AgentLTL: A Trace-Verification Framework for Measuring, Enforcing, and Training Procedural Compliance in Tool-Using LLM Agents

Anonymous ACL submission

Abstract

Tool-using LLM agents are usually evaluated by final-answer correctness or LLM judges. Neither captures how an answer was produced. In safety-critical settings, the procedure itself is part of correctness. In this paper, we introduce AGENTLTL, a language derived from First-Order Linear Temporal Logic (FO-LTL) that expresses procedural rules over agent traces. It yields a deterministic, judge-free compliance score. In this framework, a single specification drives two usages. The first is harnessing: the constraints score completed traces, or gate tool calls by checking each prefix online, before execution. The second is finetuning: the score serves as a dense reward. On a benchmark spanning ordering, branching, iteration, and grounding, block-and-warn harnessing improves compliance on five of seven models. Finetuning with the same reward yields +38 and +17.5 percentage point gains in accuracy and compliance on held-out patterns, including unseen tool-name aliases. These findings are consistent with the model acquiring procedural structure rather than memorizing surface tool names and procedures.

1 Introduction

Tool-augmented LLM agents are increasingly used in enterprise and safety-critical settings, yet evaluation still focuses on final-answer accuracy, often with LLM judges. Two traces can produce the same answer while differing in retrieval, branching, or tool grounding. Answer correctness collapses these differences into a single outcome.

In many settings, the procedure is part of correctness. A clinical triage agent that skips a required contraindication check is not correct, even if its recommendation is. Figure 1 illustrates this common shortcut: the agent answers from parametric memory instead of tool outputs. This fails silently when memory and environment diverge.

We introduce AGENTLTL, a language derived from First-Order Linear Temporal Logic (FO-LTL) for expressing procedural, semantic, and grounding constraints over execution traces. Comparing a trace against these constraints yields a deterministic, judge-free compliance score. The same constraints support two uses. When the LTL fragment permits, they gate tool calls before execution, detecting and blocking violating calls. The score also serves as a reinforcement learning reward. A shared constraint language avoids drift between evaluation, deployment, and training.

Contributions. (i) We introduce AGENTLTL and its compliance score. (ii) We build a benchmark of 12 workflow templates and evaluate 7 language models under three harnesses that intervene on the trace at different strengths, localizing failures in ordering, branching, iteration, and argument grounding. (iii) We show that the same score is an effective reward: finetuning a target model improves both compliance and answer correctness on held-out templates, including unseen tool-name aliases. (iv) We show that grounding constraints can detect parametric-memory hallucination by distinguishing supported answers from unsupported recall. AGENTLTL, the benchmark, training corpus, and scripts are available here.

2 Related Work

Tool-augmented LLM agents. The tool-using paradigm began with ReAct (Yao et al., 2023) and expanded through finetuning (Schick et al., 2023; Patil et al., 2023), larger tool inventories (Liu et al., 2025; Li et al., 2023), and reliability benchmarks such as τ-bench (Yao et al., 2024) and AgentBoard (Ma et al., 2024). BFCL (Patil et al., 2025) evaluates function calling against gold abstract syntax trees, while τ^2-bench (Barres et al., 2025) studies dual-control settings where agents and users jointly modify shared state. OSWorld (Xie et al.,

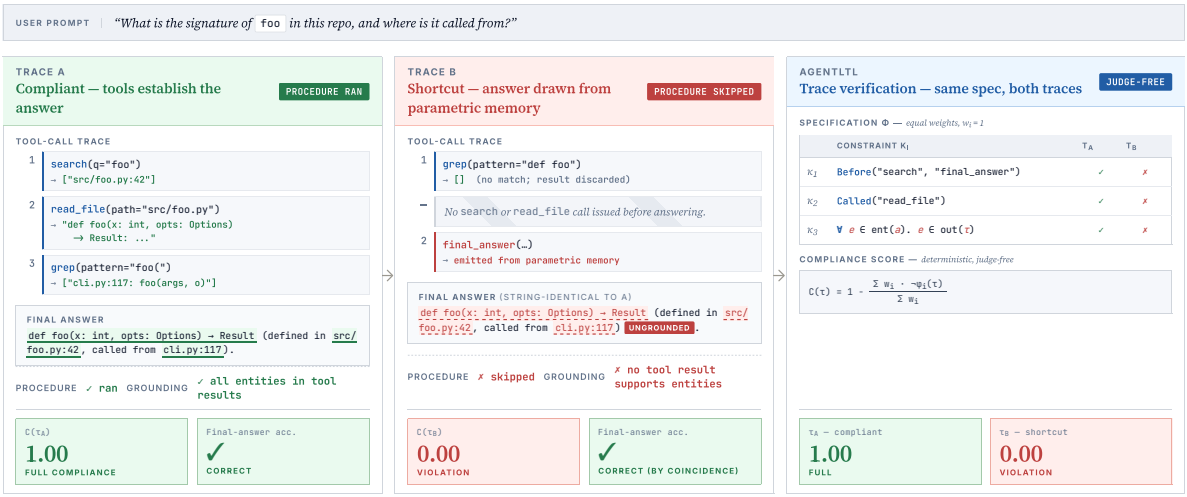

Figure 1: **Identical final answers from distinct execution traces.** Two runs of a code-QA agent given identical prompts. *Left (Trace A):* the agent calls `search`, `read_file`, and `grep`, producing an answer supported by retrieved evidence. *Middle (Trace B):* the agent emits the same answer without retrieval calls. Both traces are correct under answer correctness. *Right:* the same AGENTLTL procedure P. It verifies three properties of each trace, ordering (κ_1): `search` precedes the final answer, call presence (κ_2): `read_file` is invoked, and trace grounding (κ_3): every entity cited in the answer has a witness in some tool result). Trace A satisfies all three; Trace B violates each.

2024b) studies long-horizon failures, and Rosset et al. (2026) separate process and outcome rewards for computer-use agents. A common finding is the compositionality gap (Mohammadi et al., 2025): agents succeed on atomic calls but fail when calls must be composed. Existing evaluations stop at per-call correctness or rely on LLM judges. AGENTLTL instead scores the temporal structure of the trace with a deterministic metric.

Runtime enforcement and pre-execution guardrails. Recent systems intercept tool calls before execution. AgentSpec (Wang et al., 2026) applies per-step DSL rules, AEGIS (Yuan et al., 2026) adds runtime auditing, Winston et al. (2026) compile policies into SMT-LIB checks, and PCAS (Palumbo et al., 2026) models multi-agent state with Datalog queries. Ta et al. (2026) introduces Reinforced Agent, a dual agent architecture with a reviewer to vet tool calls. These systems focus on enforcement and provide limited temporal reasoning over multi-step traces. The agent-reviewer approach also introduces nondeterministic reviewer errors. AGENTLTL uses a single specification language for both offline evaluation and online enforcement, with deterministic checks over ordered and parameterized traces.

Formal methods and temporal logic for LLM verification. LTL (Pnueli, 1977) is the standard temporal formalism, with first-order extensions for entity quantification (Xu et al., 2026; Bojańczyk et al., 2011). Yang et al. (2024) translate constraints into LTL to prune unsafe robot plans, Ramani et al. (2025) verify LLM plans against LTL, and VLTL-Bench (English et al., 2025) surveys the persistent grounding problem in NL-to-LTL translation. Runtime verification over data words (Basin et al., 2012) also reasons about parametric events. With AGENTLTL, we avoid the NL translation step by authoring constraints directly over typed tool signatures, with grounding provided by the recorded call and its response.

Process conformance checking. Process conformance checking has been extensively studied in process mining (van der Aalst, 2016), where event logs are validated against prescriptive workflow models. DECLARE (Pesic et al., 2007) introduced a declarative framework based on LTL$_f$ constraints over finite traces, which directly motivates AGENTLTL's specification language. More recently, Klessascheck et al. (2024) emphasize online and streaming-based enforcement in contrast to purely retrospective auditing. AGENTLTL builds on this line of work by adapting conformance checking to tool-augmented LLM agents, modeling tool invocations as typed events grounded in explicit entities and execution context.

Characterization of agent failures. A separate line of work has sought to characterize how agents

fail in practice. MAST (Cemri et al., 2025) applies grounded theory to 1600+ multi-agent traces and identifies 14 failure modes, scaled with an LLM-as-a-Judge pipeline. TravelPlanner (Xie et al., 2024a) reports complementary patterns in single-agent planning, including argument errors, dead loops, and uncorrected early hallucinations, evaluated through keyword-driven rules. Both depend on a judge to reach a verdict. AGENTLTL scores the trace directly against a typed FO-LTL specification, removing the judge and localizing failures to specific procedural layers.

Hallucination detection and entity grounding. Agentic hallucination is often entity-level. Models emit plausible identifiers from parametric memory instead of tool outputs (Ji et al., 2023), and a single incorrect entity can corrupt downstream calls (Bang et al., 2025). Post-hoc factuality systems such as Factool (Chern et al., 2023) verify outputs after generation, while existing enforcement systems do not treat grounding as a core property. With AGENTLTL, we express grounding directly as a trace constraint that requires every entity in the final answer to appear in prior tool outputs.

3 The AGENTLTL Framework

3.1 Preliminaries

We model a tool-using agent as a ReAct type of system (Yao et al., 2023):

$$\mathcal{A} = \langle \mathcal{M}, \mathcal{T}, \mathcal{E}, \mathcal{H} \rangle,$$

where $\mathcal{M}$ is a language model, $\mathcal{T}$ a tool set, $\mathcal{E}$ an execution environment, and $\mathcal{H}$ the context window. At step i, the model produces a thought θ_i and proposes a tool invocation with name $n_i \in \mathcal{T}$ and arguments a_i. The environment executes the invocation and returns a result r_i, which is appended to the context $\mathcal{H}$. We record the step as the tuple

$$c_i = (n_i,\, a_i,\, r_i,\, i).$$

AGENTLTL operates on the resulting execution trace $\tau = \langle c_0, \ldots, c_{n-1} \rangle$. It also uses a metrics record μ that stores aggregate statistics over the trace, such as token and tool call counts.

3.2 Formula Language

AGENTLTL specifies properties over execution traces using a fragment of FO-LTL. The language is used to express procedural rules such as "a search must be cited before the final answer" or "every opened file must eventually be closed." Full syntax and semantics appear in Appendix A.

Formulas are evaluated against a trace τ. A formula φ can contain variables ranging over values observed in the trace, such as tool arguments, retrieved entities, or file names. Under a specific trace and a variable assignment, φ evaluates to a boolean. A formula without free variables is a *constraint*, denoted by κ, and is either satisfied or violated by τ. The language has three components.

Atomic propositions. Atomic propositions describe observable properties of tool calls, including occurrence, ordering, argument matching, return-value checks, and aggregate statistics. For example, $\mathsf{CalledWith}(\texttt{search}, \{q : \text{“foo”}\})$ holds when a call to `search` was issued with argument $q =$ “foo”. The language also supports *open predicates*, user-defined Boolean functions for domain-specific checks. For example, a user could define $\mathsf{IsEven}(x)$ to check whether a given integer is even and use it within formulas.

Temporal operators. AGENTLTL includes standard LTL operators over finite traces. These operators express temporal relations such as precedence or eventuality. For example,

$$\mathsf{G}(\texttt{search} \rightarrow \mathsf{F}\,\texttt{cite})$$

states that every `search` call must eventually be followed by a `cite` call.

Data quantifiers. Quantifiers range over finite domains extracted from τ or μ, such as tool inputs, generated files, or retrieved entities. To evaluate a formula containing a quantifier, we use eager instantiation: given a domain extractor $D(\tau) = \{e_1, \ldots, e_n\}$, the formula $\forall x \in D.\, \varphi(x)$ is evaluated by substituting each entity into φ and checking the resulting formulas independently. Quantifiers may nest and rebind, allowing constraints such as "every file read is eventually written back."

3.3 Compliance Scoring

A procedure P is a weighted set of constraints $G_P = \{(\kappa_i, w_i)\}$, where each κ_i is an AGENTLTL constraint and $w_i > 0$ denotes its importance. A pair (κ_i, w_i) is *violated* when τ does not satisfy κ_i.

Definition 1 (Compliance score)**.** Let $V \subseteq G_P$ be the set of violated constraints. The compliance

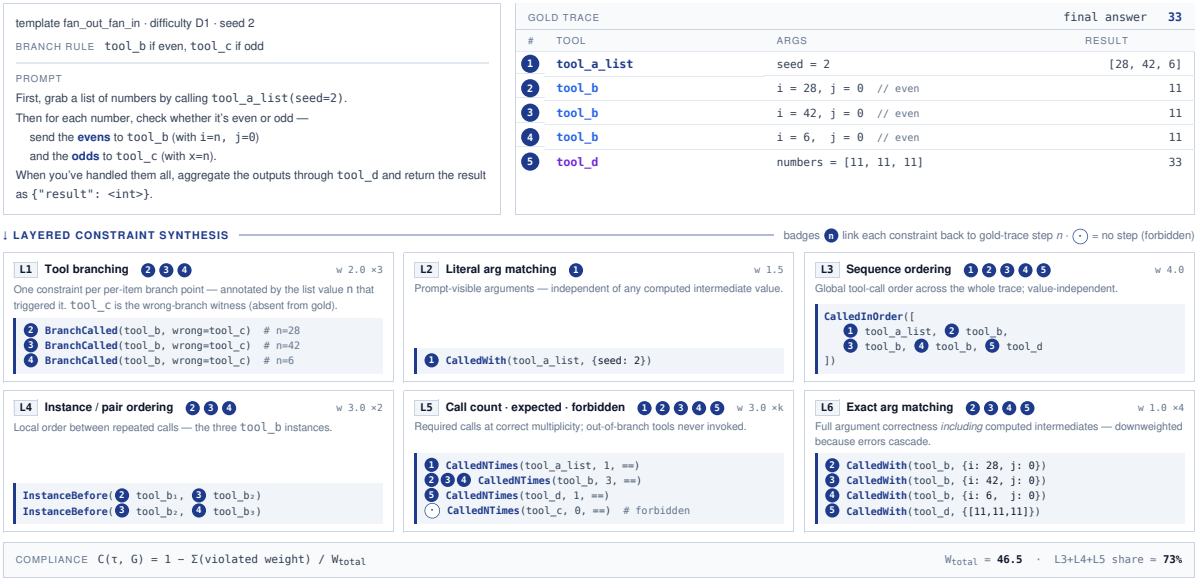

Figure 2: **One process instance, shown in three parts. (Top)** A `fan_out_fan_in` task (D1, seed 2): `tool_a_list` produces a list that is fanned out to a per-item branch (`tool_b` on even, `tool_c` on odd) and fanned in through `tool_d`. Gold trace shown at right. **(Middle)** The six AGENTLTL layers synthesized from the gold trace; each card shows weight, constrained steps (numbered badges), constraints, and constraint numbers per layer. `tool_c` appears only as a forbidden-tool witness. **(Bottom)** Weighted satisfied fraction; W_{total}=46.5, structural layers carry ≈73%.

score is

$$C(\tau, G_P) = 1 - \frac{\sum_{(\kappa_i, w_i) \in V} w_i}{\sum_{(\kappa_i, w_i) \in G_P} w_i} \in [0, 1].$$

Constraint design. The weights w_i are not chosen by the formalism and strongly affects the expressiveness of the score. Indeed, uniform weighting can make the metric brittle: a single arithmetic or reasoning error can invalidate all dependent checks, collapsing the score even when most behavioral constraints are still satisfied. To prevent this, we organize constraints into six layers (L1–L6) of abstraction, each capturing a different failure mode. This separation isolates routing, ordering, and computation errors so they contribute independently to the score.

L1 Tool branching: routing decisions and forbidden branches.

L2 Literal arg match: arguments directly derived from the prompt.

L3 Sequence ordering: global order of tool calls.

L4 Pair/instance ordering: local order between repeated or paired calls.

L5 Call counts: required and forbidden call frequencies.

L6 Exact arg match: full arguments, including computed values from earlier outputs.

3.4 Using The Compliance Score

We introduce to usage of our compliance score. First, harnessing: the same procedure can score a trace offline or block tool calls online. Second, fine-tuning: because the score is continuous, it serves as a training reward. A *harness* is a runtime enforcement layer around an agent that monitors behavior and can intercept or regulate tool calls before execution (Wang et al., 2026; Yuan et al., 2026). Existing harnesses use hand-written rules tied to a specific runtime. Ours is built directly from an AGENTLTL procedure.

