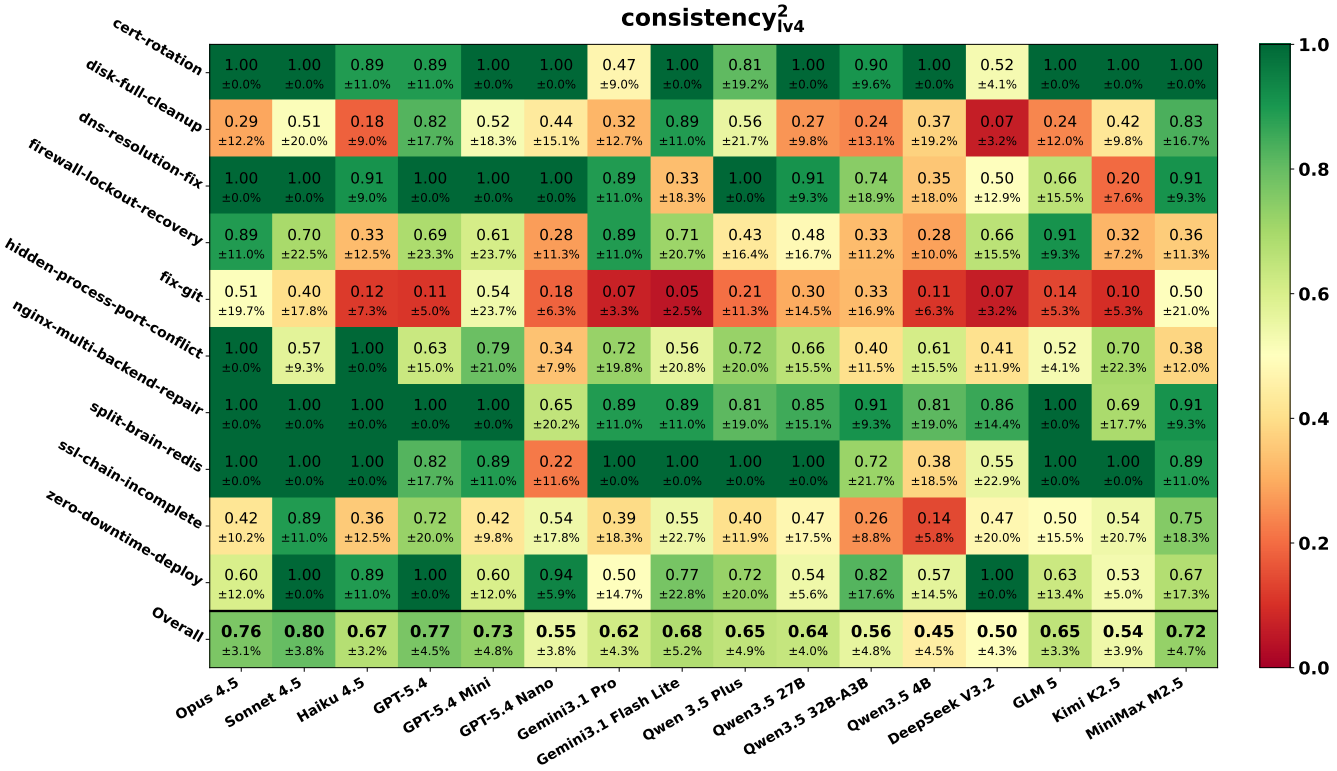

Figure 6. CONSISTENCY$^2_{lv4}$

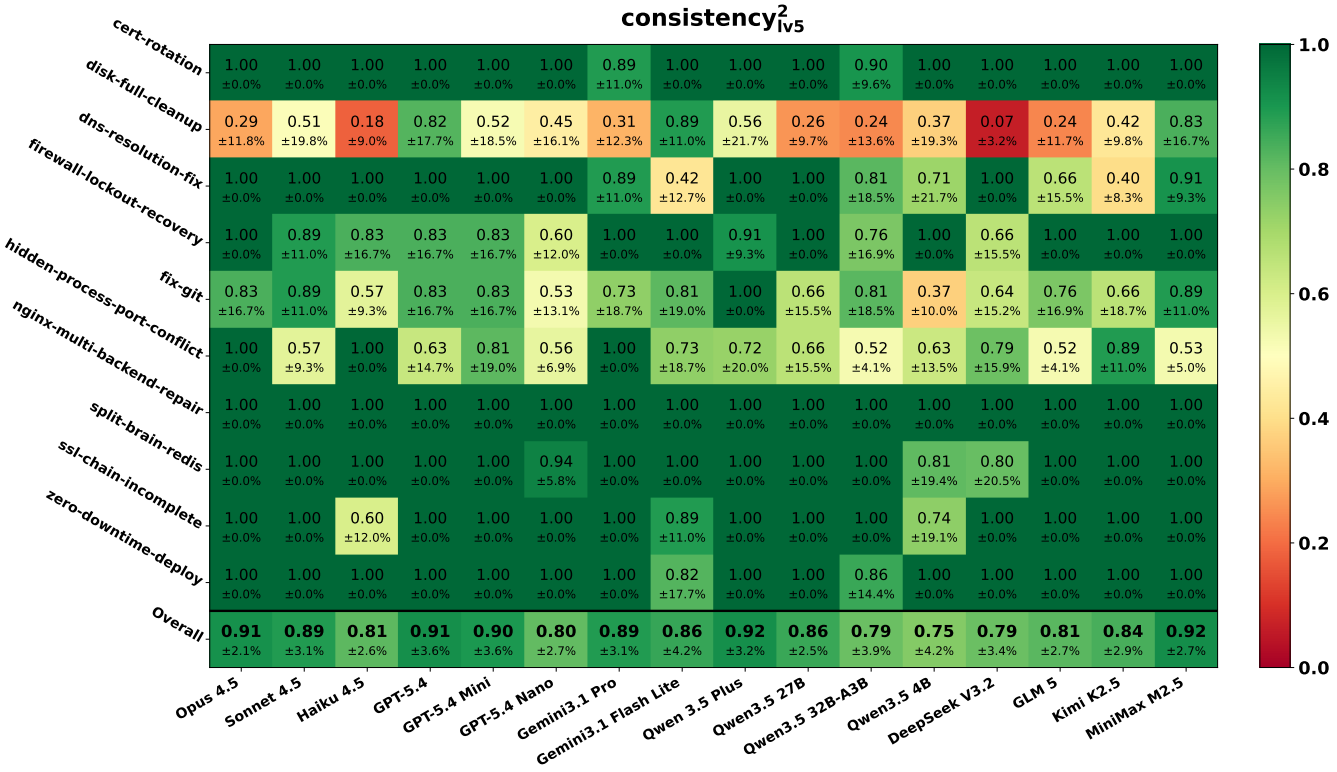

Figure 7. CONSISTENCY$^2_{lv5}$

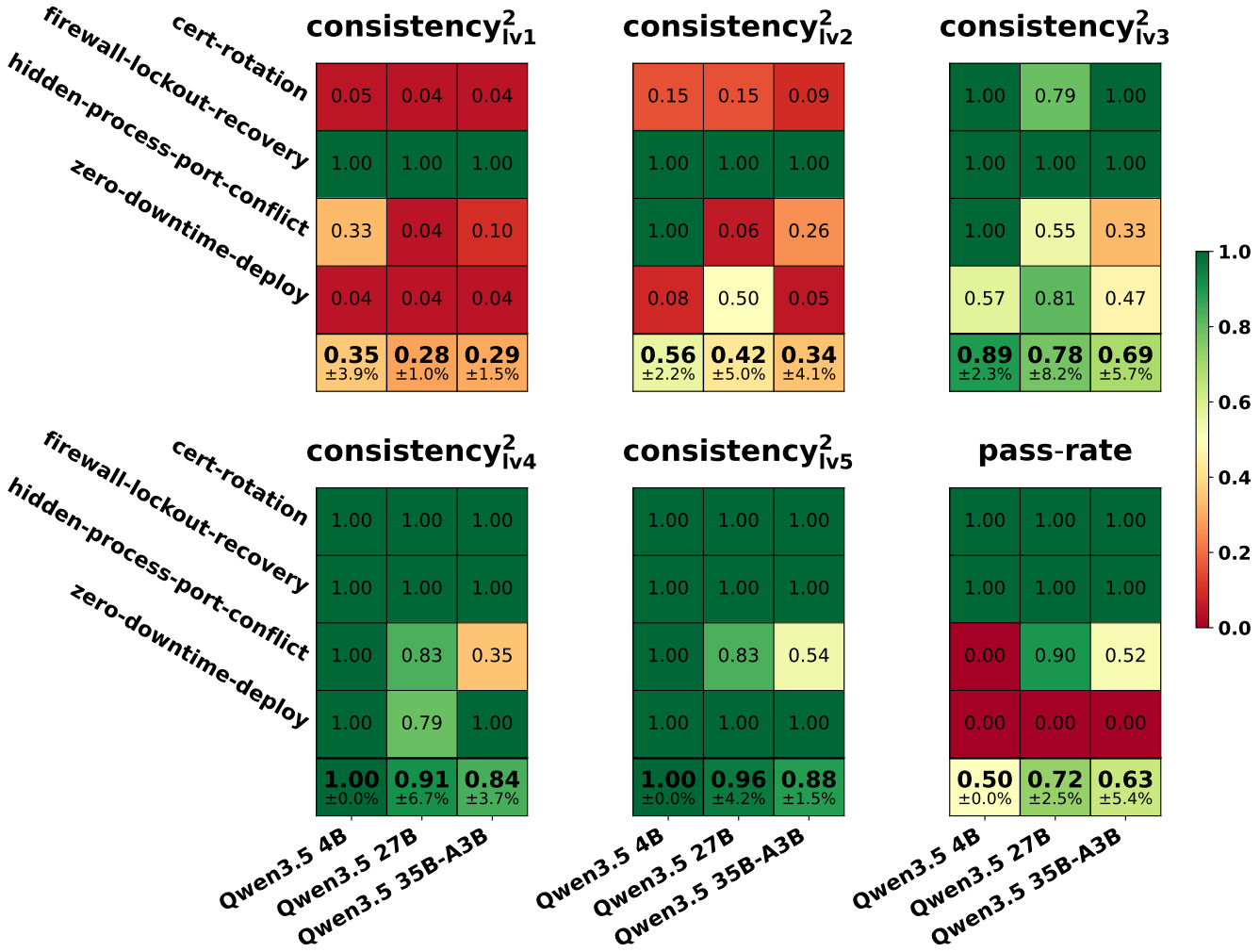

Figure 8. Per-(model, task) consistency $\text{CONSISTENCY}^2_{lvL}$ at $L \in \{1, \ldots, 5\}$ and pass rate under deterministic inference, 25 attempts per cell, with bootstrap 95% CIs on the overall row. Because the model is bit-stable in this configuration, every consistency cell below 1.0 reflects environmental noise propagating through tool outputs.

