AGENTLTL: A TRACE-VERIFICATION FRAMEWORK FOR MEASURING, ENFORCING, AND TRAINING PROCEDURAL COMPLIANCE IN TOOL-USING LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-using LLM agents are typically evaluated by final-answer correctness or by LLM judges. Neither captures how an answer was produced, yet in safety-critical settings the procedure is itself part of correctness. In this paper, we introduce AGENTLTL, a specification language derived from first-order linear temporal logic over finite traces (FO-LTL$_f$), which expresses procedural rules over agent traces and yields a deterministic, judge-free compliance score. A single specification serves two purposes. In harnessing, constraints either score completed traces offline or gate tool calls online, by checking each prefix before execution. In finetuning, the same compliance signal acts as a dense reward. We evaluate both on our novel dedicated benchmark ProcFlow covering ordering, branching, iteration, and grounding, and on other several established tool-calling benchmarks. Online harnessing raises procedural compliance across models without degrading task performance, and finetuning with compliance-aware rewards improves held-out performance on ProcFlow while also increasing accuracy over the base model on every external benchmark we evaluate. Together, these results show that compliance is a transferable measure of procedural structure: it generalizes beyond the procedures seen during training and can be leveraged directly for efficient runtime enforcement.

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 Finite First-Order Linear Temporal Logic (FO-LTL$_f$) 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. We make three contributions. (1) We introduce AGENTLTL, a framework for compiling procedural specifications into executable constraints, together with a compliance score that separates procedural adherence from task correctness. (2) We construct ProcFlow: a novel benchmark of 12 procedural templates and evaluate enforcement across four established tool-calling benchmarks and six language models, showing that online enforcement substantially improves compliance while largely preserving task accuracy, and characterizing which failure types can be recovered. (3) We show that compliance is an effective training signal: GRPO finetuning generalizes to unseen procedures and tool names and transfers to external benchmarks with real tools and unrelated domains, outperforming supervised finetuning on held-out tasks. AGENTLTL, the benchmark, training corpus, models, and scripts are available here.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

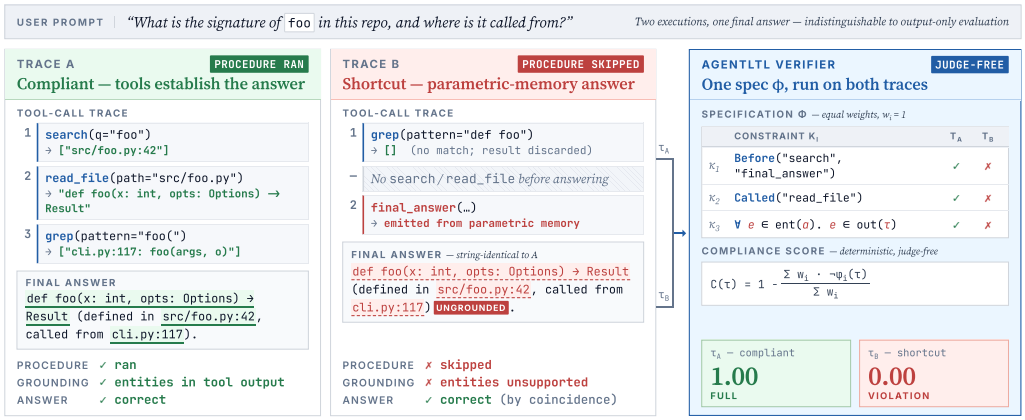


Figure 1: **The problem, and AGENTLTL as a solution.** Two runs of a code-QA agent on identical prompts. *Left (Trace A):* the agent calls the following tools: `search`, `read_file`, and `grep`, grounding its answer in retrieved evidence. *Middle (Trace B):* the same answer, emitted from parametric memory with no retrieval. Final-answer accuracy scores both correct and is blind to the difference. *Right:* the AGENTLTL procedure P separates them, checking ordering (κ_1: `search` precedes the answer), call presence (κ_2: `read_file` is invoked), and grounding (κ_3: every cited entity has a witness in a tool result). Trace A satisfies all three; Trace B violates each.

2 RELATED WORK

Tool-augmented LLM agents. Research on tool-augmented LLM agents has evolved from paradigms for interleaving reasoning and tool use, such as ReAct Yao et al. (2023), toward finetuning methods Schick et al. (2023); Patil et al. (2024), larger and more diverse tool inventories Liu et al. (2025); Li et al. (2023), and benchmarks targeting agent reliability such as τ-bench Yao et al. (2024), AgentBoard Ma et al. (2024), and BFCL Patil et al. (2025). Recent benchmarks evaluate agents in structured workflows, including τ^2-bench Barres et al. (2025), OSWorld Xie et al. (2024b), WorkBench Styles et al. (2024), and MedAgentBench Jiang et al. (2025). These benchmarks mainly score final states or returned values and expose a compositionality gap: agents handle atomic calls better than composed procedures Mohammadi et al. (2025). AGENTLTL instead scores whether the full trace follows the required procedure. For existing benchmarks, we recover this procedure per task while leaving their prompts and graders unchanged.

Runtime verification and pre-execution guardrails. Runtime verification monitors events during execution and incrementally checks a specification Basin et al. (2017; 2012), with a trade-off between expressiveness and monitoring cost Gorostiaga & Sánchez (2018); Yu et al. (2023); Gorostiaga (2022). For agent systems, prior work uses DSL rules Wang et al. (2026), auditing Yuan et al. (2026), SMT checks Winston et al. (2026), Datalog Palumbo et al. (2026), rule circuits Chen et al. (2025), or a second model Ta et al. (2026). Agent-C Kamath et al. (2025) is closest to our setting: it specifies ordering constraints over tool traces and uses SMT-based constrained decoding to block non-compliant calls. These methods primarily ask whether the next call is admissible, giving a binary prefix-level decision. AGENTLTL instead scores completed traces, supports constraints that require the full trace, and provides the same graded signal for evaluation, enforcement, and training.