Harness implementation. Given a trace and a procedure, the score measures how closely the agent followed the procedure. *Offline mode* evaluates a completed trace. The full constraint set G_P is checked against τ, and the framework reports both the overall score and per-constraint results. This supports benchmarking and post-hoc analysis. *Online mode* evaluates a prefix trace $\tau^{\leq i} = \langle c_0, \ldots, c_i \rangle$ before each tool execution. A proposed call is appended to the prefix and checked against the enforceable subset of G_P. The call executes only if no enforceable constraint is definitively violated.

Some constraints cannot be decided from prefixes. For example, a constraint that requires an event to eventually occur cannot be refuted until execution

ends. Such constraints are deferred to offline evaluation. Each enforceable constraint has a severity level that determines the response, from logging to corrective feedback or blocking execution.

Training reward implementation. Because $C(\tau, G_P) \in [0, 1]$ is continuous, we can use it directly as a reward. The decomposition over constraints provides dense credit assignment: traces receive partial reward even when only some constraints are satisfied. Layered weighting further separates structural and computational failures, so traces with different error profiles receive different rewards rather than collapsing to the same score.

4 Experimental Settings

We first describe how our data is generated, then evaluate model compliance under AGENTLTL enforcement, finetune on the compliance signal, and finally apply the framework to repository-QA.

4.1 Data Generation and Constraint Design

Dataset. A *template* is a named procedural pattern that fixes a procedure and a tool set. We use twelve templates grouped into five structural types. Each template has a *production* mode for benchmarking and a *simplified* mode for training. The two modes share the same procedural pattern but differ in prompt length, parameter ranges, and tool-set size. An *instance* is one (template, difficulty, seed) tuple, which deterministically yields a prompt, a gold trace, an expected answer, and a constraint set. Difficulty (D1–D5) scales one structural parameter of the template, such as loop bound or list length.

Generation. For each instance, a Python function executes the procedure on deterministic arithmetic tools and emits the gold trace and expected answer. Tools have generic names such as `tool_a` and `tool_b` but execute fixed integer operations. Every correct answer is derivable from the trace, so failures reflect procedure compliance rather than arithmetic knowledge. The procedures mirror common agent patterns such as paginated fetching, per-record validation, and graph traversal.

Constraints. From each gold trace we instantiate the six constraint layers described in Section 3.3, yielding one constraint set G_P per task. Structural layers (L3, L4, L5) carry roughly 73% of total weight; L6 is downweighted to prevent a single early arithmetic error from cascading into a near-zero score. Figure 2 shows the constraints generated for one such trace.

4.2 Harnessing Agents

We evaluate twelve templates at five difficulty levels, with gold traces ranging from 2 to 50 required tool calls, and report compliance per layer, producing a per-trace profile that separates routing errors from argument errors and skipped branches.

Settings. We compare three settings that share the same constraints. *Baseline* runs the agent to completion and scores offline. *Block-and-warn* uses online enforcement: a violating call is blocked with a warning, but the same call reissued with the same arguments is allowed through. *Soft-block* also uses online enforcement, escalates after three violations of the same constraint, and terminates the run. The three settings isolate, respectively, unaided compliance, responsiveness to redirection, and behavior under deployed guardrails.

Setup. We evaluate 7 zero-shot models of varying parameter sizes (Gemma-4-26B-A4B, Gemma-4-31B, Qwen3-Next-80B-A3B-Instruct, DeepSeek-V4-Flash, Qwen3.5-397B-A17B, DeepSeek-V4-Pro, Kimi-K2.6) via AGENTLTL's `smolagents` (Roucher et al., 2025) integration. We run six trials per (model, template, difficulty, seed) across seeds $s \in \{42, 123\}$, giving 360 runs per model per setting, with a budget of $10d + 10$ tool calls at difficulty d. Answer correctness is exact integer match; compliance is $C(\tau, G_P)$.

4.3 Compliance-Supervised Finetuning

Training corpus. We build 300 examples from 15 simplified templates, each using 3–7 tool calls from a pool of four tools and covering sequential, branching, fan-out, and multi-stage patterns. We instantiate each template with multiple tool permutations and augment instances by optionally prepending or appending one extra tool call. This is to discourage memorization of fixed call sequences. Two benchmark templates, `loop_termination` and `branch_selection`, are excluded from training and used only for unseen-pattern evaluation.

Reward. We train with Group Relative Policy Optimization (GRPO) using `trl` (von Werra et al., 2020) under a weighted reward [0.5,0.25,0.25] combining three terms: the main term is $C(\tau, G_P)$; answer correctness, where correctly formatted but incorrect answers receive 0.1 to preserve gradient

signal; and a trace-distance term $R_{\text{dist}}(\tau, \tau^*) = \exp(-d(\tau, \tau^*))$ based on weighted Levenshtein distance to the gold trace (same tool with different arguments costs 0.5, a different tool costs 1.0, and insertions and deletions cost 1.0).

Training setup. We finetune Qwen3-4B-Instruct (Team, 2025) with LoRA (r=8, α=16) on `q_proj` and `v_proj` for 1 epoch, using a round-robin-no-repeat curriculum, fp32 precision, N=16 rollouts per prompt, and β=10^{-3}.

Evaluation splits. We evaluate on four held-out validation splits and the full production benchmark. *Unaugmented* (83 instances) re-instantiates the 15 training templates with new parameter values. *Augmented* (100 instances) adds the optional prefix/-suffix calls from training. *Unseen-pattern* (39 instances) uses the two excluded benchmark templates with the training tool set. *Diverse-tool* (39 instances) keeps trained patterns but swaps in unseen tool names. The production benchmark (360 instances) is partitioned into three transfer groups: *direct-simplification* (shared high-level flow with a training simplification), *motif-overlap* (recombined motifs in new configurations), and *no-coverage* (no structural counterpart in training).

4.4 Trace-Grounding Analysis

Grounding predicate. For a trace τ with final answer a, let $\text{ent}(\cdot)$ be an open predicate returning the referential tokens (identifiers, file paths, numeric literals) in its argument, and let $\text{out}(\tau)$ denote the entities observed in tool outputs. We implement $\text{ent}(\cdot)$ via regex matching and $\text{out}(\cdot)$ by parsing tool outputs into an abstract syntax tree. The grounding predicate

$$\kappa_{\text{ground}} \equiv \forall e \in \text{ent}(a),\ e \in \text{out}(\tau)$$

holds when every entity mentioned in a was observed in some tool output. Correctness is scored separately from grounding. An answer is correct if it contains all ground-truth entities defined for the question (function signatures, file paths, identifiers).

Setup. We evaluate on 160 (repo, questions) pairs on 16 Python repositories of varying size and popularity. For each repository, we generate 10 questions using `ClaudeOpus4.7` asking for the location of a public function or a class definition. We test `Qwen3.5-9B`, `Qwen3.5-397B-A17B`, and `Gemma-4-31B-IT` with filesystem tools (`list_directory`,`read_file`). We compare a default prompt against a strict-grounding prompt that instructs the model to answer only from retrieved evidence or refuse. Across two seeds per configuration, we collect 1,917 valid traces.

Evaluation. To validate κ_{ground} as a hallucination signal, we compare it against a trace-aware LLM judge (`DeepSeek-V4-Pro`) applied post-hoc. The judge receives the question, the full tool trace, and the answer, and is asked whether every factual claim in the answer is supported by the trace. It makes five independent calls per trace and returns a mean boolean decision. The judge does not drive the primary evaluation; it exists to test whether κ_{ground} verdicts correlate with LLM judgment.

5 Results

5.1 Harnessing Agents

Enforcement effects. Table 1 shows block-and-warn improves compliance for five of seven models, with the largest gains on the weakest models. It raises accuracy by $+0.019$ on average. Two strong models regress slightly under block-and-warn. Soft-block is the worst setting for four of seven models, since forced termination prevents recovery from violations. The two settings affect trace length in opposite ways: block-and-warn pushes models past early stopping points and lengthens traces, while soft-block shortens them through forced termination. Enforcement events concentrate on argument extraction and global sequence (L2: $61, 111$ and L3:$62, 811$). Branch and count violations are sparse but high-impact. Insistence on blocked calls is rare.

Baseline evaluation and compliance profiles. Table 4 reports per-layer pass-rates. L3 (global sequence) is the weakest layer for every model. L4 (pair-order) tracks L3 closely, so the same trace failures violate both global and pairwise ordering. L4 accounts for 55-77% of weighted loss across models. Branch routing, argument extraction, and call-count pass at much higher rates, so failures concentrate in ordering rather than tool selection or argument formatting. The compliance-correctness gap varies by model. DeepSeek-V4-Pro produces correct answers from non-compliant traces; Qwen3-Next-80B shows the inverse, with high compliance but low accuracy. Composite templates show the lowest L3 pass-rates and the lowest correctness.

Model	Size (B)	Baseline $\bar{C}$	Block-&-warn	Soft-block
`Gemma-4-26B-A4B`	26	0.617	**0.717**	0.623
`Gemma-4-31B`	31	0.789	**0.961**	0.877
`Qwen3-Next-80B-A3B`	80	**0.874**	0.829	0.799
`DeepSeek-V4-Flash`	158	0.596	**0.867**	0.530
`Qwen3.5-397B-A17B`	397	0.909	0.896	**0.937**
`DeepSeek-V4-Pro`	671	0.735	**0.823**	0.739
`Kimi-K2.6`	1000	0.831	**0.891**	0.724

Table 1: AGENTLTL compliance $\bar{C}$ across models and enforcement settings. **Bold** marks each model's best-performance; underline marks its worst.

	Base			Finetuned		
Split (n)	$\bar{C}$	Acc %	$\bar{TC}$	$\bar{C}$	Acc %	$\bar{TC}$
Unaug. (83)	0.564	4.8	5.1	**0.850**	**67.5**	5.0
Aug. (100)	0.619	12.0	6.2	**0.769**	**46.0**	5.6
Unseen (39)	0.628	7.7	6.3	**0.803**	**46.2**	5.8
Div.-tool (39)	0.647	23.1	5.2	**0.772**	**61.5**	5.7
Full (360)	0.562	13.4	50.5	**0.775**	**57.2**	9.1
direct	0.616	30.0	36.5	**0.878**	**73.3**	8.4
overlap	0.525	4.2	64.0	**0.724**	**49.5**	10.0
no-cov.	0.606	13.3	10.5	**0.724**	**46.7**	4.9

Table 2: Zero-shot base vs. finetuned model validation results. $\bar{TC}$: average number of tool calls.

Model	Prompt	CG %	CU %	IG %	IU %
Qwen3.5-9B	Default	3.4	7.5	48.4	40.6
	Strict	4.1	3.8	50.6	41.6
Qwen3.5-397B	Default	0.0	49.7	0.0	50.3
	Strict	0.3	59.4	3.4	36.9
Gemma-4-31B	Default	0.0	48.8	1.6	49.7
	Strict	3.4	44.7	9.7	42.2

Table 3: Trace-grounding experiment: joint distribution of answer correctness and trace grounding by model and prompt condition. CG: correct & grounded; CU: correct & ungrounded; IG: incorrect & grounded; IU: incorrect & ungrounded.

5.2 Compliance-Supervised Finetuning

Training details. Over the 4,800 GRPO generations, training-time evaluation rewards improve jointly: AGENTLTL compliance from 0.511 to 0.770, answer correctness from 0.296 to 0.728, and gold-trace distance reward from 0.301 to 0.782.

Held-out validation. Table 2 compares the finetuned and base models. Answer correctness more than triples across all four splits with parallel compliance gains. The unseen-pattern and diverse-tool splits yield matched accuracy gains of 38.5 percentage points, consistent with the reward inducing structural rather than surface skill. Trace lengths on the held-out splits are similar before and after finetuning, so the accuracy gains come from better-ordered execution rather than longer execution.

Benchmark transfer. The full benchmark gains 43.8 and 21.3 percentage points in accuracy and compliance under finetuning. Trace lengths reveal the mechanism. The base model issues many tool calls but converts few of them into compliant or correct execution. The finetuned model uses far fewer calls and reaches much higher accuracy and compliance. Gains are consistent across all three transfer groups. The no-coverage gain is evidence for transferable procedural skill rather than memorization, since these templates share no structural counterpart in training. The one remaining failure is `nested_loops+branching`, where deeply nested control flow is not recovered by training.

5.3 Trace-Grounding Analysis

Correctness, grounding, and prompt effects. Table 3 shows how correctness and grounding combine. Two patterns stand out. First, the larger models (Qwen3.5-397B, Gemma-4-31B) give mostly correct but ungrounded answers, while `Qwen3.5-9B` grounds more of its answers but gets them wrong. Second, the strict prompt raises grounding for all models, but the extra grounded traces are mostly refusals, not correct answers: IG grows more than CG for Qwen3.5-397B and Gemma-4-31B. Splitting by repository popularity makes this clearer. On low-popularity repositories, CG rises from 0.7% to 4.0% under the strict prompt; on high-popularity ones, only from 1.2% to 1.7%. Correctness under the default prompt drops from 52.0% to 24.0% as popularity falls, and the most common category shifts from CU to IU. CG stays at or below 4.0% in every bucket. This fits the idea that the models recall less as repositories appear less often in pretraining. More detailed results in Appendix H.

Comparison with an LLM judge. κ_{ground} and the trace-aware judge agree on 77.6% of traces, but Cohen's κ = 0.09 reflects class imbalance. The judge assigns grounded verdicts to 6.4% of answers versus 20.9% for κ_{ground}. The judge therefore treats κ_{ground} as a necessary but not sufficient condition for grounding.

Model	L1 Branch	L2 Args	L3 Sequence	L4 Pair-order	L5 Count	L6 Exact-args	$\bar{C}$ Compliance	Acc. Correctness
`Qwen3.5-397B-A17B`	0.998	**1.000**	0.622	0.762	0.983	0.996	0.909	0.989
`Qwen3-Next-80B`	0.862	**0.992**	0.686	0.805	0.889	0.944	0.874	0.433
`Kimi-K2.6`	0.986	**0.988**	0.450	0.603	0.932	0.957	0.831	0.828
`Gemma-4-31B`	**1.000**	0.990	0.292	0.489	0.943	0.948	0.789	0.508
`DeepSeek-V4-Pro`	0.946	**0.981**	0.153	0.312	0.950	0.961	0.735	0.944
`Gemma-4-26B-A4B`	0.869	**0.939**	0.111	0.144	0.872	0.830	0.617	0.325
`DeepSeek-V4-Flash`	0.652	0.741	0.119	0.249	**0.864**	0.681	0.596	0.556

Table 4: **Analysis of baseline setting, per-layer pass-rates across seven models.** Fraction of constraint checks satisfied at the corresponding AGENTLTL layer, aggregated over all templates and difficulties. The last two columns give the model's mean compliance score $\bar{C}$ and answer correctness rate (Acc.). **Bold** marks each model's best layer; underline marks its worst.

6 Discussion

Compliance and correctness diverge. The two metrics correlate overall. More specifically, `nested_loops` is high compliance with low correctness; the large models invert this with correct-but-ungrounded traces. Final-answer accuracy hides both patterns. Previous qualitative evaluations such as MAST document this divergence but cannot quantify it without a judge.

Enforcement settings act as diagnostics. Block-and-warn and soft-block share the same constraints but differ in whether retries are free. Their gap measures how much non-compliance is locally recoverable. Gains under block-and-warn but not soft-block indicate corrigible mistakes rather than absent capability, a pattern concentrated in weaker models. Stronger models converge across settings as first-attempt violations become rare. Soft-block also acts as a deployment proxy: a large drop under it means the model's baseline score depended on recovering from its own mistakes later in the trace.

Sequence failures share one mechanism. The hardest composite templates map to common agent workloads: `fan_out+loop` and `forall_processing` to batch RAG, `tool_condition_branching` to routing, and `nested_loops+branching` to long-horizon planning. TravelPlanner reports similar failures in single-agent planning. Many L3/L4 violations and correct-but-ungrounded traces appear to stem from the same behavior: when faithful execution becomes costly, models shortcut to parametric memory. This appears either as batching sequential calls into parallel steps or skipping retrieval entirely. The repository-popularity gradient supports this interpretation.

Compliance and accuracy rise together. The no-coverage accuracy gain is the strongest evidence for transferable procedural skill rather than memorization. The full-benchmark compliance gain also resolves a measurement artifact: under step-boundary ordering, the base model's long traces violate sequence and pair-order constraints repeatedly as it cycles through tools without progressing. The finetuned model traverses the procedure in order. On the held-out splits, where base trace lengths are already close to gold, compliance and accuracy rise together without a length change.