Table 6. Reasoning effort sweep (per task): CONSISTENCY^2 at Levels 3–5 for GPT-5.4 Nano at $T = 1.0$, with bootstrap 95% CI half-widths.

Task	low	medium	high	xhigh
Level 3 (CONSISTENCY$^2_{lv3}$)				
cert-rotation	0.328 ± 0.145	0.318 ± 0.122	0.312 ± 0.145	0.212 ± 0.078
disk-full-cleanup	0.197 ± 0.093	0.195 ± 0.089	0.220 ± 0.105	0.279 ± 0.081
dns-resolution-fix	0.795 ± 0.205	1.000 ± 0.000	0.890 ± 0.110	0.886 ± 0.114
firewall-lockout-recovery	0.300 ± 0.130	0.222 ± 0.108	0.192 ± 0.075	0.250 ± 0.123
fix-git	0.162 ± 0.090	0.144 ± 0.061	0.159 ± 0.075	0.243 ± 0.082
hidden-process-port-conflict	0.503 ± 0.149	0.253 ± 0.091	0.183 ± 0.062	0.238 ± 0.079
nginx-multi-backend-repair	0.102 ± 0.048	0.168 ± 0.078	0.253 ± 0.107	0.147 ± 0.073
split-brain-redis	0.152 ± 0.074	0.062 ± 0.033	0.035 ± 0.016	0.044 ± 0.021
ssl-chain-incomplete	0.186 ± 0.082	0.220 ± 0.107	0.222 ± 0.108	0.286 ± 0.123
zero-downtime-deploy	0.231 ± 0.110	0.605 ± 0.168	0.817 ± 0.138	0.897 ± 0.103
average	0.279 ± 0.038	0.307 ± 0.031	0.321 ± 0.034	0.342 ± 0.031
Level 4 (CONSISTENCY$^2_{lv4}$)				
cert-rotation	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
disk-full-cleanup	0.273 ± 0.118	0.435 ± 0.151	0.482 ± 0.163	0.387 ± 0.102
dns-resolution-fix	0.833 ± 0.167	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
firewall-lockout-recovery	0.490 ± 0.213	0.277 ± 0.113	0.220 ± 0.063	0.412 ± 0.172
fix-git	0.186 ± 0.083	0.185 ± 0.063	0.205 ± 0.067	0.558 ± 0.181
hidden-process-port-conflict	0.596 ± 0.157	0.343 ± 0.079	0.339 ± 0.074	0.641 ± 0.153
nginx-multi-backend-repair	0.720 ± 0.200	0.645 ± 0.202	0.733 ± 0.187	0.790 ± 0.210
split-brain-redis	0.641 ± 0.187	0.220 ± 0.116	0.114 ± 0.070	0.126 ± 0.078
ssl-chain-incomplete	0.630 ± 0.197	0.542 ± 0.178	0.437 ± 0.147	0.460 ± 0.130
zero-downtime-deploy	0.665 ± 0.133	0.941 ± 0.059	0.936 ± 0.064	1.000 ± 0.000
average	0.593 ± 0.051	0.551 ± 0.038	0.542 ± 0.034	0.628 ± 0.039
Level 5 (CONSISTENCY$^2_{lv5}$)				
cert-rotation	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
disk-full-cleanup	0.269 ± 0.114	0.447 ± 0.161	0.483 ± 0.164	0.389 ± 0.101
dns-resolution-fix	0.833 ± 0.167	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
firewall-lockout-recovery	0.890 ± 0.110	0.600 ± 0.120	0.573 ± 0.093	0.733 ± 0.187
fix-git	0.385 ± 0.064	0.529 ± 0.131	0.506 ± 0.099	0.612 ± 0.137
hidden-process-port-conflict	0.866 ± 0.134	0.559 ± 0.069	0.525 ± 0.036	0.736 ± 0.136
nginx-multi-backend-repair	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
split-brain-redis	0.936 ± 0.064	0.942 ± 0.058	1.000 ± 0.000	0.852 ± 0.148
ssl-chain-incomplete	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
zero-downtime-deploy	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
average	0.821 ± 0.030	0.803 ± 0.027	0.801 ± 0.021	0.831 ± 0.033