Finite-trace and first-order temporal logic. Agent executions are naturally modeled as discrete-time traces, for which propositional LTL Pnueli (1977) is a standard specification language. However, LTL is limited to Boolean atomic propositions and is therefore ill-suited to data words. For example, properties such as "every system that was started must eventually be stopped" require predicates, variables, and quantification over entities in the trace. First-order temporal logic addresses this limitation by allowing predicates over trace data together with free and bound variables and quantification. For finite traces, related work includes LTL$_f$ for finite-horizon tasks Giacomo & Vardi (2013), first-order temporal logic Artale et al. (2022), temporal description logics Artale et al. (2018), LTL$_f$ modulo theories Gianola & Gigante (2022); Bombardelli et al. (2024), and data-word monitoring Bojańczyk

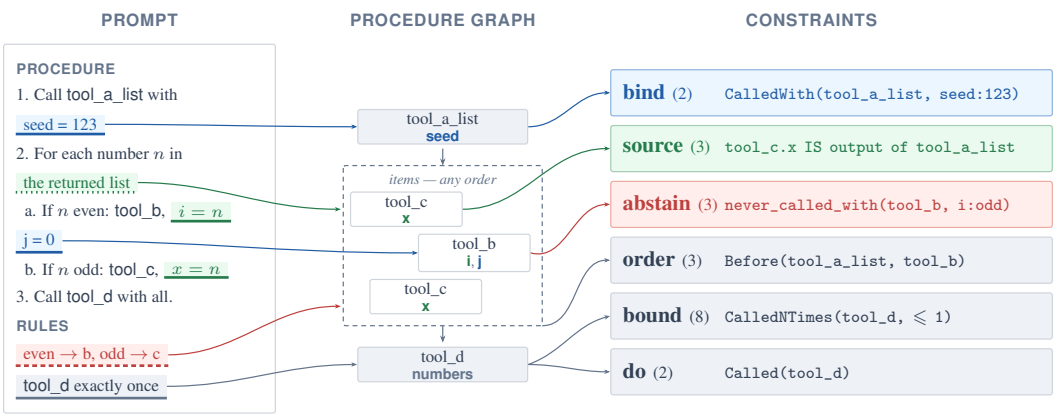


Figure 2: **From prompt to AGENTLTL procedure.** A `fan_out_fan_in` instance (D1, seed 123) from ProcFlow. **(Left)** Only `seed` $= 123$ and `j` $= 0$ are stated by the prompt; every other argument must be obtained by calling a tool. **(Middle)** The compiled procedure graph, nodes carrying their slots. Fan-out siblings share no data edge, so no ordering is emitted among them and any permutation of the item settings is accepted. **(Right)** The six constraint families compiled from that graph. Parentheses give the number of constraints in each family; we show one representative formula per family, 6 of the 21 emitted. Blue marks values the prompt states; green marks values sourced from the returned list. Colour tracks each phrase to the slot it fills. Values the agent must compute appear as provenance or prohibition, never as literals.

et al. (2011); Xu et al. (2026). AGENTLTL builds on this first-order, finite-trace perspective for tool-using agents, quantifying over entities extracted from the trace itself and the execution environment, to express procedural and provenance constraints, such as requiring that a consumed value was produced by an earlier tool call.

From natural language to temporal specifications. For translation, existing NL-to-LTL methods use synthetic data, constrained decoding, decomposition, voting, verification, code generation, and consistency checking Pan et al. (2023); Xu et al. (2025); Wu et al. (2025); Grigorev et al. (2025); Rabiei et al. (2025); Andresel et al. (2026); English et al. (2025a). Generated specifications can still be temporally ambiguous, syntactically invalid, or inconsistent, and performance can degrade as formulas grow Xu et al. (2025); Rabiei et al. (2025); English et al. (2025b). AGENTLTL avoids direct NL-to-LTL generation: the model proposes a typed procedure graph over tool schemas, while the compiler generates ordering and provenance constraints. This removes formula-level generation errors and limits ambiguity to step and argument identification, which can be checked against the schemas.

Process conformance checking. Conformance checking compares event logs with prescriptive workflow models van der Aalst (2016). DECLARE Pesic et al. (2007) provides a declarative LTL_f framework for finite traces, while later work studies online enforcement and streaming conformance Klessascheck et al. (2024); Stertz et al. (2020). Agent traces differ because events are generated by a stochastic policy that can invent arguments, skip steps, or alter the woFrkflow. AGENTLTL adapts conformance checking to this setting by treating tool calls as typed events, grounding constraints in explicit entities, and compiling a specification per task. Its graded verdict can then support both enforcement and training.

Characterisation of agent failures. MAST Cemri et al. (2025) identifies 14 failure modes from more than 1600 multi-agent traces, while TravelPlanner Xie et al. (2024a) reports argument errors, dead loops, and uncorrected early hallucinations. Both rely on LLM judges or hand-written matching for evaluation. AGENTLTL instead maps failures to explicit constraints over ordering, branching, iteration, and argument grounding. This provides a procedural failure signal that can be used for evaluation, enforcement, and training.

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 q_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 tuple $c = (n_i,\, a_i,\, r_i)$. The concatenation of these tuples form our execution traces $\tau = \langle c_0, \ldots, c_{n-1} \rangle$ on which AGENTLTL operates on . Additionally a metrics record μ storing aggregate statistics over the trace, such as token and tool call counts is created, but does not play a role in the formal evaluation.

3.2 FORMULA LANGUAGE

AGENTLTL specifies properties over executions traces using a fragment of first-order linear temporal logic (FO-LTL$_f$, Gabbay et al. (2005); Artale et al. (2019)). 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." The 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.

Predicates. Predicates describe observable properties of tool calls, such as which tool was called, whether its arguments match a given value, or whether a returned value satisfies a condition. For example, $\texttt{IsCalledWith}(\texttt{search}, \{q : \text{“foo”}\})$ holds iff `search` is called with $q =$ “foo”. Predicates may also inspect returned values or tool arguments, allowing specifications to express domain-specific conditions over execution traces.

Temporal operators. AGENTLTL is built from three temporal operators over finite traces: strong next $\mathsf{X}[!]$, which requires a next step and that the formula holds there; weak next X, which is also true when no next step exists; and until U, where $\psi \mathsf{U} \phi$ holds when ϕ eventually holds and ψ holds at all preceding steps. We use standard abbreviations such as F (eventually) and G (always). For example, $(\neg \texttt{IsCalledWith}(\texttt{write}, \{o : \text{“fii”}\}))\mathsf{U}(\texttt{IsCalledWith}(\texttt{search}, \{q : \text{“foo”}\}))$ requires that “fii” is not written before “foo” is searched for.

Data quantifiers. Quantifiers range over a finite domain of constants extracted from the trace, the specification, and, when needed, the tool set $\mathcal{T}$ and execution environment $\mathcal{E}$. This domain contains values relevant to the execution, including tool names, arguments, and return values, while keeping formula evaluation tractable. A formula is evaluated under an assignment $\mathfrak{a}$ mapping variables to domain constants; quantified formulas are checked by enumerating the finite domain. For example, $\forall x(\mathsf{F}(\texttt{IsCalledWith}(\texttt{touch}, \{o : x\})) \rightarrow \mathsf{F}(\texttt{IsCalledWith}(\texttt{rm}, \{o : x\})))$ requires every file created by `touch` to eventually be removed by `rm`.

3.3 COMPLIANCE SCORING

A procedure P is a set of constraints $G_P = \{(\kappa_i, w_i)\}$, where each κ_i is an AGENTLTL constraint and $w_i > 0$ as its associated weight denoting 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 score is

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

3.4 USING THE COMPLIANCE SCORE

We introduce the usage of our compliance score. First, harnessing: the same procedure can score a trace offline or block tool calls online. Second, finetuning: because the score is numerical, 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. In *offline mode*, the full constraint set G_P is evaluated on a completed trace. The framework reports the overall score as defined in Definition 1 and the result of each constraint, supporting benchmarking and post-hoc analysis. In *online mode*, the harness evaluates a prefix $\tau^{\leq i} = \langle c_0, \ldots, c_i \rangle$ before each tool execution. The proposed call is appended to the prefix and checked against the enforceable subset of G_P. The call executes only if it does not definitively violate an enforceable constraint.