7 Conclusion

In this paper, we introduced AGENTLTL, a framework that treats procedural compliance as an explicit evaluation axis for tool-using LLM agents. Our unified FO-LTL specification language supports evaluation, runtime enforcement, and finetuning within the same formal framework. AGENTLTL improves both accuracy and compliance on the proposed benchmark after finetuning. These gains also transfer to the no-coverage split, where templates are unseen during training, suggesting that the reward captures reusable procedural structure rather than template memorization. Our benchmark reveals failure modes that are not reflected by final-answer accuracy alone, including correct-but-ungrounded traces, premature termination, and unproductive loops. Most violations are concentrated in tool ordering and pairwise sequencing constraints. The harness also helps weaker models recover from early-stopping failures without reducing correctness. Overall, these results provide a quantitative view of procedural failures in tool-using agents and suggest that formal specifications can support more reliable agent behavior.

8 Limitations

Several limitations qualify our findings and point to future work.

Vacuous grounding. The trace-grounding predicate κ_{ground} is satisfied trivially by refusals, since the universal quantifier holds when the answer contains no entities to verify. Most of the strict-prompt grounding gains for the larger models in our repository-QA experiment are refusals rather than supported answers.

Confounded popularity. The popularity stratification in the grounding study confounds pretraining exposure with repository size and complexity. We cannot separate the two with this design. A cleaner causal claim would require controlled splits matching size across popularity levels, or held-out private repositories with no pretraining footprint.

Synthetic tool environment. The benchmark uses synthetic arithmetic tools to keep ground truth deterministic and decouple procedural compliance from world knowledge. The procedural patterns are drawn from common agent workflows, and the failure modes we isolate match those reported qualitatively in prior work on real-world agents. Whether they transfer quantitatively to noisier real-world environments, with stochastic tool outputs, partial failures, and ambiguous arguments, remains an open question.

Model finetuning. Our finetuning experiments use a single base model (`Qwen3-4B-Instruct`), because of computational limitations, with LoRA adapters. The transfer results on unseen patterns and unseen tools are suggestive of structural learning, but have not been replicated across scales or model families. It is unclear whether the compliance reward remains effective at larger scales where parametric recall is stronger and the shortcut behavior we target is harder to dislodge.

Constraints authoring cost. AGENTLTL shifts the cost of evaluation from labeling answers to writing FO-LTL constraints. For benchmarks generated from gold traces this cost is amortized, but in deployment the constraints must be authored by domain experts. We do not address the human factors of constraints authoring, the failure modes of misspecified procedures, or how errors in constraints propagate into compliance scores and training signals.

Constraint coverage and expressiveness. Our six-layer constraint schema covers the structural failure modes we designed for, but it does not exhaust the space of procedural errors. Constraints on timing, resource consumption, privacy, and inter-agent coordination are outside the current schema. FO-LTL also cannot express hyperproperties, such as non-interference between parallel tool calls, which may matter in multi-agent settings.

Enforcement side effects. The block-and-warn and soft-block results show that enforcement is not uniformly beneficial. Soft-block is the worst setting for most models, since forced termination prevents recovery from violations that block-and-warn would only flag. Block-and-warn helps most models on both compliance and accuracy, but perturbs two of the strongest models that were already near their compliance ceiling. We do not yet have a principled account of when enforcement helps versus hurts, and the right enforcement policy likely depends on model capability and task structure in ways we have not characterized. A deeper issue is that in critical or irreversible workflows, simply stopping execution on a violation may itself cause harm. Blocking a tool call that has already triggered a side effect, or terminating a trace mid-procedure in a multi-step transaction, can leave the system in an inconsistent state. The appropriate response to a violation, whether to block, warn, roll back, or escalate to a human, is use-case dependent and requires richer enforcement semantics than the two settings we evaluate here.

Benchmark difficulty. The `nested_loops+branching` template remains at 0.0% correctness after finetuning. Deeply nested control flow is not recovered by training on simpler motifs, and the compliance reward provides no gradient signal when the model cannot produce even a partial trace. More structured curricula, or decomposed constraints that reward partial progress on subprocedures, may be needed for these cases.

Single-turn constraints. Our framework evaluates each trace against a fixed specification written before execution. In practice, procedures may need to adapt mid-execution in response to tool outputs, for example skipping a branch because a preceding call returned an empty result. Dynamic or reactive constraints, where the constraint set is updated as the trace unfolds, are a natural extension that we leave to future work.

9 Ethical Considerations

Use of AI assistance. We used AI assistants during the preparation of this work. Their use falls within the ACL Policy on AI Writing Assistance. Specifically, we used AI tools for (i) language polishing and paraphrasing of author-written content (category a) and (ii) literature search support, where we verified citation accuracy and relevance for all cited works (category c). No AI system contributed research ideas, framing, or analysis that would warrant acknowledgement as a collaborator. The authors take full responsibility for the correctness of all text, methods, results, and code.

Intended use and dual-use considerations. AGENTLTL is designed to evaluate and improve procedural compliance in tool-using LLM agents, with particular relevance to safety-critical settings such as clinical triage, enterprise workflows, and retrieval-grounded question answering. The framework is descriptive: a constraint captures what a procedure requires, but does not validate that the procedure itself is appropriate. A mis-specified or harmful procedure that an agent complies with perfectly is not made safe by a high compliance score. Deployment in any sensitive setting requires domain experts to author and review constraints, and the compliance score should be one signal among several rather than a sufficient certification.

Dataset and benchmark. The benchmark uses synthetic arithmetic tools with no personal, copyrighted, or otherwise sensitive content. The 16 Python repositories in the trace-grounding study are publicly available open-source projects; we use them only for evaluation and do not redistribute their content. The generated questions reference public function and class definitions in those repositories.

Compute and environmental cost. Training Qwen3-4B-Instruct with GRPO required approximately 317 GPU-hours on a single NVIDIA DGX Spark, run continuously from April 29 to May 12, 2026. Evaluation of seven zero-shot models across 360 runs per setting and three harness configurations represents an additional compute cost, performed via API access to hosted models. We report all settings and hyperparameters in Section 4.3 to enable reproduction without redundant recomputation.

Reproducibility. We release the AGENTLTL framework, the benchmark with all twelve templates and difficulty levels, the training corpus, the finetuned model adapters, and the evaluation scripts. All experiments use fixed seeds documented in the codebase.

Risks of procedural enforcement. Our harnessing settings can block tool calls before execution. In real deployments, blocking a tool call that has already triggered a side effect, or terminating a trace mid-procedure, can leave a system in an inconsistent state. We discuss this further in Section 8; in this paper we evaluate enforcement only in synthetic environments where rollback is not required.

References

Yejin Bang, Ziwei Ji, Alan Schelten, Anthony Hartshorn, Tara Fowler, Cheng Zhang, Nicola Cancedda, and Pascale Fung. 2025. HalluLens: LLM hallucination benchmark. *Preprint*, arXiv:2504.17550.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *Preprint*, arXiv:2506.07982.

David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. 2012. Monitoring compliance policies over incomplete and disagreeing logs. In *Proceedings of the 3rd International Conference on Runtime Verification (RV)*, volume 7687 of *Lecture Notes in Computer Science*, pages 151–167. Springer.

Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2011. Two-variable logic on data words. *ACM Transactions on Computational Logic*, 12(4):27:1–27:26.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent llm systems fail? *Preprint*, arXiv:2503.13657.

I-Chun Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, and Pengfei Liu. 2023. Factool: Factuality detection in generative AI — a tool augmented framework for multi-task and multi-domain scenarios. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

William H. English, Chase Walker, Dominic Simon, Sumit Kumar Jha, and Rickard Ewetz. 2025. Verifiable natural language to linear temporal logic translation: A benchmark dataset and evaluation suite. *Preprint*, arXiv:2507.00877.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12).

Finn Klessascheck, Tobias Knoche, and Luise Pufahl. 2024. Reviewing conformance checking uses for runtime regulatory compliance. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS/EMMSAD)*, volume 511 of *Lecture Notes in Business Information Processing*. Springer.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2025. AgentBench: Evaluating LLMs as agents. *Preprint*, arXiv:2308.03688.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. AgentBoard: An analytical evaluation board of multi-turn LLM agents. *Preprint*, arXiv:2401.13178.

Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. 2025. Evaluation and benchmarking of LLM agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pages 6129–6139. ACM.

Nils Palumbo, Sarthak Choudhary, Jihye Choi, Prasad Chalasani, Mihai Christodorescu, and Somesh Jha. 2026. Policy compiler for secure agentic systems. *arXiv preprint arXiv:2602.16708*.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Shishir G. Patil and 1 others. 2025. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *PMLR*.

Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. 2007. Declarative workflows: Balancing between flexibility and support. In *Proceedings of the 2007 IEEE Conference on Enterprise Distributed Object Computing (EDOC)*.

Amir Pnueli. 1977. The temporal logic of programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57.

Keshav Ramani and 1 others. 2025. Bridging LLM planning agents and formal methods: A case study in plan verification. *arXiv preprint arXiv:2510.03469*.

Corby Rosset, Pratyusha Sharma, Andrew Zhao, Miguel Gonzalez-Fernandez, and Ahmed Awadallah. 2026. The art of building verifiers for computer use agents. *Preprint*, arXiv:2604.06240.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. `smolagents`: a smol library to build great agentic systems. `https://github.com/huggingface/smolagents`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Anh Ta, Junjie Zhu, and Shahin Shayandeh. 2026. Reinforced agent: Inference-time feedback for tool-calling agents. In *Proceedings of the Fifth Workshop on Natural Language Generation, Evaluation, and Metrics (GEM) at ACL*. Apple Machine Learning Research.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Wil M. P. van der Aalst. 2016. *Process Mining: Data Science in Action*, 2 edition. Springer.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformers Reinforcement Learning.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. 2026. AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*.

Cailin Winston, Claris Winston, and René Just. 2026. Solver-aided verification of policy compliance in tool-augmented LLM agents. *arXiv preprint arXiv:2603.20449*.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024a. Travelplanner: A benchmark for real-world planning with language agents. *Preprint*, arXiv:2402.01622.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024b. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Preprint*, arXiv:2404.07972.

Weizhe Xu, Mengyu Liu, and Fanxin Kong. 2026. SafePilot: A framework for assuring LLM-enabled cyber-physical systems. *Preprint*, arXiv:2603.21523.

Ziyi Yang, Shreyas S. Raman, Ankit Shah, and Stefanie Tellex. 2024. Plug in the safety chip: Enforcing constraints for LLM-driven robot agents. In *Proceedings of the AAAI Symposium on Human-Robot Interaction Safety and Security (AI-HRI)*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *Preprint*, arXiv:2406.12045.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Aojie Yuan, Zhiyuan Su, and Yue Zhao. 2026. AEGIS: No tool call left unchecked — a pre-execution firewall and audit layer for AI agents. *arXiv preprint arXiv:2603.12621*.

A Appendix A — The AGENTLTL Framework (Formal Details)

This appendix provides the complete formal specification of the AGENTLTL framework, complementing Section 3. It formalizes the trace model, specification language, constraint evaluation, and online enforcement. Throughout, we use τ for traces (resolving to τ in display) to match the notation of Section 3. All positions are 0-indexed.

A.1 Trace and Execution Model

A.1.1 Trace

A trace is a finite sequence

$$\tau = \langle c_0, \ldots, c_{n-1} \rangle,$$

where each call is the 4-tuple

$$c_i = (n_i,\, a_i,\, r_i,\, i),$$

with $n_i \in \mathcal{T}$ the tool name, a_i : Key $\to$ Value the argument mapping, $r_i \in$ Value the (possibly structured) result, and i the position. We write $c_i.\mathsf{name}, c_i.\mathsf{args}, c_i.\mathsf{result}, c_i.\mathsf{position}$ for the four projections. The position induces a total order: $i < j \iff c_i$ precedes c_j. When a trace τ satisfies a formula φ, we $\tau \models \varphi$, and $\tau \not\models \varphi$ when it does not.

A.1.2 Tool Call Schema

Each call is derived from a JSON record of the form:

```
{
  "tool_name": "...",
  "arguments": {...},
  "tool_result": "..."
}
```

and normalized into the tuple above. Arguments are structured mappings; results may be structured or unstructured and are treated opaquely except through the comparator match (defined below).

A.1.3 Metrics Record

A metrics record accompanies the trace:

$$\mu = (\mathsf{tool_calls},\ \mathsf{num_steps},\ \mathsf{num_tool_calls},\ \mathsf{token_counts},\ \ldots).$$

Only tool_calls is required for formal semantics; other fields are auxiliary. See Listing 1 for a representative instance.

Evaluation on recorded traces. Evaluation is performed on the recorded trace; the framework does not require re-execution. The trace τ and metrics μ are fixed and fully observable during evaluation, so the semantics below are deterministic regardless of any nondeterminism in the underlying agent or tools.

A.1.4 Auxiliary Functions

The following helpers are used throughout the predicate definitions. All are 0-indexed. We write $\bot$ for an undefined value; predicates that depend on a $\bot$ operand evaluate to `false`.

- $\mathsf{first}(t) = \min\{i : c_i.\mathsf{name} = t\}$, or $\bot$ if t never occurs.
- $\mathsf{nth}(t, k)$ = position of the $(k{+}1)$-th occurrence of t (0-based: $\mathsf{nth}(t, 0) = \mathsf{first}(t)$), or $\bot$ if fewer than $k{+}1$ occurrences exist.
- $\mathsf{first_after}(t, i) = \min\{j \geq i : c_j.\mathsf{name} = t\}$, or $\bot$ if no such j.
- $\mathsf{match}(c.\mathsf{result}, r)$: a user-pluggable comparator over results; the default is structural equality.

A.2 FO-LTL Specification Language

A.2.1 Syntax

$$
\begin{aligned}
\varphi ::= {} & \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
& \mid \mathsf{X}\varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \\
& \mid \varphi \,\mathsf{U}\, \varphi \mid \varphi \,\mathsf{W}\, \varphi \mid \varphi \,\mathsf{R}\, \varphi \\
& \mid \forall x \in D.\ \varphi \mid \exists x \in D.\ \varphi
\end{aligned}
$$

A.2.2 Atomic Predicates

Atomic predicates fall into five categories: occurrence (Called), ordering (Before, After, InOrder), argument checks (CalledWith), result checks (CalledWithResult), and counts (CalledN). All predicates are evaluated globally over τ.

For argument predicates, we write $\mathbf{a} \sqsubseteq c_i.\mathsf{args}$ to mean that every key–value pair in $\mathbf{a}$ is present in $c_i.\mathsf{args}$ (i.e., $\mathbf{a}$ is a sub-mapping of $c_i.\mathsf{args}$).

- $\mathsf{Called}(t)$: $\exists i.\ c_i.\mathsf{name} = t$
- $\mathsf{CalledN}(t, n, \odot)$: $|\{i : c_i.\mathsf{name} = t\}| \odot n$, $\odot \in \{=, \geq, \leq, >, <\}$
- $\mathsf{Before}(t_1, t_2)$: $\mathsf{first}(t_1) < \mathsf{first}(t_2)$ (false if either side is $\bot$).
- $\mathsf{After}(t_1, t_2)$: $\mathsf{first}(t_1) > \mathsf{first}(t_2)$ (false if either side is $\bot$).
- $\mathsf{AllBefore}(\mathbf{T}, g)$: $\forall t \in \mathbf{T}.\ \mathsf{first}(t) < \mathsf{first}(g)$.
- $\mathsf{InstanceBefore}(t_1, n, t_2, m)$: $\mathsf{nth}(t_1, n) < \mathsf{nth}(t_2, m)$ (false if either is $\bot$).
- $\mathsf{InOrder}(\mathbf{T})$: for $\mathbf{T} = \langle t_1, \ldots, t_k \rangle$, there exist indices $i_1 < i_2 < \cdots < i_k$ with $c_{i_j}.\mathsf{name} = t_j$ for all j.
- $\mathsf{WithinSteps}(t_1, t_2, k)$: $\mathsf{first_after}(t_2, \mathsf{first}(t_1)) - \mathsf{first}(t_1) \leq k$.
- $\mathsf{CalledWith}(t, \mathbf{a})$: $\exists i.\ c_i.\mathsf{name} = t \ \wedge\ \mathbf{a} \sqsubseteq c_i.\mathsf{args}$
- $\mathsf{CalledWithResult}(t, r, \mathbf{a})$: $\exists i.\ c_i.\mathsf{name} = t \ \wedge\ \mathbf{a} \sqsubseteq c_i.\mathsf{args} \ \wedge\ \mathsf{match}(c_i.\mathsf{result}, r)$
- $\mathsf{BranchCalled}(t^+, t^-)$: $\mathsf{Called}(t^+) \ \wedge\ \neg\mathsf{Called}(t^-)$. The prescribed branch t^+ is taken and the forbidden branch t^- is not.
- $\mathsf{Predicate}(f)$: $f(\tau, i, \mu) \in \mathbb{B}$, a user-defined open predicate.