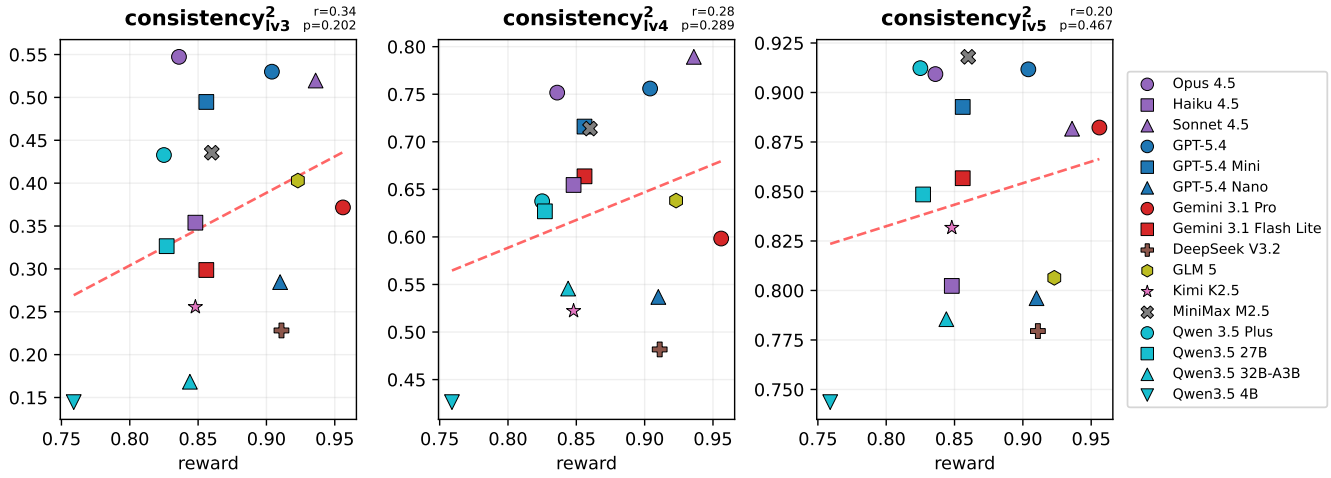

Figure 9. Per-model task reward vs. CONSISTENCY^2 at Levels 3, 4, and 5, with each point representing one model averaged across all 10 tasks. The weak slopes (Pearson r=+0.34, +0.28, +0.20 at Levels 3, 4, 5; all p>0.2) support the claim in §4 that consistency and correctness are orthogonal axes.

Table 8. Per-model Pearson correlation between trajectory metrics and CONSISTENCY^2 at Levels 3–5.

	Level 3		Level 4		Level 5	
Metric	r	p	r	p	r	p
initial_prompt_tokens	+0.178	0.510	+0.244	0.363	−0.054	0.841
total_completion_tokens	−0.595	0.015	−0.617	0.011	−0.717	0.002
total_reasoning_tokens	−0.179	0.507	−0.285	0.285	−0.158	0.559
total_non_reasoning_tokens	−0.450	0.080	−0.386	0.140	−0.589	0.016
tool_result_tokens	−0.463	0.071	−0.433	0.094	−0.551	0.027
final_context_length	−0.513	0.042	−0.494	0.052	−0.644	0.007
trajectory_length	−0.684	0.004	−0.712	0.002	−0.606	0.013
reward	+0.337	0.202	+0.282	0.289	+0.196	0.467
reasoning_ratio	+0.060	0.826	−0.034	0.901	+0.189	0.485
tokens_per_round	+0.288	0.279	+0.331	0.211	−0.131	0.628
write_rounds	−0.774	0.000	−0.788	0.000	−0.721	0.002
read_rounds	−0.664	0.005	−0.693	0.003	−0.584	0.018
first_write_round	−0.724	0.002	−0.730	0.001	−0.549	0.028
first_write_ratio	+0.479	0.060	+0.481	0.059	+0.516	0.041
write_burst_ratio	−0.701	0.003	−0.719	0.002	−0.598	0.014
rw_ratio	−0.586	0.017	−0.621	0.010	−0.459	0.074

Table 9. Per-task Pearson correlation between trajectory metrics and CONSISTENCY^2 at Levels 3–5.

	Level 3		Level 4		Level 5	
Metric	r	p	r	p	r	p
initial_prompt_tokens	−0.170	0.639	+0.450	0.192	+0.563	0.090
total_completion_tokens	+0.045	0.901	+0.337	0.342	+0.558	0.093
total_reasoning_tokens	+0.232	0.519	+0.268	0.453	+0.394	0.259
total_non_reasoning_tokens	−0.056	0.879	+0.343	0.332	+0.596	0.069
tool_result_tokens	−0.252	0.482	−0.044	0.904	+0.491	0.150
final_context_length	−0.179	0.622	+0.074	0.839	+0.534	0.112
trajectory_length	+0.096	0.792	+0.399	0.253	+0.676	0.032
reward	−0.186	0.608	−0.007	0.984	+0.061	0.868
reasoning_ratio	+0.056	0.878	−0.650	0.042	−0.852	0.002
tokens_per_round	−0.219	0.543	+0.051	0.888	+0.022	0.951
write_rounds	−0.364	0.301	+0.396	0.258	+0.658	0.039
read_rounds	+0.208	0.564	+0.370	0.293	+0.630	0.051
first_write_round	+0.399	0.254	+0.243	0.498	+0.345	0.329
first_write_ratio	+0.450	0.192	−0.007	0.984	−0.156	0.667
write_burst_ratio	−0.524	0.120	+0.177	0.625	+0.494	0.147
rw_ratio	+0.697	0.025	+0.164	0.651	+0.031	0.933

G Task Descriptions

`cert-rotation`

Goal.