Some constraints cannot be decided from a prefix. Constraints with a finite bad prefix can be detected online, whereas liveness constraints whose violation is established only when execution ends must be deferred to offline evaluation. We further distinguish enforceable constraints by when sufficient evidence becomes available: at_call constraints can be checked using the proposed call and preceding trace, while on_evidence constraints require evidence such as the result of an earlier call. Each enforceable constraint also has a severity level that determines the response, ranging from logging to corrective feedback or blocking execution.

Training reward implementation. $C(\tau, G_P) \in [0, 1]$ is used as a reward. The decomposition over constraints provides dense credit assignment: traces receive step-wise reward even when only some constraints are satisfied, rather than collapsing to the same score.

3.5 FROM PROCEDURE GRAPH TO PROCEDURE SPECIFICATION

To compile procedures into FO-LTL$_f$ contraints, we propose representing a procedure as directed acyclic graphs (DAGs), where nodes denote procedural steps and edges capture data dependencies, conditions, and ordering. We can translate natural language specifications into such a graph. We then deterministically compile it into an AGENTLTL procedure specification G_P. Figure 2 illustrates this process: a procedural description is represented as a graph, whose dependencies and requirements are emitted as trace constraints. Appendix C details the full graph-induction procedure.

Definition 2 (Procedure graph)**.** A procedure graph on a tool set $\mathcal{T}$ is a DAG $\mathcal{G} = (N, E)$ whose nodes are procedural steps $u = (\rho, m, t, S)$ with $t \in \mathcal{T}$, comprising a role ρ, a modality m, a tool t, and a set of slots S, one per argument the step must carry.

- *Roles.* OBSERVE, a read whose result later steps consume; ACT, a mutation; or ABSTAIN, an action that must not occur.
- *Modalities.* MUST, MUST-NOT, or CONDITIONAL, the last carrying a guard.
- *Slots.* A slot is a pair (p, s) of an argument path p and a source s: either a value fixed by the task or the tool schema, or a reference to the step that produces it.
- *Edges.* Three kinds,

$$\mathsf{data}(u \to v,\, p), \qquad \mathsf{guard}(u \to w,\, c), \qquad \mathsf{order}(u \to v),$$

  respectively, a value produced by u and consumed as argument p of v; a condition c on u's result determining whether w applies; and explicit sequencing.

Compilation translates node-level requirements, such as required or forbidden actions and argument constraints, together with graph structure into trace constraints. A data-flow edge induces both an ordering constraint and a provenance constraint: a value consumed by a step must be produced by its designated predecessor. Guard edges similarly require the observations needed to determine the applicable branch. Ordering is derived only from dependencies and explicit sequencing, so independent steps remain unconstrained and may occur in either order.

Compilation operates on the procedure structure rather than on a reference execution. Values that the agent must compute are therefore represented as dependencies rather than fixed literals. The resulting specification can thus accept different valid executions. Details of the graph representation and constraint emission rules are given in Appendix D.

4 Experimental Settings

We organize our experiments around four research questions: **RQ1:** Is procedural compliance a distinct signal from task correctness, or merely a proxy for it? **RQ2:** Can online enforcement of FO-LTL$_f$ constraints raise procedural compliance without degrading task accuracy? **RQ3:** Which procedural failures can online enforcement repair, and which are beyond its reach? **RQ4:** Can a compliance-based reward teach procedure following that generalizes to unseen patterns, unseen tools, and external benchmarks?

4.1 Evaluation Environments

We evaluate on five environments: our novel benchmark ProcFlow and four established procedure-centric tool-calling benchmarks, BFCL (Patil et al., 2025), τ-bench Yao et al. (2024), Work-Bench (Styles et al., 2024), and MedAgentBench (Jiang et al., 2025). The established benchmarks use real tools and concern a variety of different domains. For ProcFlow, we construct the procedure graph directly. For the established benchmarks, we first induce a graph for each task and then apply the same compilation process. All evaluation after compilation is identical across environments. More information in Appendix F.

ProcFlow. ProcFlow isolates procedure following from domain knowledge. It contains twelve templates covering five structural types. Each template has a production mode for evaluation and a simplified mode for training. An instance is a $(\text{template}, \text{difficulty}, \text{seed})$ tuple that deterministically generates a prompt, gold trace, expected answer, and procedure graph. Difficulty varies one structural parameter, such as a loop bound or list length. The compiled graph defines the constraints used to evaluate correctness and compliance on the same run.

Inducing the procedure graph. For the established benchmarks, we induce the procedure graph with Qwen3.6-27B (Qwen Team, 2026). The model proposes the steps, their roles, inputs, outputs, and stated arguments. Edges and constraints are then derived from the resulting graph using the same procedure as for ProcFlow. Induction uses only information available to the agent and has no access to reference answers, gold traces, or graders.

4.2 Harnessing Agents

Evaluation settings. We compare five evaluation settings. *Baseline* runs the agent without constraints and scores its final trace offline. *Block-and-warn* blocks violating calls but allows the agent to retry with the same arguments. *Persistent-block* blocks violations without allowing overrides. *Soft-block* also blocks overrides and terminates after three violations of the same constraint. A call is blocked only if it newly introduces a violation. As a constraint-free comparison, *Reviewed* implements the Reinforced Agent of Ta et al. (2026). It reviews each proposed call and either returns feedback or selects among sampled candidates. Compliance is scored offline on the final trace.

Models and protocol. We evaluate six zero-shot models: Gemma-4-31B, Gemma-4-12B, Qwen-3.8-27B, Qwen-3.6-27B, Qwen3-4B-Instruct, and DeepSeek-V4.1-Flash. All use the same agent loop. On ProcFlow, we run three trials per $(\text{model}, \text{template}, \text{difficulty}, \text{seed})$ configuration with seeds 42 and 123, giving 360 runs per model and setting. Task correctness is exact integer match. On the established benchmarks, we use the benchmark-given graders. Compliance is evaluated separately from task correctness.

4.3 Compliance-Supervised Finetuning

Training corpus. We construct 300 training examples from 15 simplified templates from ProcFlow. Examples use 3–7 tool calls from four tools and cover sequential, branching, fan-out, and multi-

Table 1: Compliance and accuracy per benchmark, averaged over six models. Compliance is $C(\tau, G_P)$ (Definition 1), scored post hoc against the same specification in every setting. *Base* = Baseline (unconstrained); *soft* = Soft-block; *b&w* = Block-and-warn; *pers.* = Persistent-block; *rev.* = Reviewed, the constraint-free Reinforced Agent of Ta et al. (2026).