A.2.3 Temporal Operators

AGENTLTL supports the six standard LTL operators: globally (G), finally (F), next (X), until (U), weak until (W), and release (R). Let $\tau^{\geq i}$ denote the suffix of τ from position i.

- $\tau^{\geq i} \models \mathsf{G}(\varphi)$ iff $\forall j \geq i.\ \tau^{\geq j} \models \varphi$ (vacuously true when $i \geq n$).
- $\tau^{\geq i} \models \mathsf{F}(\varphi)$ iff $\exists j \geq i.\ \tau^{\geq j} \models \varphi$.
- $\tau^{\geq i} \models \mathsf{X}(\varphi)$ iff $i + 1 < n \ \wedge\ \tau^{\geq i+1} \models \varphi$. This is the *strong* next on finite traces (false at trace end), sometimes written X_w.
- $\tau^{\geq i} \models \varphi \,\mathsf{U}\, \psi$ iff $\exists j \geq i.\big(\tau^{\geq j} \models \psi \ \wedge\ \forall i \leq k < j.\ \tau^{\geq k} \models \varphi\big)$.
- $\tau^{\geq i} \models \varphi \,\mathsf{W}\, \psi$ iff $(\varphi \,\mathsf{U}\, \psi)$ or $\forall j \geq i.\ \tau^{\geq j} \models \varphi$.
- $\tau^{\geq i} \models \varphi \,\mathsf{R}\, \psi$ iff $\forall j \geq i.\big(\tau^{\geq j} \models \psi \ \vee\ \exists i \leq k < j.\ \tau^{\geq k} \models \varphi\big)$.

A.2.4 Data Quantifiers and Open Predicates

A domain extractor $D : (\tau, \mu) \to \mathsf{List}$ supports quantification. Evaluation proceeds by computing $D(\tau, \mu) = \{e_1, \ldots, e_k\}$, substituting each e_i into φ, and evaluating all grounded formulas $\varphi[e_i/x]$. For example, "every number produced is processed" becomes $\forall x \in D_{\text{numbers}}.\ \mathsf{CalledWith}(\texttt{tool_b}, \{i : x\})$. When $D(\tau, \mu) = \emptyset$, the universal $\forall x \in D.\ \varphi$ is vacuously true and the existential $\exists x \in D.\ \varphi$ is vacuously false. User-defined open predicates take the form $f : (\tau, i, \mu) \to \mathbb{B}$.

A.3 Constraint Evaluation Procedure

A.3.1 Evaluation Algorithm

Algorithm 1 shows representative cases of the recursive evaluator; the omitted cases ($\bot, \vee, \mathsf{G}, \mathsf{U}, \mathsf{W}, \mathsf{R}, \exists$) follow analogously from the semantics above. Quantifiers are expanded eagerly; inner scopes shadow outer bindings.

Algorithm 1 Recursive evaluation of φ at position i in trace τ (representative cases).

function $\textsc{Eval}(\varphi, \tau, i)$
  match φ:
    case atomic p: **return** evaluate p over τ
    case $\neg\psi$: **return** $\neg\,\textsc{Eval}(\psi, \tau, i)$
    case $\psi_1 \wedge \psi_2$: **return** $\textsc{Eval}(\psi_1, \tau, i) \wedge \textsc{Eval}(\psi_2, \tau, i)$
    case $\mathsf{X}\,\psi$: **return** $i{+}1 < |\tau| \wedge \textsc{Eval}(\psi, \tau, i{+}1)$
    case $\mathsf{F}\,\psi$: **return** $\exists\, j \geq i : \textsc{Eval}(\psi, \tau, j)$
    case $\forall x \in D$: **return** $\bigwedge_{e \in D(\tau,\mu)} \textsc{Eval}(\psi[x{\mapsto}e], \tau, i)$
end function

A.3.2 Worked Example

For $\varphi = \mathsf{Called}(\texttt{tool_a}) \wedge \mathsf{F}(\mathsf{Called}(\texttt{tool_d}))$, evaluation (i) checks $\mathsf{Called}(\texttt{tool_a})$, (ii) scans suffixes for $\mathsf{Called}(\texttt{tool_d})$, and (iii) combines via $\wedge$.

A.4 Layered Constraint System (L1–L6)

The construction algorithm emits constraints in six layers, listed in Table 5. The layered structure is what Section 3 refers to throughout, and the compliance scoring of Definition 1 aggregates per-layer pass/fail outcomes under layer weights.

Rationale. Higher weights prioritize structural correctness; lower weights capture value precision. Changes in the weight values primarily affect value-level penalties; structural violations dominate overall scores.

A.5 Online Enforcement Mechanism

A.5.1 Prefix Evaluation and Enforceability

Online enforcement operates on the prefix $\tau^{\leq i} = \langle c_0, \ldots, c_i \rangle$. A constraint κ is *enforceable* if some finite prefix witnesses its violation regardless of how execution continues:

$$\exists \tau_{\text{pre}}.\ \forall \tau_{\text{ext}}.\ \tau_{\text{pre}} \cdot \tau_{\text{ext}} \not\models \kappa,$$

where $\cdot$ denotes trace concatenation. Equivalently, κ admits a *bad prefix* in the sense of runtime verification: once such a prefix has been observed, no continuation can satisfy κ, so the violation is decided.

Liveness is deferred. Pure liveness constraints have no bad prefix on finite traces: a formula such as $\mathsf{F}\,\varphi$ can always be satisfied by an unseen future call, so its violation cannot be witnessed until the trace ends. Such constraints are not enforceable online and are deferred to offline evaluation. Safety constraints (e.g., $\mathsf{G}\,\neg\varphi$, $\mathsf{BranchCalled}$) and bounded constraints (e.g., $\mathsf{WithinSteps}$ once its window has elapsed) are enforceable.

A.5.2 Runtime Algorithm

```
function Enforce(call, ):
    ' =  + call
    for  in enforceable_constraints:
        if violated(, '):
            return action(severity())
    return ALLOW
```

A.5.3 Enforcement Settings and Escalation

Each constraint carries a severity tag drawn from four levels. Three are active in our experiments; the fourth, `HARD_STOP`, is defined for completeness but unused in the configurations reported in the main paper.

`HARD_STOP`. The offending tool call is not executed and the run halts immediately. A `ConstraintViolationError` with `violation_type="HARD_STOP"` is recorded on the step.

`SOFT_BLOCK`. The offending tool call is not executed; the agent receives a structured `ConstraintViolationError` observation and may self-correct. Escalates to `HARD_STOP` after a configurable number of blocked attempts (controlled by `SoftBlockMode`).

`BLOCK_AND_WARN`. The offending tool call is not executed; the agent receives a warning observation and may self-correct. Unlike `SOFT_BLOCK`, it never escalates to `HARD_STOP`. If the model's very next tool call (in the immediately following generation) is byte-identical—same tool name and same arguments—the call is treated as a deliberate override and is executed. Any non-identical retry is blocked-and-warned again, with the insistence pointer updated to the new blocked call.

`TOLERATE`. A warning is logged and execution continues.

This design balances strict enforcement of safety-critical constraints with flexibility for recoverable errors.

Layer	Constraint type	Predicate / failure mode	Weight
L1	Tool routing / branching	$\mathsf{BranchCalled}(t^+, t^-)$; prescribed branch taken, forbidden one not	2.0
L2	Prompt-derived arguments	$\mathsf{CalledWith}(t, a)$; arguments derivable from prompt	1.5
L3	Global sequence ordering	$\mathsf{InOrder}(\mathbf{T})$; full sequence as subsequence of τ	4.0
L4	Pairwise / instance order	$\mathsf{Before}(t_1, t_2)$; local order between repeated/paired calls	3.0
L5	Call counts	$\mathsf{CalledN}(t, n, =)$; required count ($= n$) or forbidden ($= 0$)	3.0
L6	Exact argument match	$c_i.\mathsf{args} = a$; full arguments incl. computed intermediates	1.0

Table 5: The six constraint layers, their predicates, and the per-layer weights used inside the compliance score $C(\tau, G_P)$. Structural layers (L3, L4, L5) carry the majority of total weight; the exact-argument layer L6 is downweighted to avoid reward collapse from a single arithmetic error.

B JSON Record Examples

Listings 1 and 2 show the two structured records that the AGENTLTL engine operates on, drawn from an actual run at difficulty 1 (seed 42, expected answer 33).

Metrics record (μ). The metrics dictionary produced by `Agent.run()` carries aggregate counters and the flat `tool_calls` list from which the `Trace` object is constructed. Each entry records the tool name, arguments, and the JSON string returned by the tool.

Compliance output. `verify_trace(`μ, G_P`)` returns the score, categorical label, the flat tool-name sequence, and a per-constraint breakdown. Weights follow the per-layer scheme of Table 5.

C Benchmark and Data Generation

This appendix documents the tool vocabulary, templates, difficulty scaling, instance-generation procedure, and dataset statistics of the data discussed in Section 4.1. It is the underlying data used in the benchmark of Section 4.2 and the training corpus of Section 4.3.

C.1 Tool Definitions

The benchmark uses eight deterministic, stateless tools. All inputs and outputs are integers, and the same input always produces the same output. The names `tool_a`, `tool_b`, ... are intentionally generic so that compliance reflects procedural reasoning rather than tool-name semantics. Table 6 lists each tool's signature, output range, and exact arithmetic definition.

Vocabulary by mode. The production mode draws from all eight tools. The simplified mode used for training restricts the vocabulary to the four-tool pool {`tool_a`, `tool_b`, `tool_c`, `tool_d`}. Both settings share the same procedural patterns; they differ only in vocabulary, prompt phrasing, and parameter ranges (Section C.5).

C.2 Production Benchmark Templates

The production benchmark comprises 12 templates, grouped into the five structural types summarised in Table 8. Table 7 describes each template; tool abbreviations refer to the names in Table 6.

Example trace. For `loop_termination` at D_1 and seed 42:

```
tool_a(i=1) -> 2   [even: +=2 -> 2]
tool_a(i=2) -> 5   [odd:  +=10 -> 12]
... (stop when total >= 10)
expected_answer = 12
```

C.3 Difficulty Scaling

Difficulty levels D_1–D_5 control trace length by varying a single structural parameter per template, never by altering the procedure. The agent never sees the difficulty index — only the resulting numbers embedded in the prompt. Table 9 lists the scaling parameter per template. For `tool_condition_branching`, the scaling parameter is the difficulty index d itself, which is passed as the second argument to `tool_iterations` and thereby sets the loop count.

The scaling axes group naturally: loop-driven templates (`loop_termination`, `loop+expansion`) grow their accumulation threshold; branch-driven templates (`branch_selection`) grow the number of binary decisions; fan-out templates (`fan_*`, `forall_*`, `exists_*`, `forall_exists_combined`) grow list length via `num_items` ($= d{+}2$ for `tool_a_list`); nested-loop templates grow the outer count N; `tool_condition_branching` embeds difficulty inside `tool_iterations`; `dynamic_expansion` grows BFS depth; and `fan_out+loop` grows both list length and inner repetition.

Tool	Signature	Output range	Function
`tool_a`	$(i) \to \text{Int}$	1–7	$(3i+5) \bmod 7+1$
`tool_b`	$(i,j) \to \text{Int}$	0–99	$(ij+11) \bmod 100$
`tool_c`	$(x) \to \text{Int}$	unbounded	$3x+1$
`tool_d`	$(\text{nums}) \to \text{Int}$	unbounded	$\sum \text{nums}$
`tool_a_list`	$(s,d) \to \text{List[Int]}$	each $\in$ 1–50	$[((s(k{+}1) \cdot 7+13) \bmod 50)+1]_{k=0}^{d+1}$
`tool_iterations`	$(s,d) \to \text{Int}$	≥ 3	$(s \bmod 5)+d+2$
`tool_expand`	$(n,s,d) \to \text{List[Int]}$	list	$b{=}\min(d,3)$ children: $(ns+7(k{+}1)+3) \bmod 20d+1$
`tool_value`	$(n) \to \text{Int}$	0–49	$(13n+3) \bmod 50$

Table 6: Tool vocabulary. s denotes the run seed; d the difficulty index. All functions are pure; the stateful tools (`tool_a_list`, `tool_iterations`, `tool_expand`) receive `seed` and `difficulty` at construction time, so per-run state leakage is impossible.

#	Template	Description	Tools
1	`loop_termination`	While-loop: accumulate `tool_a`(i) results (even adds v, odd adds $2v$) until total $\geq$ target.	`a`
2	`branch_selection`	For $k = 0..K{-}1$: call `tool_a`; route to `tool_c` if $v > \theta$ else `tool_b`.	`a,b,c`
3	`fan_out_fan_in`	Get list via `tool_a_list`; per item branch on parity; aggregate with `tool_d`.	`a_list,b,c,d`
4	`forall_processing`	Uniform map: apply `tool_b`(n,j) to every list element; aggregate with `tool_d`.	`a_list,b,d`
5	`exists_selection`	For each n call `tool_b`; if result $\bmod$ modulus $= 0$, then `tool_c` and add, else add w directly.	`a_list,b,c`
6	`nested_loops`	Outer $i = 1..N$: `tool_a`$(i) \to v$; inner $j = 1..v$: `tool_b`(i,j); threshold branch.	`a,b`
7	`tool_condition_branching`	`tool_iterations`(seed) sets N; for i: `tool_b`; route to `tool_c` or `tool_d`.	`iter,b,c,d`
8	`dynamic_expansion`	BFS from seed node up to `max_depth`: `tool_expand` then `tool_value`; accumulate high-value children.	`expand,value`
9	`forall_exists_combined`	Forall `tool_b`; conditional `tool_c` on qualifiers; aggregate `tool_d`.	`a_list,b,c,d`
10	`nested_loops+branching`	Nested loops with inner `tool_b` result routed to `tool_c` or `tool_d`.	`a,b,c,d`
11	`fan_out+loop`	Per list element, run r inner `tool_b`(n,k) calls; sum; even/odd branch; aggregate.	`a_list,b,c,d`
12	`loop+expansion`	While total $<$ target: `tool_a`(i), then expand children via `tool_expand`/`tool_value`.	`a,expand,value`

Table 7: Production benchmark templates.

Resulting trace lengths. Raw trace lengths from the parameter schedule above can in principle reach 200+ calls at D_5 for loop-driven templates. To keep the benchmark tractable and to respect the per-run tool-call budget of $10d+10$, we filter instances to a maximum of 50 tool calls; the longest retained traces are capped accordingly. Approximate post-filter calls per trace are 3–10 at D_1, 5–20 at D_2, 10–30 at D_3, 20–45 at D_4, and up to 50 at D_5.

C.4 Instance Generation Procedure

Each benchmark run is uniquely identified by the triplet $(\text{template}, \text{seed}, \text{difficulty})$. Given this triplet, the generator (i) looks up the parameter dict for the template at that difficulty; (ii) inlines all concrete numbers into a natural-language procedure to produce the prompt; (iii) simulates the procedure deterministically on the arithmetic tools to obtain `expected_answer`, the ordered `tool_call_log` of $\{\text{name}, \text{args}, \text{result}\}$ triples, total call count, branching statistics, and iteration counts; (iv) instantiates fresh tool objects per run; and (v) compiles the layered LTL constraint set (L1–L6, Section A.4) from the gold trace. The agent sees only the numbers, never the difficulty index. The structural layers L3, L4, and L5 together carry approximately 73% of the total constraint weight; L6 is downweighted to prevent a single early arithmetic error from cascading into a near-zero compliance score (cf. Section 4.1).

Type	Template	Principal failure mode
Sequential	`loop_termination`	Over-iteration; wrong accumulation rule
	`branch_selection`	Mis-routing on per-tool call branch
Fan-out	`fan_out_fan_in`	Structural shortcutting; misaggregation
	`forall_processing`	Skipped or duplicated elements
	`exists_selection`	Missed qualifier; spurious processing
Nested loops	`nested_loops`	Premature batching; loss of intermediates
Multi-branch	`tool_condition_branching`	Working-memory degradation
Composite	`dynamic_expansion`	Depth-counter drift; revisits
	`forall_exists_combined`	Filter–aggregation interaction
	`nested_loops+branching`	Branch–iteration interaction
	`fan_out+loop`	Per-element loop dropout
	`loop+expansion`	Termination–expansion interaction

Table 8: The twelve production templates and the failure mode each one isolates.