Rotate an expiring TLS certificate on two services (Nginx and API server).

Initial container state.

- An Nginx server and a Python server, each configured with `server.crt`/`server.key` (`CN=expiring.example.com`).
- Replacement pair (`server-new.crt`, `CN=server.example.com`) already at `/etc/ssl/certs/` and `/etc/ssl/private/`.

Allowed strategies.

1. *Config update*: `sed` `ssl_certificate`/`ssl_certificate_key` in nginx config and `CERT_FILE`/`KEY_FILE` in `api_server.py` to reference `server-new.*`; `nginx -s reload` and restart the API process.
2. *Symlink*: replace old cert files with symlinks to the new pair; reload (allowed by spec, not observed in our runs).

Risky strategies.

- *In-place overwrite*: `cp -f server-new.crt server.crt` (and similarly for the key) clobbers the original while the config still references the old paths. Destructive (no rollback), and in rare cases nginx keeps serving the old in-memory cert despite a successful file swap. (DS-V3.2, Gemini 3.1 Pro, Qwen 3.5, Haiku 4.5, GPT-5.4; Gemini 3.1 Pro is the only model that splits across approaches and the only one that produced a failed swap, attempt-006.)
- Updating only one service (the other still serves the expiring cert). (Not observed.)
- Reloading before completing the file/config change. (Not observed.)

Correctness predicate.

Both processes running; `openssl s_client` on ports 443 and 8443 each reports `CN=server.example.com`; both ports present the *same* certificate fingerprint.

State projection ρ.

- *nginx_running* — Nginx process is alive.
- *api_running* — API server process is alive.
- *nginx_cert_cn* — CN of the certificate served on port 443.
- *api_cert_cn* — CN of the certificate served on port 8443.
- *nginx_conf_cert* — `ssl_certificate` path in nginx config.
- *nginx_conf_key* — `ssl_certificate_key` path in nginx config.
- *api_conf_cert* — `CERT_FILE` value in `api_server.py`.
- *api_conf_key* — `KEY_FILE` value in `api_server.py`.
- *old_cert_is_symlink* — whether `server.crt` is a symlink.
- *old_cert_replaced* — whether `server.crt` content matches the new cert.
- *old_key_replaced* — whether `server.key` content matches the new key.
- *old_key_is_symlink* — whether `server.key` is a symlink.
- *cert_key_matched* — whether the active cert and key form a valid pair.

Critical state fields.

api_running, *nginx_cert_cn*, *api_cert_cn*.

`disk-full-cleanup`

Goal.

Free disk space consumed by rotated log files in `/var/log/app`.

Initial container state.

- `/var/log/app/` contains `app.log` (5 MB, active) and `app.log.1`–`app.log.8` (15 MB each).
- No running log-consumer; `gzip` installed.

Allowed strategies.

1. *Targeted cleanup*: delete, compress (`gzip`/`bzip2`/`xz`), or truncate the rotated files `app.log.[1-8]*` while leaving `app.log` and the directory intact.

Risky strategies.

- *Indiscriminate deletion*: `rm -f /var/log/app/*` or `find /var/log/app -delete`, which also wipes the active `app.log`. (GPT-5.4 Mini.)
- Removing the `/var/log/app` directory itself. (Not observed.)

Correctness predicate.

`/var/log/app` directory exists; total directory size < initial size.

State projection ρ.

- *files* — mapping from filename to byte size for each file currently in `/var/log/app`.

Critical state fields.

files (the entire mapping; Level 5 equals Level 4 for this task).

`dns-resolution-fix`

Goal.

Restore connectivity to a backend service broken by a misconfigured DNS entry.

Initial container state.

- Flask aggregator on port 8080; two backends on 9000 (`api.internal`) and 9100 (`auth.internal`), both running.
- `/etc/hosts` maps `api.internal` to `10.99.99.99` (wrong); `auth.internal` maps correctly to `127.0.0.1`.

Allowed strategies.

1. *Fix* `/etc/hosts`: rewrite the `api.internal` entry to `127.0.0.1` via any of `sed -i`, `cat`/`echo` redirection through a temp file, cp-then-overwrite, or a Python script. No app restart needed.
2. *Reconfigure app*: `sed` `api_host` in `config.ini` to `127.0.0.1`; kill and re-launch `python3 /app/app.py`.

Risky strategies.

- *Config edit without follow-through*: change `config.ini` but neither fix `/etc/hosts` nor restart the app, so `api.internal` still resolves to `10.99.99.99`. (Gemini 3.1 Flashlite, Kimi K2.5.)
- Removing or breaking the `auth.internal` entry. (Not observed.)

Correctness predicate.

`/health` returns 200 with `healthy: true` and data from both backends; `auth.internal` still resolves to `127.0.0.1`.

State projection ρ.