	Compliance $C(\tau, G_P)$					Accuracy (%)				
	Base	soft	b&w	pers.	rev.	Base	soft	b&w	pers.	rev.
BFCL	.903	+.010	**+.010**	+.009	±.000	43.3	−.8	**+.5**	−.8	+.5
τ-bench	.882	**+.022**	+.010	−.002	+.001	41.6	−7.5	−2.1	−.7	**+.1**
WorkBench	.908	**+.051**	+.034	+.050	±.000	69.2	−3.7	+.9	−4.3	**+1.7**
MedAgentBench	.845	**+.147**	+.135	+.145	+.009	18.9	+22.3	**+23.5**	+22.7	+5.0
Mean	.885	**+.057**	+.047	+.050	+.002	43.2	+2.6	**+5.7**	+4.2	+1.8
ProcFlow	.942	+.031	+.022	**+.032**	+.011	64.9	+2.6	+2.9	**+3.6**	+3.5

stage procedures. We vary tool permutations and optionally add one prefix or suffix call to reduce sequence memorization. The `loop_termination` and `branch_selection` templates are held out for unseen-pattern evaluation.

Reward. We train Qwen3-4B-Instruct with GRPO (Shao et al., 2024) using a weighted reward $[0.5, 0.25, 0.25]$ for compliance $C(\tau, G_P)$, answer correctness, and gold-trace similarity. Incorrect valued but correctly formatted answers receive 0.1 correctness reward. Gold-trace similarity is $R_{\text{dist}}(\tau, \tau^{)} = \exp(-d(\tau, \tau^{)})$, where d is weighted Levenshtein distance.

Ablations and SFT baseline. We compare the full reward $c{+}a{+}g$ with three ablations: $c{+}a$, c, and g, corresponding to compliance plus answer correctness, compliance only, and gold-trace similarity only. We also train SFT baselines on gold traces for one, two, and three epochs.

Training setup. GRPO uses LoRA (r=8, α=16) on `q_proj` and `v_proj`, one epoch, fp32 precision, 16 rollouts per prompt, β=10^{-3}, and a round-robin-no-repeat curriculum. All reward ablations use the same configuration. SFT uses the same LoRA setup for one, two, or three epochs.

Evaluation splits. We evaluate on four held-out splits from ProcFlow, the full production benchmark, and the four established benchmarks. *Unaugmented* (83 instances) uses new parameter values for the training templates. *Augmented* (100) additionally includes the optional prefix and suffix calls. *Unseen-pattern* (39) uses the two held-out procedure templates with the same tool set. *Diverse-tool* (39) keeps trained procedures but replaces the tool names. The production benchmark is evaluated in full. Established benchmarks test transfer to new tasks, tools, and domains.

5 Results

5.1 Harnessing Agents

Table 1 reports compliance and accuracy per environment, averaged over six models.

Compliance. On the four established benchmarks, constraints increase mean compliance from 0.885 to 0.932–0.942. The reviewed setting reaches 0.887. Soft-block and block-and-warn improve compliance in all 24 model–benchmark cases. Persistent-block improves it in 21 cases, and Reviewed in 17. The reviewed setting also repairs fewer episodes than it disrupts on every benchmark. The gap is largest for constraint-based intervention: the intervention-to-failure ratio is 1.11× on MedAgentBench, compared with 2.40–3.45 for the constraint-based methods.

Accuracy. Enforcement has limited effect on accuracy. At least one enforcement setting matches or exceeds the baseline in 24 of 30 model–environment cases. The best enforcement setting is within one point of the baseline in 28 cases. The largest drops are for DeepSeek on WorkBench (−3.7) and Qwen3.6 on τ-bench (−2.0). Block-and-warn improves accuracy by 5.7 points on the

Table 2: Task accuracy of the base model and all reward/SFT ablations across ProcFlow and the established benchmarks. Reward terms: c = constraint compliance, a = answer correctness, g = gold-trace; $c{+}a{+}g$ is the main paper model. SFT_k = supervised finetuning for k epochs.

		GRPO (reward ablations)				SFT		
Benchmark (n)	Base	c+a+g	c+a	c	g	SFT_1	SFT_2	SFT_3
ProcFlow (Acc %)								
Unaugmented. (83)	4.8	67.5	**73.5**	69.9	68.7	38.6	62.7	68.7
Augmented (100)	12.0	46.0	**67.0**	58.0	61.0	32.0	45.0	45.0
Unseen (39)	7.7	46.2	89.7	87.2	**94.9**	17.9	15.4	23.1
Diverse-tool (39)	23.1	**61.5**	56.4	59.0	59.0	43.6	46.2	48.7
Full (360)	13.4	57.2	**86.9**	85.6	**86.9**	33.3	23.9	26.7
Established benchmarks (Acc %)								
BFCL (800)	24.7	27.5	27.8	28.2	27.5	25.6	27.6	**28.6**
τ-bench (165)	24.1	25.0	**28.0**	27.3	25.0	25.5	27.4	26.2
WorkBench (568)	26.8	**36.0**	30.0	30.6	30.4	26.1	23.9	23.4
MedAgentBench (300)	34.9	37.1	36.2	**37.6**	36.6	27.2	22.5	22.1

established benchmarks and by 2.9 points on ProcFlow. On the established benchmarks, 85% of blocks on already-solved episodes correspond to constraints violated by the correct trace, indicating that answer-based graders accept traces that do not follow the procedure.

Gains by failure type. We classify each wrong baseline episode by the constraints it violates. Enforcement can prevent surplus calls, wrong arguments, and forbidden branches, but it does not recover missing calls. On ProcFlow, enforcement recovers 31 points on instances with only refusable failures, but none when all failures are missing calls. MedAgentBench has the largest share of refusable failures and shows the best accuracy gain of block-and-warn harnessing. None of the block corresponds to a constraint that requires an additional call.

5.2 Compliance-Supervised Finetuning

Table 2 reports task accuracy for each finetuned and ablated model accross different evaluation environments.

Held-out performance. All GRPO variants outperform the base model on every split from ProcFlow. The full reward reaches 46.2% on unseen patterns, compared with 7.7% for the base model. It reaches 61.5% on Diverse-tool.

Reward ablations. Removing gold-trace supervision does not reduce performance. The $c{+}a$ variant is strongest on three of five ProcFlow splits. Compliance alone reaches 87.2% on unseen patterns. SFT performs below GRPO on every split, reaching only 23.1% on unseen patterns. More SFT epochs do not close this gap.

Gains by failure type. On ProcFlow, GRPO recovers 88–89 points on instances with only refusable failures. It recovers 74–79 points on mixed instances that include a missing call. SFT recovers at most 26 points in any failure class.

Transfer. All GRPO variants improve over the base model on all four established benchmarks. The full reward improves accuracy by 9.2 points on WorkBench, 2.8 on BFCL, 2.2 on MedAgentBench, and 0.9 on τ-bench. On WorkBench, the full reward reaches 36.0%, compared with 30.0% for $c{+}a$ and 30.4% for g. SFT falls below the base model on WorkBench and MedAgentBench, with larger degradation at higher epoch counts.