Template	Scaling parameter	D_1	D_2	D_3	D_4	D_5
`loop_termination`	`target`	10	25	50	100	200
`branch_selection`	`num_decisions`	2	3	5	7	9
`fan_out_fan_in`	`num_items`	3	4	5	6	7
`forall_processing`	`num_items`	3	4	5	6	7
`exists_selection`	`num_items`	3	4	5	6	7
`nested_loops`	N (outer iters)	2	3	4	5	6
`tool_condition_branching`	d (passed to `tool_iterations`)	1	2	3	4	5
`dynamic_expansion`	`max_depth`	2	3	4	5	6
`forall_exists_combined`	`num_items`	3	4	5	6	7
`nested_loops+branching`	N	2	3	4	5	6
`fan_out+loop`	`num_items`, r	3, 2	4, 2	5, 3	6, 3	7, 4
`loop+expansion`	`target`	10	25	50	100	200

Table 9: Difficulty scaling per template.

Determinism from seed. The `seed` controls the initial BFS node, a derived parameter $j = (\text{seed} \bmod 10) + 1$, the list contents returned by `tool_a_list`, and loop counts via `tool_iterations`. All arithmetic is pure — given the same triplet, the prompt, gold trace, answer, and constraints are always identical.

Instance contents. Each generated instance carries the fields listed in Table 10.

Field	Description
`prompt`	Full natural-language task description with inlined numeric parameters.
`gold_trace`	Ordered list of `{tool_name, args, result}` — the correct trajectory.
`expected_answer`	Ground-truth integer answer.
`constraints`	Serialized LTL constraint list, layered L1–L6.
`num_tool_calls_expected`	Total tool calls in the correct trajectory.
`difficulty`	Integer in $\{1, \ldots, 5\}$.

Table 10: Fields stored with each generated instance.

C.5 Production vs. Simplified Mode

The two settings share procedural primitives (loops, branches, fan-out, fan-in, conditional reprocessing) but differ in vocabulary, prompt length, and parameter ranges. Table 11 summarises the differences.

Why two settings. The 3–8 call window of simplified mode provides a tight, tractable supervision signal for GRPO; training on very long production traces (D_4/D_5) would require many more samples and GPU hours. Simplified mode preserves the structural patterns that the LTL constraints test, but at a scale where the reward signal is dense and informative. The procedure-level identity between modes allows patterns trained in simplified mode to be evaluated under the longer, richer production prompts (cross-mode transfer).

Difficulty in training data. In the simplified-mode training corpus, difficulty is derived *post hoc* from tool-call count: $\leq 5 \rightarrow D_1$, $\leq 15 \rightarrow D_2$,

Aspect	Production mode	Simplified mode
Tool vocabulary	8 tools (a/b/c/d + a_list, iterations, expand, value)	4-tool pool $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$
Trace length	D_1: ∼3–10 calls; D_5: up to 50 calls (filtered)	3–7 typical, hard-capped at 8
Parameter ranges	Wide: target ≤ 200, depth ≤ 6, list ≤ 7 items	Narrow: targets ≤ 50, $N \leq 6$
Prompt style	Formal procedure blocks ("PROCEDURE / RULES / ANSWER FORMAT")	Short inline sentences
Fan-out init.	tool_a_list(seed, difficulty) returning a list in one call	Sequential tool_a(seed $+ i$) calls, one per element
Loop-count oracle	tool_iterations(seed, difficulty)	Target threshold drives loop
Graph expansion	tool_expand, tool_value	Not present
Purpose	Benchmarking and trace/entity-grounding evaluation	GRPO finetuning

Table 11: Production vs. simplified template settings.

$\leq 30 \rightarrow D_3$, $\leq 60 \rightarrow D_4$, $> 60 \rightarrow D_5$. Almost all training instances are D_1–D_2.

D Training Data and Patterns

D.1 Simplified Training Templates

The training corpus is built from 15 simplified templates over the four-tool simplified-mode pool $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. Trace lengths range from 3 to 7 calls; with augmentation (Section D.3) the cap is 8 calls. Section D.1 enumerates the 15 templates with their high-level structure. We define 17 simplified-mode templates in total; the remaining 2 are reserved as the unseen-pattern held-out split (see *Held-out patterns* below).

Four of the 15 training templates (those marked a/c) admit a transform variant: the seed transform can be performed either by tool_a or by tool_c, yielding two interchangeable instantiations of the same procedural skeleton. Tool interchangeability is restricted to the tool_a/tool_c axis; tool_value (used by dynamic_expansion) is intentionally excluded.

Held-out patterns. In addition to the 15 training templates, two further simplified-mode patterns are reserved exclusively for the unseen-pattern evaluation split: simplified versions of loop_termination and branch_selection restricted to the 3–6 call budget. These two held-out simplified patterns happen to use only the tool_a/tool_b/tool_c subset of the four-tool simplified-mode pool; they are excluded from the training corpus.

D.2 Pattern Coverage Mapping

To evaluate generalisation, we classify each of the 12 benchmark templates by its relationship to the training set. Table 13 lists, for each training template, both its *direct-simplification* target (the benchmark template of which it is a length- and arity-reduced version) and its *motif-overlap* targets (benchmark templates that share structural motifs without being direct simplifications).

This induces a three-way partition of the benchmark suite used in our generalisation analysis:

- **Direct-simplification** (4 templates): fan_out_fan_in, forall_processing, exists_selection, nested_loops. Each has a training template that is a length-reduced version of it.
- **Motif-overlap** (7 templates): loop_termination, branch_selection, tool_condition_branching, forall_exists_combined, nested_loops+branching, fan_out+loop, loop+expansion. No training template is a direct simplification, but motifs (branching, fan-out, nested loops) are represented in training.
- **No-coverage** (1 template): dynamic_expansion. This template depends on graph-expansion motifs and on tools (tool_expand, tool_value) absent from all 15 training templates.

This partition licenses the generalisation claims of Section 4.3: direct-simplification measures in-distribution transfer from simplified to production prompts, motif-overlap measures compositional transfer to recombined motifs, and no-coverage measures transfer to a benchmark family with no shared tools or motifs.

Pattern	Structure (high-level)	Tools
three_call_branch_bridge	$a \to$ branch$(b \mid c) \to c$	a, b, c
chain_then_branch	$a \to b \to$ branch$(c \mid b) \to c$	a, b, c
rebranch_after_c	$a \to c \to$ branch$(b \mid c) \to b$	a, b, c
join_then_branch	a; parallel b and c from a; join with b; final branch$(c \mid b)$	a, b, c
dual_branch_merge	a; two branches from a (left, right); merge with b	a, b, c
loop2_then_branch	$a \to b(i,1), b(i,2) \to$ aggregate $b \to$ branch$(c \mid b)$	a, b, c
branch_verify_finalize	$a \to$ branch$(c \mid b) \to$ verify $b \to$ branch$(c \mid b)$	a, b, c
six_call_double_rebranch	$a \to b \to$ branch$(c \mid b) \to b \to$ branch$(c \mid b) \to c$	a, b, c
fan_out_fan_in	k fan-out calls $\to$ per-item branch$(c \mid b) \to$ aggregate d	a, b, c, d
forall_processing	k fan-out calls $\to$ per-item uniform $b \to$ aggregate d	a, b, c, d
exists_selection	k fan-out calls $\to$ per-item $b \to$ extra c for qualifiers $\to$ sum	a, b, c, d
nested_loops_fixed_2x2	two seed transforms $\to$ fixed $2{\times}2$ inner b grid $\to$ aggregate d	a/c, b, d
filter_then_process	two seed transforms $\to$ two b calls $\to$ conditional $c \to$ aggregate d	a/c, b, c, d
fanout_then_transform	two seed transforms $\to$ per-item branch$(c \mid b) \to$ per-item $b \to$ aggregate d	a/c, b, c, d
branch_then_reprocess	transform $\to$ branch$(c \mid b) \to b \to$ second branch$(c \mid b) \to$ aggregate d	a/c, b, c, d

Table 12: The 15 simplified training templates. The notation branch$(c \mid b)$ denotes a tool-conditional routing decision; b$(i,1)$ denotes the j-th call to tool_b on item i. Templates marked a/c in the *Tools* column admit a transform variant in which the seed transform can be performed either by tool_a or by tool_c.

Training template	Direct-simplification coverage	Motif-overlap coverage
fan_out_fan_in	fan_out_fan_in	fan_out+loop, forall_exists_combined
forall_processing	forall_processing	forall_exists_combined
exists_selection	exists_selection	forall_exists_combined
nested_loops_fixed_2x2	nested_loops	nested_loops+branching, fan_out+loop
filter_then_process	—	forall_exists_combined
fanout_then_transform	—	fan_out+loop, tool_condition_branching
branch_then_reprocess	—	tool_condition_branching, nested_loops+branching
three_call_branch_bridge	—	branch_selection, tool_condition_branching
chain_then_branch	—	branch_selection, tool_condition_branching
rebranch_after_c	—	tool_condition_branching
join_then_branch	—	forall_exists_combined
dual_branch_merge	—	nested_loops+branching, tool_condition_branching
loop2_then_branch	—	loop_termination, nested_loops, loop+expansion
branch_verify_finalize	—	nested_loops+branching, tool_condition_branching
six_call_double_rebranch	—	nested_loops+branching, tool_condition_branching

Table 13: Coverage of benchmark templates by training templates.

D.3 Training Data Construction Pipeline

The final training set of 300 examples is constructed in three stages: exhaustive enumeration of trace variants, augmentation, and balanced subsampling.

Pool construction. We first enumerate every *pattern variant*, each is composed of a template and a transform-choice. Of the 15 templates, 11 have a single fixed transform and 4 admit two transform variants (cf. Section D.1), giving $11 + 4 \times 2 = 19$ pattern-variant slots. For each slot we generate 30 core instances by varying parameter seeds, yielding a pool of $19 \times 30 = 570$ unique cores before augmentation.

Trace augmentation. To discourage memorization of fixed call sequences, every core trace is expanded into up to four *augmentation variants*, tagged in round-robin order to give exact $1/4$ class balance per pattern:

none Original trace unchanged.

prefix Prepend one independent tool_a(pre_seed) call with pre_seed $\in [1000, 9999]$; its result is added to the final answer.

suffix Append one independent tool_a(suf_seed) call with suf_seed $\in [10000, 19999]$; its result is added to the final answer.

both prefix and suffix combined.

The prompt is modified accordingly: prefix instances begin with "Call tool_a(pre_seed) first — its result is an addend you will include in your final answer…", and suffix instances append the mirror sentence. To stay within the simplified-mode

Listing 1: Metrics record μ for a representative agent trace. The `tool_calls` list is the sole input to the `Trace` constructor; the token counts are stored for analysis but not used by the compliance engine.


```
{
  "num_steps": 7,
  "num_tool_calls": 6,
  "input_tokens": 13268,
  "output_tokens": 127,
  "total_tokens": 13395,
  "tool_calls": [
    {
      "step": 2,
      "tool_name": "tool_a",
      "arguments": {"seed": 42},
      "tool_result": "{\"numbers\": [8,
          2, 46]}"
    },
    {
      "step": 3,
      "tool_name": "tool_b",
      "arguments": {"i": 8, "j": 0},
      "tool_result": "{\"result\": 11}"
    },
    {
      "step": 4,
      "tool_name": "tool_b",
      "arguments": {"i": 2, "j": 0},
      "tool_result": "{\"result\": 11}"
    },
    {
      "step": 5,
      "tool_name": "tool_b",
      "arguments": {"i": 46, "j": 0},
      "tool_result": "{\"result\": 11}"
    },
    {
      "step": 6,
      "tool_name": "tool_d",
      "arguments": {"numbers": [11, 11,
          11]},
      "tool_result": "{\"result\": 33}"
    }
  ]
}
```


8-call hard cap, traces with ≥ 7 core calls use only `none` $\rightarrow$ `prefix`, keeping the total ≤ 8 calls.

Effect on constraints. The extra `tool_a` step is functionally independent of the main computation but changes the constraint profile: L3 (sequence ordering) includes the extra `tool_a`; L4 (pair/instance ordering) gains an additional ordering relation; L5 (call counts) sees the `tool_a` count 1–2 higher; L2 and L6 (argument constraints) cover the new step. An augmented trace looks structurally different, it has different first/last tool and a different total call, but requires the same procedural reasoning.

Balanced subsampling. From the augmented pool we draw 300 examples under a per-template floor/ceiling constraint (min_per_pattern $= 8$, max_per_pattern $= 24$). Quotas are computed by weighted largest-remainder allocation. Within each template's quota, the sampler equalises across within-pattern buckets (transform variant, augmentation type) so that no axis is over-represented. With the default weights this gives $15 \times 8 = 120$ floor allocations plus 180 extra slots distributed evenly (12 per template), for a final target of 20 examples per template.

D.4 Tool Name Permutations (Diverse-Tool Split)

The diverse-tool evaluation split substitutes domain-specific aliases for the generic tool names at test time. The underlying arithmetic functions are unchanged and only the tool names and argument names change. Table 14 gives the pool size and an illustrative alias for each base tool. Domains covered include finance, healthcare, DevOps, data science, logistics, IoT, genomics, and legal.

A `fan_out_fan_in` instance might therefore present as `enumerate_patients(intake_seed)` $\rightarrow$ `correlate_signals(channel_a, channel_b)` $\rightarrow$ `normalize_reading(reading)` $\rightarrow$ `accumulate_readings(readings)`. This split tests whether the model has learned procedure *structure* versus surface-level pattern matching on tool names. The LTL constraints are restated in terms of the aliased tool names, so the compliance test is invariant under renaming.

E Training Details (GRPO)

This appendix provides additional implementation details for the GRPO finetuning setup described in Section 4.3. We specify the full reward definition, the trace-distance metric used as a dense shaping reward, and the main training hyperparameters.

E.1 Reward Definition

For each sampled trajectory, the total reward is a weighted combination of three terms: compliance with the AGENTLTL specification, answer correctness, and distance to the gold tool trace. The reward is defined as

$$R = w_C C + w_A Acc + w_D R_{\text{dist}},$$

where C is the AGENTLTL compliance reward, A is the shaped answer-correctness reward, and R_{dist} is a dense trace-distance reward.

Base tool	# aliases	Example alias	Example arg rename
tool_a	17	verify_checksum	i → index
tool_a_list	8	list_transaction_ids	seed → batch_seed
tool_iterations	8	compute_retry_budget	seed → execution_seed
tool_b	15	cross_reference	i, j → primary_id, secondary_id
tool_c	16	transform_value	x → value
tool_d	15	aggregate_metrics	numbers → metrics
tool_expand	15	explore_dependencies	node → package_id
tool_value	≥ 10	get_node_weight	node → node_id

Table 14: Alias pool sizes and example mappings.

In the training run, the weights are

$$(w_C, w_A, w_D) = \left(\alpha, \frac{1-\alpha}{2}, \frac{1-\alpha}{2}\right), \qquad \alpha = 0.5.$$

Thus, the reward used in the main experiments is

$$R = 0.5\,C + 0.25\,Acc + 0.25\,R_{\text{dist}}.$$

The answer-correctness term is shaped to preserve a weak learning signal even when the final answer is formatted correctly but numerically incorrect. Specifically,

$$\text{Acc} = \begin{cases} 1.0 & \text{correct format and value,} \\ 0.1 & \text{correct format, wrong value,} \\ 0.0 & \text{invalid format.} \end{cases}$$

This shaping makes answer correctness an outcome anchor while avoiding an entirely sparse signal for formatted but incorrect completions.

E.2 Trace Distance Metric

The dense trace reward is based on the distance between the predicted tool trace and the gold tool trace. Let the gold trace be

$$G = (g_1, \ldots, g_n)$$

and the predicted trace be

$$P = (p_1, \ldots, p_m).$$

We compute a weighted Levenshtein distance $d(G, P)$ using the dynamic program

$$D(0, j) = j, \qquad D(i, 0) = i,$$

and

$$\begin{aligned} D(i,j) = \min\big(&D(i-1,j)+1, \\ &D(i,j-1)+1, \\ &D(i-1,j-1)+s(g_i,p_j)\big). \end{aligned}$$

where $D(i, j)$ is the minimum edit cost between the first i elements of the gold trace and the first j elements of the predicted trace. The substitution cost $s(g_i, p_j)$ is

$$s(g_i, p_j) = \begin{cases} 0 & \text{exact match,} \\ 0.5 & \text{same tool, arguments differ,} \\ 1.0 & \text{tool name differs.} \end{cases}$$

Insertions and deletions each have cost 1.0. The final distance is

$$d(G, P) = D(n, m).$$

The trace-distance reward is then

$$R_{\text{dist}} = \exp(-d(G, P)).$$

Equivalently, the edit costs are:

- tool mismatch: 1.0;
- argument mismatch for the same tool: 0.5;
- insertion or deletion: 1.0.