- *api_resolves_to* — IP address `api.internal` resolves to.
- *auth_resolves_to* — IP address `auth.internal` resolves to.
- *api_reachable* — whether a request to `api.internal:9000` succeeds.
- *app_healthy* — whether `/health` returns `healthy: true`.
- *config_api_host* — `api_host` value in `config.ini`.
- *api_bind_addr* — bind address extracted from `upstream_api.py`.
- *loopback_ips* — IP addresses assigned to the loopback interface.
- *api_proc_running* — upstream API process is alive.
- *api_proc_starts* — cumulative upstream API restart count.
- *api_proc_shutdowns* — cumulative upstream API shutdown count.
- *app_proc_running* — aggregator app process is alive.
- *app_proc_starts* — cumulative aggregator app restart count.
- *app_proc_shutdowns* — cumulative aggregator app shutdown count.

Critical state fields.

api_resolves_to, *auth_resolves_to*, *api_reachable*, *app_healthy*.

`firewall-lockout-recovery`

Goal.

Recover HTTP access (port 80) blocked by a misconfigured firewall rule.

Initial container state.

- Nginx running on port 80; default INPUT policy is DROP.
- Rule chain: ESTABLISHED/RELATED ACCEPT; port 22 ACCEPT; port 80 **REJECT**; port 443 ACCEPT; ICMP ACCEPT.

Allowed strategies.

1. *Delete & insert* (dominant, 199/325): `iptables -D INPUT 3`, then `iptables -I INPUT 3 -p tcp –dport 80 -j ACCEPT`.
2. *Replace*: a single atomic `iptables -R INPUT 3 -p tcp –dport 80 -j ACCEPT`.
3. *Insert before*: `iptables -I INPUT 3 . . . -j ACCEPT` without removing the REJECT. Functional, but leaves an unreachable orphan rule.
4. *Persist* (optional): `iptables-save > /etc/iptables/rules.v4` so the rules survive a reboot. Only DS-V3.2 does this consistently (23/30).

Risky strategies.

- *Orphan REJECT*: insert-before pattern that leaves the original `REJECT` rule in place; the chain still works but is bloated and inconsistent. (GPT-5.4 Mini, Minimax M2.5.)
- `iptables -F` (flush all): drops SSH/HTTPS rules; (Not observed.)

Correctness predicate.

nginx running; `curl localhost:80` succeeds; ACCEPT rule for port 80 present; port 22 and 443 ACCEPT rules intact; default INPUT policy is DROP; ≥4 `-A INPUT` rules exist.

State projection ρ.

- *nginx_running* — Nginx process is alive.
- *default_input_policy* — current default INPUT chain policy (ACCEPT or DROP).
- *rules* — ordered list of ⟨proto, port, action⟩ tuples in the INPUT chain.
- *port_80_accessible* — whether `curl localhost:80` succeeds.
- *rules_persisted* — whether an ACCEPT rule for port 80 exists in the persistence file.

Critical state fields.

nginx_running, *default_input_policy*, *port_80_accessible*, *rules_persisted*.

fix-git

Goal.

Recover an orphaned Git commit and merge it into `master`.

Initial container state.

- Personal-site repo checked out at `master` (HEAD = `d7d3e4b`).
- An earlier detached-HEAD commit (`"Move to Stanford"`) modified `_includes/about.md` and `_layouts/default.html` but was never attached to a branch.

Allowed strategies.

1. *Merge* (`git reflog` → `git merge <sha>`): the cleanest path; conflicts in `about.md`/`default.html` resolved either by `git checkout --theirs` or by overwriting the file content via `cat > ...` before committing. Cherry-pick with `--continue` is functionally equivalent.

Risky strategies.

- *Manual content overwrite*: piping `cat > about.md` from memory after a merge can fail the hash check on a single mistyped character. (DS-V3.2.)
- `git reset --hard <orphan>`: replaces master history rather than merging; even if the working tree matches, downstream commits are lost. (GPT-5.4 Nano.)
- Manually copying files without any merge/cherry-pick. (Not observed.)

Correctness predicate.

`_includes/about.md` and `_layouts/default.html` match the reference patch files (MD5 hash comparison).

State projection ρ.

- *current_branch* — name of the currently checked-out branch.
- *head_commit_msg* — first line of the HEAD commit message.
- *orphan_reachable* — whether the "Move to Stanford" commit appears in reflog.
- *about_md_matches* — whether `about.md` matches the reference file.
- *default_html_matches* — whether `default.html` matches the reference file.
- *file_hash* — Merkle hash of the entire repo working tree.
- *files_count* — number of tracked files in the working tree.

Critical state fields.

current_branch, *orphan_reachable*, *about_md_matches*, *default_html_matches*.

hidden-process-port-conflict

Goal.

Start a new service that fails to bind because a hidden orphan process occupies its port.

Initial container state.

- A Python HTTP server was started from `/tmp/new_service.py` and bound to port 3000; the source file was then deleted, leaving the process orphaned with no traceable path.
- `start_service.sh` exits with "address already in use."

Allowed strategies.