6 DISCUSSION

Compliance and correctness diverge (RQ1). Compliance is a distinct signal, not a proxy for correctness. On established benchmarks, 85% of blocks on accepted episodes correspond to constraints the accepted trace violates. Some may stem from errors in the induced specifications, but the pattern matches the gap described in Section 1: the answer is right, the procedure is not. Answer-only benchmarks therefore overstate how often agents behave as intended.

Explicit specifications make enforcement precise (RQ2). The reviewed setting intercepts the same calls as our harness but, lacking a specification, cannot tell failing episodes from correct ones and disrupts more than it repairs. Explicit constraints are what let enforcement raise compliance at little cost in accuracy.

Online enforcement can only refuse (RQ3). A prefix gate blocks orderings, arguments, and forbidden branches, but cannot add a call the agent never makes: missing calls violate "eventually φ" obligations, which no single step violates. Refusability is also not sufficient. On MedAgentBench task 8, enforcement blocks calls in every Gemma episode but recovers none, since a block says what not to do, not what to do instead. Because missing calls are common in agent failures (Cemri et al., 2025; Xie et al., 2024a), this limits what online enforcement can repair.

Finetuning teaches the procedure (RQ4). GRPO recovers failures with missing calls almost as well as refusable ones. SFT sees gold traces directly yet does worse and degrades on transfer: it copies surface features such as tool names and argument values, whereas the compliance reward scores procedural structure, which persists across tools and domains.

The two approaches are complementary. Enforcement gives a hard guarantee but requires a precise per-environment specification, and trades measured accuracy for compliance where that specification and the grader disagree. Finetuning gives no guarantee but covers missing calls and needs no specification at deployment.

7 LIMITATIONS

Procedure graphs for the established benchmarks are induced by a medium-sized model (Qwen3.6-27B), so translation errors in the induced specifications may bias the measured effect of enforcement. Our finetuning results rely on a single small model trained on domain-free procedures, and it is unmeasured whether the observed transfer holds at larger scales. Finally, all constraints are weighted equally in the compliance score; other weightings could change both the reported compliance and the training reward.

8 CONCLUSION

In this paper, we introduce AGENTLTL, a trace-verification framework that expresses procedural rules over tool-using agent traces in a fragment of FO-LTL$_f$ and converts them into a deterministic, judge-free compliance score. We evaluate the framework ProcFlow, on a novel benchmark of twelve procedural templates designed to isolate procedure following from domain knowledge, as well as on four established benchmarks spanning six models. Our results show that compliance can serve as an effective reward for both harnessing and training. Enforcement and fine-tuning are therefore complementary, the former provides hard guarantees when a precise specification exists, while the latter teaches procedural structure that can transfer across tools and domains. We hope that AGENTLTL, together with its benchmark, training corpus, and models, will help make procedural correctness a standard component of how agents are evaluated, deployed, and trained. More broadly, we hope this work encourages further research at the intersection of formal methods and LLM alignment, where rigorous specifications and verification can complement the learning-based approaches that underpin modern language agents.

AI USE STATEMENT

In this work, we have not used generative AI tools for any tasks with required disclosure. We used generative AI tools to polish the writing, to search for relevant literature, and to write documentation for the code repository. We have reviewed all AI-assisted work and verified all cited references. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects, crowdsourcing, or the collection of personal data. ProcFlow is fully synthetic and built from domain-free procedures. All established benchmarks (BFCL, τ-bench, WorkBench, MedAgentBench) are publicly available and used under their respective licenses. MedAgentBench simulates a clinical environment. Our results on it measure procedural compliance in that simulation and should not be read as evidence that any agent is fit for clinical use.

AGENTLTL is intended to make agents behavior more auditable and controllable, particularly in safety-critical settings. Its guarantees, however, are only as good as the specifications it enforces. Incomplete or erroneous specifications, including those induced automatically as in our experiments, can block correct behavior or leave harmful behavior unconstrained. A high compliance score shows that an agent followed the specified procedure, not that the procedure itself is safe, fair, or appropriate. Harnessing and compliance-based finetuning should therefore complement human oversight rather than replace it. As with any mechanism for steering agent behavior, the same tools could in principle be used to enforce procedures that are themselves harmful. We believe the transparency of explicit, human-readable specifications makes such misuse easier to audit than implicit alternatives.

REPRODUCIBILITY STATEMENT

All code, data and trained models are released in an anonymous repository, `https://github.com/anonsubmission480/agentltl_procedural_compliance_iclr`, whose README maps each component to the paper. It contains the AgentLTL library (trace verification, the compiler from procedure graphs to constraints, and the three runtime enforcement settings and the reviewed setting baseline), the novel benchmark ProcFlow with its deterministic tools and simulator, and the 120 compiled constraint specifications used in our runs. Section 4 gives the environments, evaluation settings, models, trial counts, seeds and finetuning hyperparameters. Appendix E.4 describes how benchmark instances and their constraints are generated; the released builder regenerates the shipped specifications exactly. Appendix G.2 details the construction of the training data and the validation splits, including the tool-name permutations of the Diverse-tool split (Appendix G.3). The repository ships the training set and validation splits together with the generator that rebuilds them deterministically. It also includes the GRPO and SFT training scripts with the used configurations, and the seven LoRA adapters evaluated in the paper (GRPO c+a+g, c+a, c and g; SFT for 1, 2 and 3 epochs), so the finetuned models can be evaluated without retraining. For the established benchmarks (BFCL, τ-bench, WorkBench and MedAgentBench), we release the induced constraint specifications of every instance, so no specification has to be regenerated with an LLM, along with a Dockerized harness, a script that fetches each upstream benchmark at a pinned commit, and a serving script (vLLM, with adapters loaded natively) that runs the replicates reported in Appendix I.4. Stochastic components are seeded; the established benchmark replicates are decoded at temperature 0. Remaining sources of variance (served-model nondeterminism, the LLM user simulator of τ-bench, and API-served zero-shot models) are reported through the replicate statistics in the appendix.

REFERENCES

Medina Andresel, Cristinel Mateis, Dejan Ničković, Spyridon Kounoupidis, Panagiotis Katsaros, and Stavros Tripakis. LTLGuard: Formalizing LTL specifications with compact language models and lightweight symbolic reasoning, 2026. URL https://arxiv.org/abs/2603.05728.

A. Artale, Andrea Mazzullo, and A. Ozaki. Temporal description logics over finite traces. *Description Logics*, 2018.

A. Artale, Andrea Mazzullo, and A. Ozaki. First-order temporal logic on finite traces: Semantic properties, decidable fragments, and applications. *ACM Transactions on Computational Logic*, 25: 1–43, 2022. doi: 10.1145/3651161.

Alessandro Artale, Andrea Mazzullo, and Ana Ozaki. Do You Need Infinite Time? In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 1516–1522, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1. doi: 10.24963/ijcai.2019/210. URL https://www.ijcai.org/proceedings/2019/210.