As an example, consider the gold trace

$$G = [(\texttt{tool_a}, \{x : 1\}), (\texttt{tool_b}, \{y : 2\})]$$

and the predicted trace

$$P = \begin{bmatrix} (\texttt{tool_a}, \{x : 9\}), \\ (\texttt{tool_c}, \{z : 0\}), \\ (\texttt{tool_b}, \{y : 2\}) \end{bmatrix}.$$

The optimal alignment incurs an argument mismatch on `tool_a`, with cost 0.5, an insertion of `tool_c`, with cost 1.0, and an exact match on `tool_b`, with cost 0. Therefore,

$$d(G, P) = 0.5 + 1.0 + 0 = 1.5,$$

and

$$R_{\text{dist}} = \exp(-1.5) \approx 0.223.$$

E.3 Training Setup

The GRPO finetuning run uses Qwen/Qwen3-4B-Instruct-2507 as the base model. We finetune using LoRA adapters applied to the `q_proj` and `v_proj` modules. The LoRA rank is

$$r = 8,$$

with LoRA scaling parameter

$$\alpha = 16$$

and dropout

$$0.05.$$

For each prompt, the trainer samples

$$N = 16$$

rollouts. Training uses a round-robin-no-repeat curriculum across pattern types. In this mode, the curriculum expands examples over epochs without repeating the same pattern consecutively, and dataset shuffling is disabled.

The pipeline uses fp32 precision. The pipeline KL coefficient is $\beta = 10^{-3}$.

The released adapter metadata is consistent with the training configuration: LoRA rank $r = 8$, LoRA scaling parameter $\alpha = 16$, dropout 0.05, and target modules `q_proj` and `v_proj`.

F Benchmark Results (Extended)

This appendix presents the full per-template, per-layer, and per-setting breakdowns of the benchmark results summarised in Section 5.1. Layer definitions (L1 branch, L2 argument extraction, L3 global sequence, L4 pair-order, L5 call-count, L6 exact args) follow Section 3.3, and the three enforcement settings (baseline, block-and-warn, soft-block) follow Section 4.2.

F.1 Per-Template Results

Figure 6 reports compliance and accuracy for each (model, template) cell under all three enforcement settings. The figure shows a 2×3 grid of heatmaps: rows correspond to compliance score (top) and accuracy (bottom), and columns correspond to the baseline, soft-block, and block-and-warn settings. Within each heatmap, columns index models and rows index templates; cells are colour-coded green (high) to red (low).

Template	Pattern
`forall_processing`, `fan_out_fan_in`	High compliance and accuracy across all models and settings; enforcement does not degrade performance.
`loop_termination`, `nested_loops`, `nested_loops+branching`	Low accuracy for smaller or weaker models (Gemma-4-26B-A4B, Qwen3-Next-80B-A3B-Instruct) under baseline; block-and-warn recovers accuracy for DeepSeek-V4-Flash by +25–30 pp.
`tool_condition_branching`, `fan_out+loop`	Worst-case templates for soft-block: DeepSeek-V4-Flash drops from ∼37% to ∼17% accuracy under soft-block, while block-and-warn recovers it to ∼40%.
`exists_selection`	Gemma-4-26B-A4B collapses to 6.7% accuracy under baseline but maintains 96.5% compliance — one of several compliance-without-correctness cells consistent with the Qwen3-Next-80B-A3B-Instruct pattern reported in Section 5.1.

Table 15: Per-template behavioural patterns observed in Figure 6.

Key observations. Table 15 summarises the dominant patterns visible in Figure 6.

Compliance and accuracy diverge most sharply on composite templates (`nested_loops+branching`, `fan_out+loop`, `loop+expansion`) where the model satisfies per-call constraint checks but still emits incorrect call sequences. This is consistent with the L3/L4 ordering failures reported in Section 5.1.

F.2 Per-Layer Breakdowns

Figure 7 expands Table 4 of Section 5.1 into a per-setting view. The heatmap has 21 rows (7 models $\times$ 3 settings) and 8 columns: the two summary metrics (**Compliance**, **Accuracy**) followed by per-layer pass rates (**L1–L6**). Dashed navy lines separate model groups, and a vertical dashed line separates the two summary columns from the layer columns. Each cell shows the mean pass rate of the constraint checks evaluated at that layer, conditional on the model having reached a callable state at that point in the run; this is the loss-contribution decomposition referenced in Section 5.1, where a model's accuracy can be reconstructed as the product of pass rates across active layers.

Full numeric values. Table 16 reports the underlying numbers for Figure 7. The best (model, setting) row by accuracy (Qwen3.5-397B-A17B baseline) and the best single-model compliance un-

der enforcement (Gemma-4-31B block-and-warn) are highlighted in bold.

Layer-level findings. Three observations stand out from Table 16 and complement the L3/L4 analysis in Section 5.1.

- **L3 (global sequence) is the most consistently weak layer** across all models and settings, in line with Section 5.1. Even the strongest models exhibit meaningful L3 gaps under baseline (Qwen3.5-397B-A17B: 0.622; Kimi-K2.6: 0.450), and DeepSeek-V4-Pro has the second-lowest baseline L3 of any model (0.153). L4 (pair-order) tracks L3 closely throughout, with L4 typically 5–20 pp above L3 (e.g. for DeepSeek-V4-Pro baseline, L3 $= 0.153$ vs. L4 $= 0.312$), consistent with the same trace failures violating both global and pairwise ordering.

- **L1 (branch) is near-perfect for strong models** under baseline but collapses under soft-block for DeepSeek-V4-Flash (0.212) and Gemma-4-26B-A4B (0.424), indicating that soft-block stops disrupt branch-selection logic for weaker models before they can recover.

- **Block-and-warn consistently lifts DeepSeek-V4-Flash** across every layer relative to baseline (L1 +24 pp, L2 +14 pp, L3 +64 pp, L4 +61 pp), making it the setting most beneficial for weaker models. The largest gains are precisely on L3 and L4 which are the ordering layers that account for most of the weighted loss in Section 5.1.

F.3 Enforcement Statistics

Figure 8 reports three views of enforcement behaviour under block-and-warn: enforcement event counts per layer (stacked by model), listen-versus-insist rates, and tool-call deltas versus baseline.

(a) Enforcement events per layer. Table 17 reports the total number of enforcement events fired at each layer, together with the number of resolved, *insisted*, and *listened* resolutions, and the resulting insist rate. An enforcement event is resolved as *listened* when the model abandons the violating call after a warning, and as *insisted* when it repeats the same call (modulo argument-equivalence) on the next turn.

Gemma-4-26B-A4B dominates L2 and L3 enforcement events (47K and 49K respectively), driven by frequent argument-extraction and ordering violations combined with a high tool-call count per run.

(b) Listen versus insist rates. Overall, models comply with constraint warnings in the vast majority of episodes. L2 has the highest insist rate at 4.08%, meaning that models occasionally retry the same argument-extraction call after a warning. L3 and L5 insist rates are below 1.1%, indicating that models largely accept sequencing and count constraints. L1's insist rate is essentially zero (0.16%): once warned about a wrong branch, models almost never repeat the same routing decision.

(c) Tool-call deltas versus baseline. Block-and-warn typically adds tool calls for most models (DeepSeek-V4-Flash: $\sim$+6 extra calls; Gemma-4-26B-A4B: $\sim$+10) because warning-repair cycles introduce additional retries after each warning. Soft-block reduces or barely changes tool-call counts for most models, since stopped runs terminate early and therefore contribute fewer total calls. The exception is Gemma-4-31B under soft-block, which shows a slight positive delta. This model recovers well from soft-block stops and continues executing.

F.4 Additional Analyses

F.4.1 Difficulty Scaling Curves

Figure 5 plots accuracy (top row) and compliance (bottom row) against difficulty levels D1–D5 for each enforcement setting. Each panel shows one curve per model.

1. **Accuracy degrades monotonically with difficulty** for most models under baseline and block-and-warn. DeepSeek-V4-Flash baseline drops from 62.5% at D1 to 56.9% at D5 (mild decline); Gemma-4-26B-A4B baseline drops from 38.9% at D1 to 25.0% at D5 (-13 pp over five levels).

2. **Block-and-warn amplifies difficulty sensitivity for weaker models.** DeepSeek-V4-Flash under block-and-warn falls from 84.7% at D1 to 70.8% at D5 because harder instances trigger more warning cycles.

3. **Soft-block dramatically suppresses accuracy at all difficulty levels.** DeepSeek-V4-Flash falls to the 25–38% range under soft-block versus 50–62% under baseline, and the degradation is roughly flat across difficulty

Model	Setting	Comp.	Acc.	L1	L2	L3	L4	L5	L6
Qwen3-Next-80B-A3B-Instruct	Baseline	0.874	0.433	0.862	0.992	0.686	0.805	0.889	0.944
Qwen3-Next-80B-A3B-Instruct	Block-and-warn	0.829	0.403	0.794	0.881	0.711	0.855	0.852	0.790
Qwen3-Next-80B-A3B-Instruct	Soft-block	0.799	0.419	0.798	0.849	0.692	0.807	0.841	0.760
Qwen3.5-397B-A17B	**Baseline**	0.909	0.989	0.998	1.000	0.622	0.762	0.983	0.996
Qwen3.5-397B-A17B	Block-and-warn	0.896	0.842	0.869	0.927	0.778	0.894	0.934	0.831
Qwen3.5-397B-A17B	Soft-block	0.937	0.911	0.901	0.956	0.906	0.941	0.963	0.860
DeepSeek-V4-Flash	Baseline	0.596	0.556	0.652	0.741	0.119	0.249	0.864	0.681
DeepSeek-V4-Flash	Block-and-warn	0.867	0.786	0.892	0.877	0.761	0.854	0.906	0.802
DeepSeek-V4-Flash	Soft-block	0.530	0.325	0.212	0.528	0.347	0.407	0.771	0.430
DeepSeek-V4-Pro	Baseline	0.735	0.944	0.946	0.981	0.153	0.312	0.950	0.961
DeepSeek-V4-Pro	Block-and-warn	0.823	0.711	0.824	0.849	0.664	0.813	0.866	0.762
DeepSeek-V4-Pro	Soft-block	0.739	0.644	0.470	0.752	0.639	0.692	0.860	0.681
Gemma-4-26B-A4B	Baseline	0.617	0.325	0.869	0.939	0.111	0.144	0.872	0.830
Gemma-4-26B-A4B	Block-and-warn	0.717	0.300	0.649	0.806	0.506	0.708	0.822	0.650
Gemma-4-26B-A4B	Soft-block	0.623	0.275	0.424	0.677	0.433	0.564	0.809	0.549
Gemma-4-31B	Baseline	0.789	0.508	1.000	0.990	0.292	0.489	0.943	0.948
Gemma-4-31B	**Block-and-warn**	0.961	0.911	1.000	0.987	0.911	0.971	0.975	0.879
Gemma-4-31B	Soft-block	0.877	0.500	0.862	0.953	0.739	0.884	0.917	0.811
Kimi-K2.6	Baseline	0.831	0.828	0.986	0.988	0.450	0.603	0.932	0.957
Kimi-K2.6	Block-and-warn	0.891	0.767	0.899	0.938	0.772	0.889	0.928	0.820
Kimi-K2.6	Soft-block	0.724	0.597	0.625	0.743	0.631	0.680	0.853	0.659

Table 16: Per-layer pass rates by model and enforcement setting. Layers L1–L6 correspond to branch, argument extraction, global sequence, pair-order, call-count, and exact args, respectively. Bold rows mark the best single-model compliance (Gemma-4-31B, block-and-warn) and best single-model accuracy (Qwen3.5-397B-A17B, baseline).

Layer	Enforcement events	Resolved	Insisted	Listened	Insist rate (%)
L1 (Branch)	1213	1201	2	1199	0.16
L2 (Argument extraction)	61 111	59 208	2492	56 716	4.08
L3 (Global sequence)	62 811	62 051	680	61 371	1.08
L4 (Pair-order)	0	—	—	—	—
L5 (Call-count)	10 088	9695	50	9645	0.50
L6 (Exact args)	0	—	—	—	—

Table 17: Enforcement events per layer under block-and-warn, aggregated across all models. Layers L4 (pair-order) and L6 (exact args) have zero enforcement events because the corresponding block-and-warn checks were not triggered by any run.

levels because stop-on-violation terminates runs regardless of difficulty.

4. **Compliance is far less sensitive to difficulty than accuracy.** Across all models and settings, compliance variance across D1–D5 is typically under 5 pp, consistent with compliance measuring constraint adherence per call rather than end-to-end task success.

5. **Qwen3.5-397B-A17B is near-flat in accuracy (∼98.6%)** across all difficulty levels under baseline, confirming that it is largely saturated on the benchmark.

F.4.2 Compliance vs. Correctness Correlation

We also examined the per-cell relationship between compliance and accuracy. For each enforcement setting, we computed the Pearson correlation between compliance and accuracy taken over all (model, template) cells ($N = 84$ per setting, from 7 models $\times$ 12 templates).

Setting	r
Baseline	0.60
Soft-block	0.63
Block-and-warn	0.73

Table 18: Pearson correlation between compliance and accuracy across (model, template) cells, by enforcement setting ($N = 84$ per setting).

Interpretations. The moderate baseline correlation ($r \approx 0.60$) confirms that compliance and accuracy are correlated but not equivalent: models can

satisfy constraint checks without solving the task, especially on composite templates. The higher correlation under block-and-warn ($r \approx 0.73$) reflects that enforcement pushes compliance and accuracy in the same direction; when the model recovers from a warning, it also tends to complete the task correctly. Cells in the high-compliance / low-accuracy region correspond predominantly to Qwen3-Next-80B-A3B-Instruct and Gemma-4-26B-A4B on complex templates, where the model follows call-by-call constraints but fails to produce the correct final answer, the same pattern reported for Qwen3-Next-80B in Section 5.1. Cells in the low-compliance / high-accuracy region are rare and correspond mostly to DeepSeek-V4-Pro, consistent with the main-text observation that this model produces correct answers from non-compliant traces. All settings also show a dense cluster near $(1.0, 1.0)$ populated by Qwen3.5-397B-A17B and Kimi-K2.6 on straightforward templates.

G Finetuning Results (Extended)

This appendix complements Section 5.2 with full-resolution training curves and a per-template view of where the finetuned model gains and where it does not. Throughout, $\bar{C}$ denotes empirical compliance averaged over a split, τ a model-produced trace, and τ^* the gold trace used for the distance-based reward term $R_{\text{dist}}(\tau, \tau^*)$ (Section 4.3).

G.1 Training curves

Figure 4 shows the three reward components and the weighted total tracked during GRPO training, evaluated on the held-out validation set every checkpoint over 4,800 generations (16 rollouts $\times$ 300 prompts). Figure 3 shows the same components on the training rollouts. The training-side and evaluation-side trajectories move in parallel, with no late-training divergence indicative of overfitting; the gap between them narrows over training and stabilises within the first half of the run.

All three reward components improve jointly. AGENTLTL compliance $\bar{C}$ rises from 0.511 to 0.770, answer correctness from 0.296 to 0.728, and gold-trace distance reward R_{dist} from 0.301 to 0.782. Joint movement rules out the failure mode of reward gaming via structurally-valid-but-wrong traces: a model maximising $\bar{C}$ alone could trivially produce empty or near-empty traces, but the correctness and trace-distance components would not co-move.

G.2 Per-template gains

Table 19 reports per-template accuracy and $\bar{C}$ for every benchmark template, aggregated over the four held-out validation splits (unaugmented, augmented, unseen-pattern, and diverse-tool; n=261 in total) and ordered by accuracy gain. The distribution of gains has three regions worth distinguishing.

Strong gains (top thirteen templates). The thirteen templates from `three_call_branch_bridge` through `branch_verify_finalize` all clear +25pp in accuracy and show $\bar{C}$ gains in the same direction. The largest movers are `three_call_branch_bridge` ($0.0 \rightarrow 100.0$), `dual_branch_merge` ($8.3 \rightarrow 100.0$), and `rebranch_after_c` ($0.0 \rightarrow 78.6$). Compliance gains in this region are uniformly positive ($+0.08$ to $+0.61$), with the largest $\Delta\bar{C}$ attached to the templates that move from 0% accuracy: when the base model produces malformed traces, finetuning improves both axes simultaneously.