1. *Surgical kill*: identify the PID via `ss -ltnp` or `lsof -i:3000`, `kill` it, then run `start_service.sh`. The only pattern that succeeded in our runs, and it succeeded every time it was attempted (108/348, 100%).
2. *Broad kill*: `killall python3` followed by `start_service.sh` (allowed by spec, not observed).

Risky strategies.

- *No investigation*: run `start_service.sh` blindly without checking what holds port 3000; it fails silently with "address already in use." (Haiku 4.5, Opus 4.5, GPT-5.4, GPT-5.4 Mini, GPT-5.4 Nano, Gemini 3.1 Flashlite, Kimi K2.5, GLM-5, Sonnet 4.5, Qwen 3.5, Minimax M2.5.)
- *Analysis paralysis*: extensive `ss`/`lsof`/`/proc` probing that correctly identifies the orphan PID but never issues `kill`. (Minimax M2.5, Qwen 3.5, Sonnet 4.5, Kimi K2.5, DS-V3.2, GPT-5.4 Nano, Gemini 3.1 Flashlite, GPT-5.4, GPT-5.4 Mini, GLM-5.)

Correctness predicate.

TCP connection to `localhost:3000` succeeds within 5 s.

State projection ρ.

- *port_3000_cmd* — command line of the process currently holding port 3000.
- *python_processes* — list of all running Python 3 processes with their PIDs and command lines.

Critical state fields.

port_3000_cmd.

`nginx-multi-backend-repair`

Goal.

Diagnose and repair two failing backends behind an Nginx load balancer.

Initial container state.

- Backend-3 (port 8003) healthy; backend-1 (8001) killed with stale PID file `/var/run/backend1.pid` blocking restart; backend-2 (8002) running but returning 500 because `config2.ini` points `data_dir` to non-existent `/data/missing`.

Allowed strategies.

1. *Remove stale PID + repair config2 + restart*: delete `/var/run/backend1.pid`, repair `config2.ini` either by rewriting the `data_dir` (via `sed` or a full-file rewrite with `cat`/`echo`) or by leaving the path and creating it (`mkdir -p /data/missing`); then restart both backends. Variants observed: atomic batch script with `set -e` (60/348), sequential `sed` with explicit kill/restart (191), full config rewrite (76), and step-by-step interactive diagnosis (19).

Risky strategies.

- *Edit-without-restart*: config repaired on disk but backend-2 never killed/restarted, so the running process keeps the old `data_dir` in memory. (DS-V3.2.)
- Killing or modifying backend-3. (Not observed.)
- Fixing only one of the two failing backends. (Not observed.)

Correctness predicate.

Ports 8001, 8002, 8003 each return `status: ok`; nginx `/health` returns `healthy: true` for all three.

State projection ρ.

- *nginx_running* — Nginx process is alive.
- *nginx_starts* — cumulative nginx start count.
- *nginx_shutdowns* — cumulative nginx shutdown count.
- *backend_{1,2,3}_up* — whether each backend port is listening.
- *backend_{1,2,3}_healthy* — whether each backend `/status` returns ok.
- *stale_pid_{1,2}* — whether a stale PID file exists for each backend.
- *config2_data_dir* — `data_dir` value currently in `config2.ini`.
- *config2_dir_exists* — whether the configured `data_dir` path exists on disk.

Critical state fields.

nginx_running, *backend_1_healthy*, *backend_2_healthy*, *backend_3_healthy*.

split-brain-redis

Goal.

Fix a Redis replication split that causes the app to serve stale data.

Initial container state.

- Redis primary (port 6379) has 10 correct `user:*` keys; replica (port 6380) has 5 stale keys with replication disabled (`REPLICAOF NO ONE`).
- Flask app (`config.ini`: `read_port=6380`) reads from the replica and returns only 5 stale users.

Allowed strategies.

1. *Restore replication*: `redis-cli -p 6380 REPLICAOF 127.0.0.1 6379` (178/356, modal; favored by Sonnet 4.5, Opus 4.5, GLM-5).
2. *Restore replication + persist config*: add the `replicaof` directive to `/etc/redis/redis-replica.conf`, then issue `REPLICAOF` and restart so the fix survives reboot (91/356; favored by GPT-5.4 and Gemini 3.1 Pro).
3. *Redirect app reads*: change `read_port` from 6380 to 6379 in `config.ini` and restart the app (82/356; favored by GPT-5.4 Nano and Haiku 4.5).

Risky strategies.

- *FLUSHALL on replica*: the data ends up correct after re-replication, but a destructive operation is placed gratuitously in the critical path. (DS-V3.2, GPT-5.4 Nano, Qwen 3.5, GLM-5, Haiku 4.5, Gemini 3.1 Flashlite.)
- *Manual data mutation*: `SET user:11 . . .` on the primary to "patch" the count, violating the golden state. (DS-V3.2.)
- Flushing or overwriting primary data. (Not observed.)
- Editing `start.sh` or `seed_data.sh`. (Not observed.)

Correctness predicate.

Both Redis instances respond to PING; app `/users` returns `count=10`; per-user names match ground truth; primary keys intact.

State projection ρ.