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

D. Basin, Felix Klaedtke, and E. Zalinescu. Runtime verification of temporal properties over out-of-order data streams. pp. 356–376, 2017. doi: 10.1007/978-3-319-63387-9_18.

David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. 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*, pp. 151–167. Springer, 2012. doi: 10.1007/978-3-642-35632-2_14.

Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Transactions on Computational Logic*, 12(4):27:1–27:26, 2011. doi: 10.1145/1970398.1970403.

A. Bombardelli, A. Cimatti, A. Griggio, and S. Tonetta. Another look at ltl modulo theory over finite and infinite traces. pp. 419–443, 2024. doi: 10.1007/978-3-031-75783-9_17.

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. Why do multi-agent llm systems fail?, 2025. URL https://arxiv.org/abs/2503.13657.

Zhaorun Chen, Mintong Kang, and Bo Li. ShieldAgent: Shielding agents via verifiable safety policy reasoning, 2025. URL https://arxiv.org/abs/2503.22738.

William English, Chase Walker, Dominic Simon, and Rickard Ewetz. GinSign: Grounding natural language into system signatures for temporal logic translation, 2025a.

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

Dov M Gabbay, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. Many-dimensional modal logics: theory and applications. *Studia Logica*, 81(1), 2005.

G. De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. pp. 854–860, 2013.

Alessandro Gianola and N. Gigante. Ltl modulo theories over finite traces: modeling, verification, open questions. pp. 13–19, 2022.

Felipe Gorostiaga. Theory and practice of stream runtime verification for sequences and real-time event based systems. 2022. doi: 10.20868/upm.thesis.70504.

Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. pp. 282–298, 2018. doi: 10.1007/978-3-030-03769-7_16.

Danil S. Grigorev, A. Kovalev, and A. Panov. Verifyllm: Llm-based pre-execution task plan verification for robots. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 18489–18496, 2025. doi: 10.1109/iros60139.2025.11246320.

Yixing Jiang, Kameron C. Black, Gloria Geng, Danny Park, James Zou, Andrew Y. Ng, and Jonathan H. Chen. Medagentbench: A realistic virtual ehr environment to benchmark medical llm agents, 2025. URL https://arxiv.org/abs/2501.14654.

Adharsh Kamath, Sishen Zhang, Calvin Xu, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Enforcing temporal constraints for LLM agents, 2025. URL https://arxiv.org/abs/2512.23738.

Finn Klessascheck, Tobias Knoche, and Luise Pufahl. 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, 2024.

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

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, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents, 2025.

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

Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In Cynthia Dwork (ed.), *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pp. 377–410. ACM, 1990. doi: 10.1145/93385.93442. URL https://doi.org/10.1145/93385.93442.

Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. Evaluation and benchmarking of LLM agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 6129–6139. ACM, August 2025. doi: 10.1145/3711896.3736570.

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

Jiayi Pan, Glen Chou, and D. Berenson. Data-efficient learning of natural language to linear temporal logic translators for robot task specification. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11554–11561, 2023. doi: 10.1109/icra48891.2023.10161125.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Shishir G. Patil et al. 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*, 2025.

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

Amir Pnueli. The temporal logic of programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57, 1977. doi: 10.1109/SFCS.1977.32.

Qwen Team. Qwen3.6-Plus: Towards real world agents, April 2026. URL https://qwen.ai/blog?id=qwen3.6.

Behrad Rabiei, Mahesh Kumar A. R, Zhi-Rui Dai, Surya L.S.R. Pilla, Qiyue Dong, and N. Atanasov. Ltlcodegen: Code generation of syntactically correct temporal logic for robot task planning. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 19240–19247, 2025. doi: 10.1109/iros60139.2025.11246351.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Florian Stertz, Juergen Mangler, and S. Rinderle-Ma. Temporal conformance checking at runtime based on time-infused process models. *ArXiv*, abs/2008.07262, 2020. doi: 10.48550/arxiv.2008.07262.

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. Workbench: a benchmark dataset for agents in a realistic workplace setting, 2024. URL https://arxiv.org/abs/2405.00823.

Anh Ta, Junjie Zhu, and Shahin Shayandeh. 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*, 2026. Apple Machine Learning Research.

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

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

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

Yi Wu, Zikang Xiong, Yiran Hu, S. Iyengar, Nan Jiang, Aniket Bera, Lin Tan, and Suresh Jagannathan. Selp: Generating safe and efficient task plans for robot agents with large language models. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2599–2605, 2025. doi: 10.1109/icra55743.2025.11128420.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. Travelplanner: A benchmark for real-world planning with language agents, 2024a. URL https://arxiv.org/abs/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. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024b.

Shao-Jun Xu, Xu-Sheng Luo, Yu-Tong Huang, Le-Tian Leng, Ruixuan Liu, and Changliu Liu. Nl2hltl2plan: Scaling up natural language understanding for multi-robots through hierarchical temporal logic task specifications. *IEEE Robotics and Automation Letters*, 10:10482–10489, 2025. doi: 10.1109/lra.2025.3598648.

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

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

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

Bin Yu, X. Lu, Ming Lei, Chu Chen, M. Wang, Cong Tian, and Z. Duan. Adaptively parallel runtime verification based on distributed network for temporal properties. *Parallel Comput.*, 117:103034, 2023. doi: 10.1016/j.parco.2023.103034.

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

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 3-tuple

$$c_i = (n,\ a,\ r),$$

with $n \in \mathcal{T}$ the tool name, $a : \text{Key} \to \text{Value}$ the argument mapping, and $r \in \text{Value}$ the (possibly structured) result. We write $c.\mathsf{name}, c.\mathsf{args}, c.\mathsf{result}$ for the three projections. When a trace τ satisfies a formula φ, we write $\tau \models \varphi$, 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:

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

This record is created for convenience and is not necessary for formal semantics. 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 is fixed and fully observable during evaluation, so the semantics below are deterministic regardless of any nondeterminism in the underlying agent or tools.

A.2 SPECIFICATION LANGUAGE

We mostly follow standard FO-LTL$_f$ syntax and semantics as found in for instance , however we use the more standard non-strict until operator together with the strong and weak next operators.

A.3 SYNTAX

The alphabet of FO-LTL$_f$ consists of a set of predicates N_P and a set of constants N_C. Together with the variables in Var, quantification, the boolean and temporal operators, FO-LTL$_f$ formulas are built by the grammar:

$$\varphi \coloneqq P(\bar{t}) \mid \neg\varphi \mid \varphi_0 \wedge \varphi_1 \mid \mathsf{X}\varphi \mid \mathsf{X}[!]\varphi \mid \varphi_0 \mathsf{U} \varphi_1 \mid \exists x \varphi$$

where $P \in N_P$ denotes a predicate of arity $\mathtt{ar}(P) \in \mathbb{N}$, $\bar{t} = \left(t_1, \ldots, t_{\mathtt{ar}(P)}\right)$ denotes a tuple of terms and X, $\mathsf{X}[!]$ and U are the temporal operators.