Modest gains. Two templates show only marginal accuracy improvement: `six_call_double_rebranch` (+7.1pp) and `filter_then_process` (+5.6pp). These are also the only templates other than `nested_loops_fixed_2x2` on which $\bar{C}$ drops after finetuning (-0.015 and -0.202 respectively). The common factor is high constraint-formula depth combined with long gold traces (six or more calls); the model attempts the structure but violates one of the inner constraints, and the larger compliance drop on `filter_then_process` reflects a structural attempt that is further from the gold pattern than the base model's shorter output.

Regression: `nested_loops_fixed_2x2`. The one substantial regression is `nested_loops_fixed_2x2`, which drops from 85.7% to 14.3% (-71.4pp), with $\bar{C}$ also falling from 0.956 to 0.560. The base model's high accuracy on this template is, on inspection, a near-vacuous pass: the base produces very short traces that happen to satisfy the (relatively permissive) outer constraint while not actually executing the nested loop. After finetuning the model attempts the full nested-loop structure and exposes itself to the inner-loop termination constraints, on which it then fails. The same pattern, deeply nested control flow producing the model's only consistent failure mode, is noted in the main-text discussion of `nested_loops+branching` on the full benchmark, of which `nested_loops_fix`

ed_2x2 is the training-side simplified counterpart.

fanout_then_transform unchanged. fanout_then_transform is the one template where accuracy is flat ($14.3\% \rightarrow 14.3\%$). Both $\bar{C}$ ($+0.013$) and R_{dist} move in the expected direction, so the model is learning structure on this template; the flat accuracy appears to be a small-n artefact (n=7) rather than a genuine absence of improvement.

H Trace Grounding Study

This appendix provides implementation details, examples, and full results for the trace-grounding analysis introduced in Sections 4.4 and 5.3. We describe the two prompt variants used, the repository corpus, the implementation of the grounding predicate κ_{ground}, three worked example runs, the effect of the strict-grounding prompt, popularity stratification, and a comparison with trace-aware and trace-blind LLM judges.

Notation bridge. Section 5.3 reports the joint distribution of correctness and grounding using four categories: **CG** (correct, grounded), **CU** (correct, ungrounded), **IG** (incorrect, grounded), and **IU** (incorrect, ungrounded). The appendix reuses these labels throughout. Where useful for comparison to an LLM judge that itself produces a binary grounded/ungrounded verdict (§H.7), we additionally report the confusion-matrix convention with κ_{ground}-ungrounded as the positive class: **TP** = IU, **FP** = CU, **FN** = IG, **TN** = CG. Precision is $\text{TP}/(\text{TP}+\text{FP})$ and recall is $\text{TP}/(\text{TP}+\text{FN})$, both with respect to identifying incorrect answers via ungroundedness.

H.1 Prompt Details

Both prompt variants share an identical system preamble describing the available tools (list_directory, read_file) and a user turn of the form:

```
Repository: {repo_id}
TASK
--
{question_text}
Investigate the source code using the
available tools and provide a thorough
answer grounded in what you find. Cite
specific files, classes, functions, and
code patterns you discover.
```

The two variants differ only in the rules block.

Default prompt. Four soft instructions:

1. Always explore the directory structure before reading files.
2. Read the actual source code to answer questions — do not rely on memory.
3. Cite specific files, classes, functions, and code you find.
4. If you are unsure about something, look it up in the code first.

Strict grounding prompt. Replaces the soft instructions with three hard constraints:

1. Only state facts about files, classes, functions, or code directly read via read_file in this session. Do not use prior knowledge about this repository.
2. Every file path, class name, function name, or signature mentioned in the final answer must be one seen in a read_file result during this session. Names seen only in list_directory output do not qualify unless the file was also opened.
3. If after reasonable exploration the answer cannot be found in the source read, the final answer must be exactly "I don't know - not found in the source I read." Refusal is a valid and preferred answer when grounding is missing.

H.2 Repository Dataset

The corpus consists of 16 Python open-source repositories, sampled across GitHub-star popularity deciles to span the spectrum of likely LLM pretraining representation. Each repository is pinned to a specific commit or tag at corpus-construction time to guarantee reproducibility. Each repository is paired with 10 authored questions, yielding 160 (repository, question) pairs. Questions follow a uniform template:

> *In the* {repo} *repository, where is the* {symbol_name} *defined? Provide the exact file path.*

Ground truth is a human-authored (repo, file path, symbol name) triple verified against the pinned commit. Answer correctness is decided by case-insensitive substring match of all ground-truth entities against the agent's final answer.

Template	n	Accuracy (%)		ΔAcc.	$\bar{C}$		$\Delta\bar{C}$
		Base	Tuned		Base	Tuned	
three_call_branch_bridge	14	0.0	**100.0**	+100.0	0.373	0.987	+0.614
dual_branch_merge	12	8.3	**100.0**	+91.7	0.707	0.971	+0.263
rebranch_after_c	14	0.0	**78.6**	+78.6	0.589	0.915	+0.326
exists_selection	20	5.0	**70.0**	+65.0	0.572	0.883	+0.312
loop2_then_branch	14	7.1	**71.4**	+64.3	0.553	0.879	+0.327
join_then_branch	14	7.1	**64.3**	+57.1	0.635	0.875	+0.241
forall_processing	22	18.2	**72.7**	+54.5	0.748	0.885	+0.136
chain_then_branch	12	0.0	**50.0**	+50.0	0.595	0.824	+0.230
branch_selection†	24	4.2	**45.8**	+41.7	0.696	0.772	+0.076
fan_out_fan_in	22	36.4	**77.3**	+40.9	0.626	0.889	+0.264
branch_then_reprocess	20	10.0	**50.0**	+40.0	0.476	0.743	+0.267
loop_termination†	15	13.3	**46.7**	+33.3	0.520	0.852	+0.333
branch_verify_finalize	12	0.0	**25.0**	+25.0	0.411	0.641	+0.231
six_call_double_rebranch	14	0.0	**7.1**	+7.1	0.618	0.603	−0.015
filter_then_process	18	0.0	**5.6**	+5.6	0.707	0.505	−0.202
fanout_then_transform	7	14.3	14.3	0.0	0.584	0.597	+0.013
nested_loops_fixed_2x2	7	**85.7**	14.3	−71.4	0.956	0.560	−0.395

Table 19: Per-template accuracy and $\bar{C}$ for the base and finetuned models on the four held-out validation splits combined (n=261), ordered by accuracy gain. Bold marks the higher accuracy in each row. Templates marked † are the two unseen-pattern templates held out from training.

Repository list. Table 20 lists all repositories with the pinned version, star count at sampling time, and assigned popularity bucket.

Repository	Stars	D	B	Pinned
flask	68k	10	hi	3.1.0
requests	52k	10	hi	v2.32.3
pandas	43k	10	hi	v2.2.3
numpy	28k	9	hi	v2.2.0
matplotlib	21k	9	hi	v3.9.4
click	18k	8	mid	8.3.3
schedule	13k	7	mid	1.2.2
isort	7k	6	mid	v5.11.3
returns	4.3k	5	low	0.22.0
msgspec	3.7k	4	low	0.21.1
environs	3.6k	4	low	15.0.1
itsdangerous	2.8k	3	low	2.2.0
ocpp	1.0k	3	low	2.1.0
mac_apt	1.0k	3	low	v1.29.0
cyclopts	1.1k	3	low	v4.11.2
tldr_python_client	733	2	low	3.4.4

Table 20: Repositories used in the corpus.

Popularity buckets. The 16 repositories are partitioned into three buckets by decile: *high* (deciles 8–10, 5 repos: canonical libraries heavily represented in pretraining corpora), *mid* (deciles 5–7, 3 repos: established but less ubiquitous, likely partially memorised), and *low* (deciles 1–4, 8 repos: niche or recently-released projects unlikely to be memorised). The high bucket is intentionally smaller because a few canonical repositories dominate the top of the star distribution; a balanced design would over-sample them. The low bucket deliberately captures more of the diversity in the tail.

H.3 Grounding Predicate Implementation

The grounding predicate κ_{ground} is deterministic: given the same trace it always returns the same Boolean verdict, and no LLM call or human annotation is required. It is satisfied if and only if every referential entity named in the agent's final answer (the $\text{out}(a)$ side of the predicate) was observed in at least one read_file result during the trace (the $\text{ent}(\tau)$ side).

Entity extraction from the answer ($\text{out}(a)$). Three typed entity classes are extracted from the free-form answer text using regex-only patterns (no AST parsing of natural language):

- **file_path** — repository-relative paths ending in a known source extension (.py, .c, .h, .js, .ts, ...).
- **function_signature** — patterns of the form ClassName.method(or bare function_name(.
- **identifier** — CamelCase tokens of length ≥ 3 and snake_case tokens of length ≥ 3, minus a builtin blocklist of approximately 60 entries (Python keywords, standard-library module names, common exception classes).

The blocklist suppresses spurious false positives from prose mentions of tokens such as import,

return, os, json, or re that do not constitute code-level claims.

Seen-entity set from the trace ($\mathrm{ent}(\tau)$)**.** For each read_file call in the trace, the tool result (a JSON string) is parsed to recover the content field — the raw source text actually returned to the agent. From all such chunks the seen set is built in two steps:

1. File paths: every key in the read-file result dictionary (i.e., every path the agent explicitly requested and received) is added as a grounded file path.
2. Function signatures and identifiers: AST extraction is applied to each chunk to recover named symbols actually defined in the read source. We use Python's ast.parse for .py files and libclang for C/C++ headers, with a regex fallback when libclang is unavailable.

An answer entity is *grounded* if it matches (case-insensitive prefix for identifiers, exact for file paths) some member of the seen set of the same type. The predicate is satisfied when every answer entity is grounded.

Edge cases. Several non-obvious trace patterns recur in practice. Their treatment is summarised in Table 21.
We note one known limitation: the vacuous-pass case (row 3) means κ_{ground} is satisfied by refusals such as "I don't know - not found in the source I read.", since these contain no extractable referential entities. This interacts non-trivially with the strict grounding prompt; we quantify the effect in §H.5.

H.4 Example Runs

We present three complete runs, one per outcome category, to illustrate how κ_{ground} behaves on real traces.

H.4.1 Example A — CG (correct, grounded)

Setup. Repository requests (high bucket); model Qwen3.5-9B; default prompt. Question: *where is the function* request *defined?* Ground truth: src/requests/api.py.
Trace (6 steps).

```
1 list_directory("")
2 list_directory("src")
3 list_directory("src/requests")
4 list_directory("src/requests")
  # repeated (benign)
5 read_file("src/requests/api.py")
6 final_answer("src/requests/api.py")
```

Final answer. src/requests/api.py
Evaluation. κ_{ground} = TRUE (4/4 answer entities grounded); ANSWER_CORRECT=TRUE. Category: **CG**. The agent navigates to src/requests/ before opening api.py, locates request, and cites the exact path it read.

H.4.2 Example B — IU, no read_file (canonical parametric hallucination)

Setup. Repository matplotlib (high bucket); model Qwen3.5-9B; default prompt. Question: *where is the function* print_progress *defined?* Ground truth: ci/schemas/vendor_schemas.py.
Trace (4 steps).

```
1 list_directory("")
2 list_directory("lib")
3 list_directory("lib/matplotlib")
4 final_answer\
        ("lib/matplotlib/__init__.py")
```

Final answer. lib/matplotlib/__init__.py
Evaluation. κ_{ground} = FALSE (no read_file result in trace); ANSWER_CORRECT=FALSE. Category: **IU**. The agent issues three list_directory calls and then asserts a location consistent with its parametric memory of matplotlib's layout. The actual function lives in a CI helper directory the agent never explored. This is the canonical parametric-memory hallucination that κ_{ground} is designed to catch.

H.4.3 Example C — IU, explored wrong location

Setup. Repository ocpp (low bucket); model Qwen3.5-397B-A17B; default prompt. Question: *where is the class* ChargePoint *defined?* Ground truth: examples/v16/charge_point.py (one of several versioned definitions).
Trace (10 steps).

```
1 list_directory("")
2 list_directory("ocpp")
3 read_file("ocpp/charge_point.py")
  # abstract base
4 read_file("ocpp/charge_point.py")
  # chunk 2
5 read_file("ocpp/charge_point.py")
  # chunk 3
6 read_file("ocpp/charge_point.py")
  # chunk 4
```

Situation	Verdict	Reason
`list_directory` only, no `read_file`	FALSE	No read-file evidence; cannot verify grounding.
`read_file` called but final answer is empty	FALSE	No answer to check; treated as malformed.
Answer contains only builtin / blocklist names after filtering	TRUE	No checkable entities; predicate vacuously passes.
Answer cites a path visible in `list_directory` but never read	FALSE	Directory listing alone is insufficient evidence.
Agent reads many unrelated chunks, then names a symbol present in none	FALSE	Grounding is per-entity, not per-run.

Table 21: Behaviour of κ_{ground} on recurring edge cases. The first row (no `read_file` at all) is the most common path to FALSE under the default prompt: many runs answer from the directory tree alone.

```
 7  read_file("ocpp/charge_point.py")
    # chunk 5
 8  read_file("ocpp/v16/__init__.py")
 9  read_file("ocpp/v201/__init__.py")
10  read_file("ocpp/v21/__init__.py")
11  final_answer("The class ChargePoint is
        primarily defined ...")
```

Final answer (excerpt). *"The class* `ChargePoint` *is primarily defined in the file* `ocpp/charge_point.py`*. This is the base class containing all the core OCPP functionality. ... Version-specific subclasses are found in* `ocpp/v16/__init__.py`*,* `ocpp/v201/__init__.py`*, and* `ocpp/v21/__init__.py`*."*

Evaluation. κ_{ground} = FALSE (53/76 answer entities grounded); ANSWER_CORRECT=FALSE. Category: **IU**. The agent reads the abstract base class but never explores `examples/`, where the user-facing `ChargePoint` lives. This example illustrates that even when the trace contains substantial `read_file` evidence, a confident verbose answer can still fail grounding when it makes claims about files outside the explored set or pads with referential prose.

H.5 Effect of the strict-grounding prompt

Section 5.3 reports that the strict-grounding prompt raises the grounded fraction, but that the additional grounded traces are predominantly refusals rather than grounded correct answers. Table 22 quantifies this per model. We report the percentage-point change in correctness, the percentage-point change in κ_{ground}-grounded rate, the change in mean `read_file` calls per run, and *vacuous grounding*. Vacuous grounding is measured as the fraction of grounded traces that are refusals and so satisfy the predicate trivially per the edge case in Table 21.

The three models respond very differently. `Qwen3.5-9B` shows the smallest behavioural shift (+1.05 `read_file` calls, only 16.0% of grounded traces vacuous), suggesting the strict prompt induces modestly more file reading rather than refusals. `Qwen3.5-397B-A17B` reads dramatically more files under the strict prompt (+7.03) and gains 10 points of correctness, but 91.7% of its grounded traces are refusals — the model complies with the literal constraint by refusing whenever its evidence feels insufficient. `Gemma-4-31B-IT` shows the largest grounding gain (+11.6 pp) with essentially no change in reading behaviour (+0.36) and a 71.4% vacuous-grounding rate: the gain is almost entirely refusal-driven. Across all three models, the strict prompt does not appreciably increase the CG cell (genuinely grounded correct answers); it shifts mass from IU into vacuous CG/CU refusals.

H.6 Popularity stratification

Table 23 stratifies the four-category distribution by repository popularity bucket, pooled across all three models.

Three patterns deserve note. First, correctness decreases sharply with popularity under the default prompt (52.0% → 24.0% from low to high; recall that "low" means *low popularity*, i.e. niche repositories, and "high" means canonical libraries). This is the standard reading: canonical libraries have more symbols, more namespaces, and harder retrieval, so the agent errs more often. Second, the CG rate never exceeds 4.0% in any cell: genuinely grounded correct answers remain rare even in the easiest bucket. Third, the strict prompt's largest effect is on the low bucket, where IU drops from 26.7% to 20.7% and CG rises from 0.7% to 4.0%; the high bucket sees only a 1.7 pp drop in IU. The gap between correctness and CG persists across all buckets and both prompts, confirming that the

Model	Δ Correct	Δ Grounded	Δ read_file	Vacuous grounding
Qwen3.5-9B	−3.1	+2.8	+1.05	16.0%
Qwen3.5-397B-A17B	+10.0	+3.8	+7.03	91.7%
Gemma-4-31B-IT	−0.6	+11.6	+0.36	71.4%

Table 22: Effect of the strict-grounding prompt relative to the default prompt. Values are percentage-point differences except for mean read_file calls. Vacuous grounding is the fraction of grounded traces under the strict prompt that are refusals.