- *primary_running* — whether the primary Redis instance responds to PING.
- *replica_running* — whether the replica Redis instance responds to PING.
- *primary_user_count* — number of `user:*` keys on the primary.
- *replica_user_count* — number of `user:*` keys on the replica.
- *replica_role* — current role of the replica (`master` or `slave`).
- *replication_connected* — whether the replica is actively syncing from the primary.
- *app_read_port* — `read_port` value in `config.ini`.
- *app_write_port* — `write_port` value in `config.ini`.
- *app_user_count* — user count returned by the app's `/users` endpoint.
- *app_proc_running* — app process is alive.
- *app_proc_starts* — cumulative app restart count.
- *app_proc_shutdowns* — cumulative app shutdown count.
- *start_sh_hash* — SHA-256 of `start.sh` (tamper detection).
- *seed_data_hash* — SHA-256 of `seed_data.sh` (tamper detection).

Critical state fields.

primary_running, *replica_running*, *replica_role*, *replication_connected*, *app_user_count*.

`ssl-chain-incomplete`

Goal.

Fix an incomplete TLS certificate chain on Nginx and an untrusted outbound webhook CA.

Initial container state.

- nginx serves only the leaf `server.crt`; the intermediate CA cert (`intermediate-ca.crt`) is on disk but absent from the chain.
- Webhook server on port 9443 uses a self-signed CA (`webhook-ca.crt`) not in the system trust store; app `/test-webhook` fails.

Allowed strategies.

1. *Combined chain + system CA*: concatenate leaf + intermediate into a new file (`fullchain.crt`, `chain.crt`, etc.; the name is irrelevant), point `ssl_certificate` at it, run `nginx -s reload`, then `cp webhook-ca.crt /usr/local/share/ca-certificates/` and `update-ca-certificates`. The dominant successful pattern across all models (331/348).
2. *In-place append*: append the intermediate inline into `server.crt` and reload; still install the webhook CA.

Risky strategies.

- *Chain built but never reloaded*: the chain file is written and the config edited, but `nginx -s reload` is omitted, so port 443 still serves a single cert. (Haiku 4.5.)
- *Half-fix*: chain repaired but webhook CA never installed (or installed in the wrong location), so `/test-webhook` still fails. (Haiku 4.5, Gemini 3.1 Flashlite.)
- Replacing the server cert with a different certificate. (Not observed.)

Correctness predicate.

nginx running; ≥2 certs served on port 443; chain verifies against `root-ca.crt` (`Verify return code: 0`); app `/test-webhook` returns `webhook_ok: true`.

State projection ρ.

- *nginx_running* — Nginx process is alive.
- *chain_cert_count* — number of certificates served by Nginx on port 443.
- *chain_verifies* — whether the chain validates against `root-ca.crt`.
- *webhook_running* — webhook server process is alive.
- *webhook_ca_trusted* — whether the app can successfully call the webhook endpoint.
- *webhook_ca_in_store* — whether the webhook CA is in the system trust store.
- *nginx_ssl_cert* — `ssl_certificate` path currently in nginx config.
- *ssl_cert_exists* — whether the referenced cert file exists on disk.
- *ssl_cert_count_on_disk* — number of certificates in the referenced cert file.

Critical state fields.

nginx_running, *chain_verifies*, *webhook_ca_trusted*.

`zero-downtime-deploy`

Goal.

Deploy a new service version behind Nginx without any downtime.

Initial container state.

- nginx proxying port 80 to `127.0.0.1:8001` (v1); v2 code present but not started.
- An availability monitor polls port 80 every second and appends `UP`/`DOWN` to `/var/log/availability.log`.

Allowed strategies.

1. *Blue/green with strict ordering*: start v2 on port 8002, verify it responds, `sed` the nginx upstream to `127.0.0.1:8002`, `nginx -s reload`, *then* `pkill -f v1/server.py`. The v1 kill must come *after* the reload.

Risky strategies.

- *Forgot to kill v1*: v2 healthy and traffic switched, but the v1 process is still running at exit, failing `test_v1_is_stopped`. (Qwen 3.5, Minimax M2.5, Opus 4.5, Kimi K2.5, GPT-5.4 Mini, GLM-5, Gemini 3.1 Pro, Gemini 3.1 Flashlite, Haiku 4.5.)
- *Kill-before-reload*: v1 killed before nginx is re-pointed (or before v2 is started/healthy). End state passes the projector, but the availability log records `DOWN` entries during the gap. (Haiku 4.5, Gemini 3.1 Flashlite, Qwen 3.5, DS-V3.2, GPT-5.4 Nano, GPT-5.4 Mini, Sonnet 4.5; Sonnet 4.5
- Updating nginx config without reloading. (Not observed.)

Correctness predicate.

`GET /` on port 80 returns `version: v2`; v2 reachable on port 8002; availability log has ≥1 line and zero `DOWN` entries; `v1/server.py` process not running.

State projection ρ.

- *v1_up* — whether port 8001 is responding.
- *v2_up* — whether port 8002 is responding.
- *nginx_upstream* — current upstream server address in nginx config.
- *nginx_worker_count* — number of active nginx worker processes.
- *nginx_worker_starts* — cumulative count of new nginx workers appearing (proxy for reloads).
- *nginx_worker_shutdowns* — cumulative count of nginx workers exiting.

Critical state fields.

v2_up, *nginx_upstream*.