A.3.1 SEMANTICS

A (finite) first-order temporal interpretation is a structure $\mathfrak{M} = (D, (I_n)_{n \in [0, L-1]})$, where $L \in \mathbb{N}$ is the length of the trace and each I_n is a classical first order interpretation with domain D: I_n defines the predicates in the n-th step, while constants do not change their signification over time. We follow the constant domain interpretation, that is D is also the same for all time points and does not change. Finally we need to define an assignment in $\mathfrak{M}$, a function denoted $\mathfrak{a}$ mapping terms to values in D. We have $\mathfrak{a}(t) = \mathfrak{a}(x)$ if $t = x$, and $\mathfrak{a}(t) = c$ if $t = c \in N_I$. This is extended to tuples in the natural way. Satisfaction of a formula φ in $\mathfrak{M}$ at time point n under assignment $\mathfrak{a}$ is inductively defined as

$\mathfrak{M}, n \models^{\mathfrak{a}} P(\bar{t})$	iff	$P(\mathfrak{a}(\bar{t})) = \top$
$\mathfrak{M}, n \models^{\mathfrak{a}} \neg\varphi$	iff	not $\mathfrak{M}, n \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}, n \models^{\mathfrak{a}} \varphi_0 \wedge \varphi_1$	iff	$\mathfrak{M}, n \models^{\mathfrak{a}} \varphi_0$ and $\mathfrak{M}, n \models^{\mathfrak{a}} \varphi_1$
$\mathfrak{M}, n \models^{\mathfrak{a}} \mathsf{X}\varphi$	iff	$\mathfrak{M}, n+1 \models^{\mathfrak{a}} \varphi$ or $n = L-1$
$\mathfrak{M}, n \models^{\mathfrak{a}} \mathsf{X}[!]\varphi$	iff	$\mathfrak{M}, n+1 \models^{\mathfrak{a}} \varphi$ and $n < L-1$
$\mathfrak{M}, n \models^{\mathfrak{a}} \varphi_0 \mathsf{U} \varphi_1$	iff	$\exists m, n \leq m < L\colon\ \mathfrak{M}, m \models^{\mathfrak{a}} \varphi_1$ and $\forall i, n \leq i < m\colon\ \mathfrak{M}, i \models^{\mathfrak{a}} \varphi_0$
$\mathfrak{M}, n \models^{\mathfrak{a}} \exists x \varphi$	iff	there exists $\mathfrak{a}'$ only differing from $\mathfrak{a}$ on x and $\mathfrak{M}, n \models^{\mathfrak{a}'} \varphi$

From these rules we define in the usual way the finally / eventually operator F and the globally operartor G.

A.3.2 TRACES, PREDICATES AND SYNTACTIC SUGAR

As stated in section A.1.1, traces are sequences aggregated tool call information. Each element is a named three-tuple consisting of the name of the tool called, a key $\rightarrow$ value argument mapping and a result value. We define the following predicates over these three-tuples $c = (n, a, r)$ where we also write c.name, c.args, and c.result to access the fields of the tuple.

$\mathtt{IsCalled}(t)$	iff	$c.\text{name} = t$
$\mathtt{IsCalledWith}(t, a)$	iff	$\mathtt{IsCalled}(t)$ and $a \sqsubseteq c.\text{args}$
$\mathtt{P}_{user}(c)$		User defined predicate over a tuple

where $a_0 \sqsubseteq a_1$ denotes that for all of the keys k in a_0, $a_0(k) = a_1(k)$. Moreover we allow for user-defined predicates over these tuples. These increase the flexibility of the approach, but play no further role in the formal definition.

Additionally, we define the following formulas as syntactic sugar:

$$
\begin{aligned}
\texttt{Called}(t) &\coloneqq \mathsf{F}(\texttt{IsCalled}(t)) \\
\texttt{CalledWith}(t, a) &\coloneqq \mathsf{F}(\texttt{IsCalledWith}(t, a)) \\
\texttt{CalledN}(t, n, =) &\coloneqq \underbrace{\mathsf{F}(\texttt{IsCalled}(t) \wedge \mathsf{X}[!]\mathsf{F}(\texttt{IsCalled}(t) \ldots \mathsf{X}[!]\mathsf{F}}_{n \text{ times}}(\texttt{IsCalled}(t) \wedge (\mathsf{XG}(\neg \texttt{IsCalled}(t))))) \\
\texttt{CalledN}(t, n, \geq) &\coloneqq \underbrace{\mathsf{F}(\texttt{IsCalled}(t) \wedge \mathsf{X}[!]\mathsf{F}(\texttt{IsCalled}(t) \ldots \mathsf{X}[!]\mathsf{F}}_{n \text{ times}}(\texttt{IsCalled}(t))) \\
\texttt{CalledN}(t, n, \leq) &\coloneqq \neg \texttt{CalledN}(t, n-1, \geq) \\
\texttt{Before}(t_1, t_2) &\coloneqq (\neg \texttt{IsCalled}(t_2)) \mathsf{U} (\texttt{IsCalled}(t_1)) \wedge \texttt{Called}(t_2) \\
\texttt{After}(t_1, t_2) &\coloneqq (\neg \texttt{IsCalled}(t_1)) \mathsf{U} (\texttt{IsCalled}(t_2)) \wedge \texttt{Called}(t_1) \\
\texttt{AllBefore}(\mathbf{T}, g) &\coloneqq \forall t \in \mathbf{T}. (\texttt{Before}(t, g)) \\
\texttt{InOrder}(\mathbf{T}) &\coloneqq \forall i, j \in [1, \texttt{len}(\mathbf{T})]. ((i < j) \rightarrow \texttt{Before}(\mathbf{T}[i], \mathbf{T}[j])) \\
\texttt{BranchCalled}(t^+, t^-) &\coloneqq \texttt{Called}(t^+) \wedge \texttt{Called}(t^-) \\
\texttt{WithinSteps}(t_1, t_2, n) &\coloneqq \mathsf{G}(\texttt{IsCalled}(t_1) \rightarrow \bigvee_{i \in [1,n]} \mathsf{X}[\mathrm{i}!](\texttt{IsCalled}(t_2))
\end{aligned}
$$

where $\mathsf{X}[\mathrm{i}!] = \mathsf{X}[!](\mathsf{X}[\mathrm{i}-1!])$ for all $i > 1$, $\mathsf{X}[1!] = \mathsf{X}[!]$.

A.3.3 Data Quantifiers and Open Predicates

Quantification ranges over a domain of values D. In this work we use a domain extractor δ to obtain the values relevant to a particular trace, formula and model : $D = \delta(\tau, \varphi, \mathcal{T}, \mathcal{E})$

The domains extracted by D are by construction finite, therefore a quantified formula can be evaluated by substituting each extracted element for the quantified variable. For example, the property "every value produced is processed" can be expressed as

$$\forall x \in D = \delta(\tau, \varphi, \mathcal{T}, \mathcal{E}).\ \mathsf{Processed}(x).$$

Equivalently, evaluation considers each grounded formula $\mathsf{Processed}(e_i)$. If the extracted domain is empty, the universal quantification is vacuously true, while an existential quantification is vacuously false.