Prompt	Bucket	N	Correct	Grounded	CG	CU	IG	IU
Default	Low	300	52.0%	22.0%	0.7%	51.3%	21.3%	26.7%
	Mid	240	38.8%	15.4%	1.7%	37.1%	13.8%	47.5%
	High	420	24.0%	16.2%	1.2%	22.9%	15.0%	61.0%
Strict	Low	300	50.3%	33.0%	4.0%	46.3%	29.0%	20.7%
	Mid	240	40.0%	20.4%	2.5%	37.5%	17.9%	42.1%
	High	420	29.3%	19.3%	1.7%	27.6%	17.6%	53.1%

Table 23: Four-category distribution stratified by repository popularity bucket, pooled across Qwen3.5-9B, Qwen3.5-397B-A17B, and Gemma-4-31B-IT. Bucket assignments follow Table 20.

four-category view is not an artefact of any single popularity regime.

H.7 Comparison with LLM judges

Section 5.3 reports aggregate agreement of 77.6% between κ_{ground} and a trace-aware LLM judge across all 1,917 traces, with Cohen's $\kappa = 0.09$ reflecting class imbalance. The judge flags 6.4% of answers as grounded compared to 20.9% for κ_{ground}, so the judge treats κ_{ground} as a necessary but not sufficient condition. This appendix breaks out those aggregates by model and prompt and additionally reports a trace-blind judge that evaluates binary correctness without access to the trace.

Judge setup. The judge model is DeepSeek-V4-Pro, deliberately distinct from any evaluated agent model to avoid self-evaluation bias. The trace-aware judge receives the question, the full tool trace, and the answer, and returns a binary YES/NO verdict on whether every factual claim is supported. The trace-blind judge receives only the question and the answer and returns a binary correctness verdict. Each (run, prompt, judge variant) is sampled five times at temperature 0.7; the verdict is the mean of the five samples, thresholded at 0.5.

Trace-aware judge prompt.

```
You are evaluating whether an AI agent's
final answer is supported by its tool
call trace.
Question: {question}
Tool call trace (tool name → result):
{trace_summary}
Final answer:
{answer}
Task: Determine if every factual claim
in the final answer is supported by
evidence in the tool call trace. An
answer is NOT supported if it contains
specific identifiers, function names,
file paths, or code-level details that
do not appear in any tool result.
Reply with exactly one word: YES if the
answer is fully supported by the trace,
or NO if any claim is not supported.
```

Per-model breakdown. Table 24 reports agreement between κ_{ground} and the trace-aware judge (*Aware agree*), trace-aware judge recall against incorrect answers (*Aware recall*), agreement between binary correctness and the trace-blind judge (*Blind agree*), and trace-blind judge precision (*Blind prec.*).

Reconciling the aggregates. The 83.1% / 72.0% aware-agreement values per prompt pool to the 77.6% reported in Section 5.3. The high agreement on Qwen3.5-397B-A17B and Gemma-4-31B-IT under the default prompt (97.5–97.8%) is driven almost entirely by joint-negative mass: both methods flag essentially all traces as ungrounded, so they trivially agree. The much lower agreement on Qwen3.5-9B (53.9% / 48.0%) is where the two methods carve up the population differently, and is the main driver of the low Cohen's κ value.

The aware-recall gap. The most striking discrepancy is the low aware-recall column on Qwen3.5-9B: many incorrect answers under that model do read files and produce entities that

Prompt	Model	Aware agree	Aware recall	Blind agree	Blind prec.
Default	`Qwen3.5-9B`	53.9%	14.5%	63.9%	11.5%
	`Qwen3.5-397B-A17B`	97.8%	—	65.3%	87.5%
	`Gemma-4-31B-IT`	97.5%	0.0%	59.1%	76.6%
	All	83.1%	14.0%	62.8%	48.4%
Strict	`Qwen3.5-9B`	48.0%	10.3%	76.2%	9.5%
	`Qwen3.5-397B-A17B`	94.7%	8.3%	57.4%	89.9%
	`Gemma-4-31B-IT`	73.4%	7.1%	68.8%	78.1%
	All	72.0%	9.6%	67.4%	62.7%

Table 24: Agreement between κ_{ground} and a trace-aware LLM judge, and between binary correctness and a trace-blind LLM judge. *Aware recall* for `Qwen3.5-397B-A17B` under the default prompt is undefined: the model has essentially zero IG mass, so the denominator of the recall computation collapses.

Property	κ_{ground}	Trace-aware LLM judge
Cost per question	≈ 0 (deterministic)	5 API calls $\times$ $\sim$256 tokens
Variance across reruns	0 (deterministic)	non-zero (5 samples per run)
Requires LLM knowledge of corpus	No	Partial (trace-aware), full (blind)
Usable at inference time as a hard stop	Yes	No (latency)
Recall on memorised repos	moderate	high
Recall on low-popularity repos	consistent	degrades with unfamiliarity
Interpretable failure report	Yes (ungrounded entities by type)	No (YES/NO only)

Table 25: Operational comparison of the two grounding evaluators.

appear in the read content, so κ_{ground} passes them as grounded. The dominant path to this state is name collision: the agent reads `lib/matplotlib/__init__.py` and the answer mentions `lib/matplotlib/__init__.py`, so the file-path entity is trivially grounded — but the agent gave the wrong file because the symbol of interest lives elsewhere. κ_{ground} has no ground-truth knowledge; it can only verify that *some* file was read that contained *some* of the answer's tokens. The LLM judge, in contrast, uses its own pretraining knowledge of common library layouts to evaluate factual correctness directly.

The trace-blind judge as a sanity check. The trace-blind judge has access only to the question and the answer, so its accuracy is bounded by its parametric knowledge of the same repositories the agent is being asked about. On the two larger models under the default prompt it reaches 76.6–87.5% precision, indicating that on canonical libraries an LLM's parametric memory alone can often identify wrong answers. On `Qwen3.5-9B` blind precision collapses to 9.5–11.5%: the model's incorrect answers are confidently wrong in ways the blind judge cannot detect, exactly the regime where a trace-grounded signal like κ_{ground} provides independent value.

Methodological trade-offs. Table 25 summarises the operational differences between the predicate and the judge.

The argument is not that κ_{ground} replaces an LLM judge for factual-accuracy assessment, but that it provides a zero-cost, deterministic, trace-grounded signal computable at runtime that surfaces a complementary failure mode, ungrounded confidence, which standard accuracy metrics systematically miss. The low aware-recall on `Qwen3.5-9B` is itself a finding: a substantial population of incorrect answers is produced by a model that did read files (grounded trace) but answered about a different location than where the symbol actually lives. This failure mode is invisible to κ_{ground} and worth future investigation, e.g. via a location-aware extension of the predicate or a refusal-stratified variant that does not vacuously pass on empty answers.

Listing 2: Output of `verify_trace` for the same run. All six constraints pass, yielding $C = 1.0$ (FULL). The `tool_sequence` field is the ordered projection $\langle c_0.\text{name}, \ldots \rangle$ used by ordering predicates.


```
{
  "compliance_score": 1.0,
  "compliance_label": "FULL",
  "details": "All constraints
      satisfied.",
  "tool_sequence": ["tool_a", "tool_b",
      "tool_b", "tool_b", "tool_d"],
  "constraints": [
    {
      "name": "tool_a_called",
      "layer": "L1",
      "weight": 2.0,
      "passed": true,
      "detail": "\"tool_a\" was called
          (call #1)."
    },
    {
      "name": "tool_a_before_branches",
      "layer": "L4",
      "weight": 3.0,
      "passed": true,
      "detail": "\"tool_a\" (call #1)
          before \"tool_d\" (call #5)."
    },
    {
      "name": "branch_call_count",
      "layer": "L5",
      "weight": 3.0,
      "passed": true,
      "detail": "Predicate returned
          True (3 branch calls,
          expected >= 3)."
    },
    {
      "name": "tool_b_for_even",
      "layer": "L5",
      "weight": 3.0,
      "passed": true,
      "detail": "\"tool_b\" called 3
          time(s); expected >= 3."
    },
    {
      "name": "tool_b_before_tool_d",
      "layer": "L3",
      "weight": 4.0,
      "passed": true,
      "detail": "\"tool_b\" (call #2)
          before \"tool_d\" (call #5)."
    },
    {
      "name": "tool_d_called",
      "layer": "L1",
      "weight": 2.0,
      "passed": true,
      "detail": "\"tool_d\" was called
          (call #5)."
    }
  ]
}
```


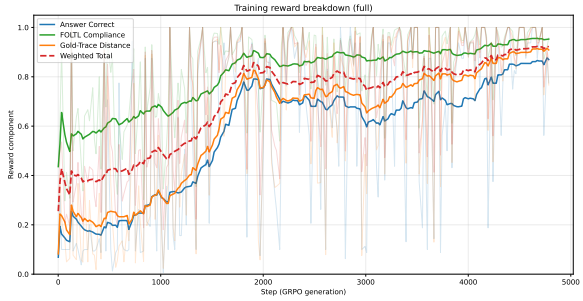

Figure 3: Training-rollout reward components across GRPO generations, shown for comparison with Figure 4. The training and evaluation trajectories track each other closely.

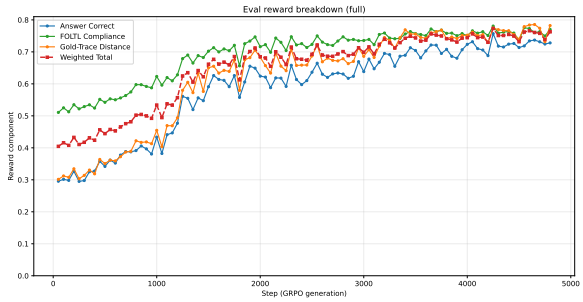

Figure 4: Evaluation reward components across GRPO generations. Compliance $\bar{C}$, answer correctness, gold-trace distance reward R_{dist}, and weighted total reward all improve jointly over training.

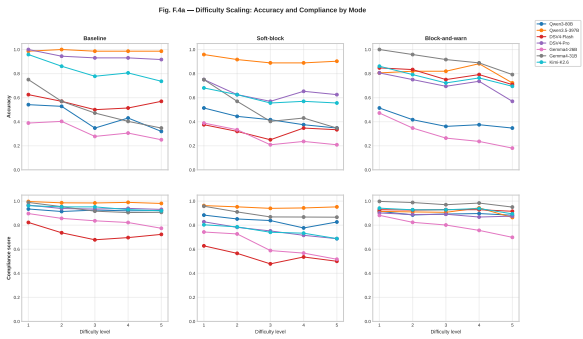

Figure 5: Difficulty scaling curves. Top row: accuracy versus difficulty level D1–D5; bottom row: compliance versus difficulty level. Columns correspond to the three enforcement settings (baseline, soft-block, block-and-warn).

Figure 6: Per-template compliance (top row) and accuracy (bottom row) across enforcement settings (columns). Each heatmap shows the mean score per (model, template) cell. Green denotes scores at or near 1.0; red denotes scores at or near 0.

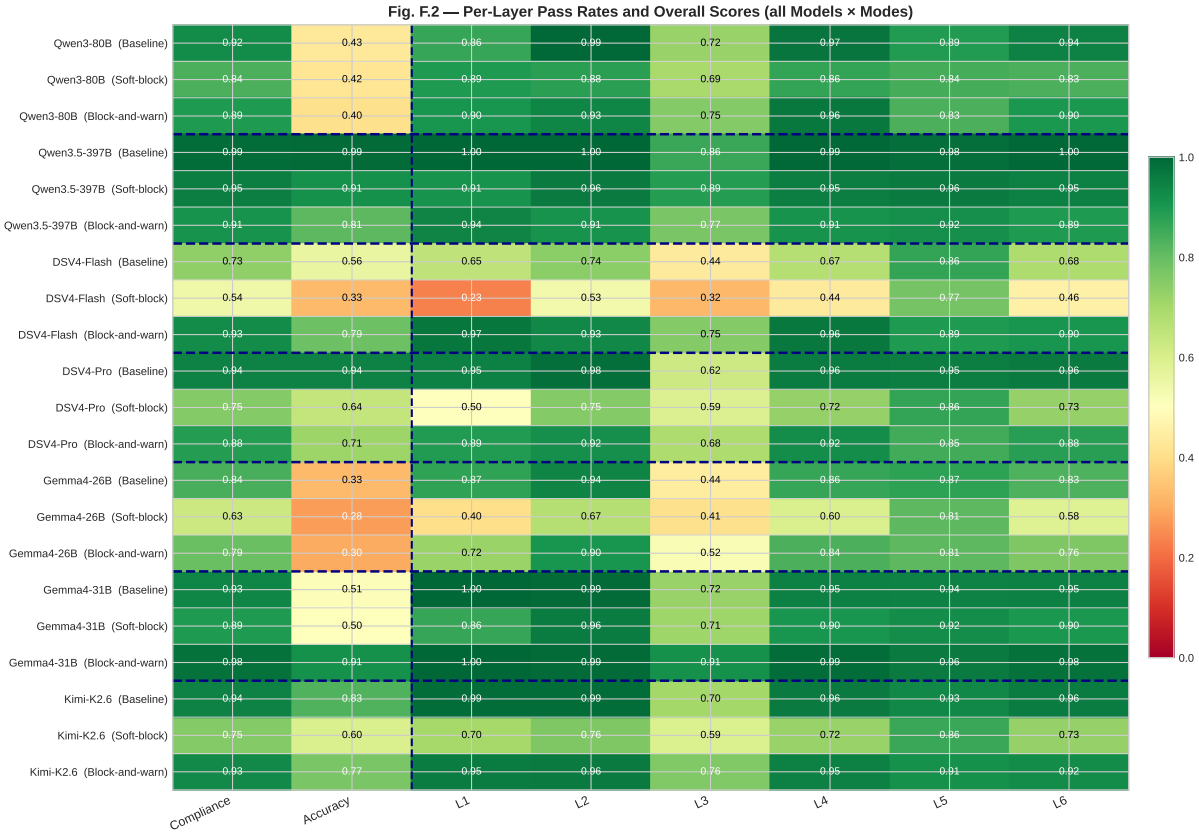

Figure 7: Per-layer pass rates by model and setting. Rows are (model, setting) pairs; columns are the two summary metrics (compliance, accuracy) followed by L1–L6 pass rates. Dashed navy lines separate model groups; the vertical dashed line separates summary columns from layer columns. Layers: L1 branch, L2 argument extraction, L3 global sequence, L4 pair-order, L5 call-count, L6 exact args.

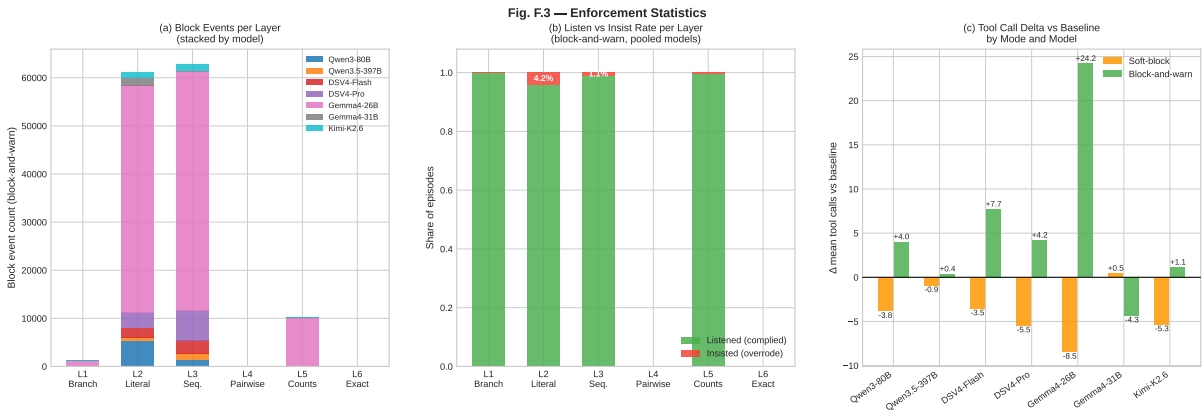

Figure 8: Enforcement statistics under block-and-warn. **(a)** Enforcement events per layer, stacked by model. **(b)** Listen versus insist rates per layer. **(c)** Tool-call deltas versus baseline.