The domain need not be partitioned according to the type of the values it contains. In particular, this permits polymorphic operations whose arguments may have different types. Type- or value-specific restrictions can instead be expressed by predicates on the quantified elements. For example,

$$\forall x \in D.\ \mathsf{Number}(x) \Rightarrow \mathsf{Processed}(x)$$

states that every numeric value in the domain is processed, without requiring a separate numeric subdomain.

Open predicates are n-ary Boolean-valued predicates

$$f : D^n \rightarrow \mathbb{B}.$$

During evaluation, the assignment function $\mathfrak{a}$ will map terms in the argument to the domain D.

A.4 Online Enforcement Mechanism

A.4.1 Prefix Evaluation and Enforceability

Online enforcement operates on the observed prefix $\tau^{\leq i} = \langle c_0, \ldots, c_i \rangle$. A finite prefix τ_{pre} is a *bad prefix* for a constraint κ if every continuation of that prefix violates κ:

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

where $\cdot$ denotes trace concatenation. Thus, once a bad prefix has been observed, no continuation can restore satisfaction of the constraint.

A constraint κ is a *safety constraint* if every violating trace has a finite bad prefix. Formally,

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

where $\tau_{\text{pre}} \preceq \tau$ denotes that τ_{pre} is a prefix of τ. This formulation captures the standard notion of safety: every violation is witnessed by a finite prefix that remains violating under every possible continuation. Consequently, such violations can be detected online as soon as the corresponding bad prefix is observed.

Liveness is deferred. Liveness properties do not admit finite bad prefixes. For example, a formula such as $\mathsf{F}\,\varphi$ can remain satisfiable after any finite prefix in which φ has not yet occurred, since a future continuation may still satisfy φ. Its violation therefore cannot be established from a finite prefix and is deferred to offline evaluation.

The distinction between safety and liveness is semantic rather than syntactic. In particular, the presence of a global operator G does not by itself make a formula a safety constraint. For example, $\mathsf{G}\,\neg\varphi$ is a safety constraint because an occurrence of φ provides a finite bad prefix. In contrast, $\mathsf{G}(\varphi\,\mathsf{U}\,\psi)$ and $\mathsf{GF}\,\varphi$ are not generally safety constraints, since their satisfaction may depend on events occurring arbitrarily far in the future.

Bounded constraints can become enforceable once their relevant window has elapsed. For example, a constraint expressed by WithinSteps can be decided once the specified number of steps has been observed, provided that failure to satisfy the constraint within that window constitutes a permanent violation. For a full classification of temporal properties see Manna & Pnueli (1990).

A.4.2 Runtime Algorithm

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

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.

C Inducing a Procedure Graph from Natural Language

ProcFlow provides a procedure graph by construction. The established benchmarks provide a prompt, an environment, and a grader, but no procedure. For these benchmarks, we must recover the graph of Definition 2 from the task's natural language before any stage after §3.5 can run. This appendix describes how we recover the graph and how we validate it. Appendix D describes how constraints are emitted from a graph. Emission is deterministic, and the graph is the only interface between the two stages.

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}"
    }
  ]
}
```


The design separates the work of a language model from the work of code. The model reads natural language and fills in node-local fields: the action a step requests, the tool it likely uses, and what it consumes and produces. Code derives all relational structure (which step feeds which, and what must precede what) and all formulas from these fields. The model never asserts a dependency and never writes a formula. This design has two benefits. First, every model output is local and can be checked against the task text or a tool schema. An asserted edge or formula cannot be checked in this way. Second, once a graph is stored, the full downstream pipeline can be replayed without any model call. This lets us evaluate changes to the emission rules on a fixed corpus.

C.1 The Agent View and the Admissibility Invariant

Definition 3 (Agent view)**.** The *agent view* of a task is $\mathcal{V} = \langle U, \Sigma, \Pi, \tau \rangle$, where U are the user turns in order; Σ assigns to each $t \in \mathcal{T}$ its *leaf* argument paths, per-argument documentation, and any values the schema itself prescribes; Π is the operating policy the agent is given; and τ is the frozen episode clock.

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",
      "weight": 2.0,
      "passed": true,
      "detail": "\"tool_a\" was called (call #1)."
    },
    {
      "name": "tool_a_before_branches",
      "weight": 3.0,
      "passed": true,
      "detail": "\"tool_a\" (call #1) before \"tool_d\" (call #5)."
    },
    {
      "name": "branch_call_count",
      "weight": 3.0,
      "passed": true,
      "detail": "Predicate returned True (3 branch calls, expected >=
         3)."
    },
    {
      "name": "tool_b_for_even",
      "weight": 3.0,
      "passed": true,
      "detail": "\"tool_b\" called 3 time(s); expected >= 3."
    },
    {
      "name": "tool_b_before_tool_d",
      "weight": 4.0,
      "passed": true,
      "detail": "\"tool_b\" (call #2) before \"tool_d\" (call #5)."
    },
    {
      "name": "tool_d_called",
      "weight": 2.0,
      "passed": true,
      "detail": "\"tool_d\" was called (call #5)."
    }
  ]
}
```


Induction depends only on $\mathcal{V}$. The view contains no reference answer, gold trace, reward, or grader, and no stage of induction accepts one. This is necessary for downstream measurements to be meaningful. Constraints derived from a reference execution would encode the answer. The claim that enforcement improves task success would then reduce to the claim that giving an agent the answer improves task success.

C.2 Stages

Induction consists of three model-assisted stages and one deterministic stage, followed by emission and validation.

stage	model	output
step extraction	yes	nodes: steps with role, modality, guard, and what each needs and produces
binding	yes, code-validated	tool binding and leaf-path slot binding
entity naming	yes, code-validated	entities threaded through the procedure
edge inference	no	data, guard and order edges; cycles broken; topological order
emission	no	constraints (Appendix D)

Interview protocol. Each model-assisted stage is an iterative interview. We ask one question per call, since requesting extra fields silently degrades the others. We validate each answer inside the loop and return the reason for any rejection to the model. We parse tolerantly but validate strictly, so that a correct answer in an unexpected format is not lost. A round that accepts nothing is rephrased, and the interview stops after two such rounds.

Step extraction: orders become nodes. We make one call per user turn and ask only for imperative steps. Each node records the step; a verbatim span; whether the step comes from an instruction or from the policy; a role (OBSERVE, ACT, or ABSTAIN, plus DECIDE and REPORT, which carry no tool obligation); a modality; a guard and the subject it reads; any explicitly stated ordering; candidate tools; and what the step needs and produces, in natural language. The last field is the most important in the pipeline. Models name what a step consumes and yields reliably, but they name dependencies unreliably. Recording consumption and production turns edge inference into a matching problem, which code solves and checks against Σ. We merge nodes that share a tool and a role and whose slots do not conflict, since one action described twice is still one action. Appendix D derives from the number of act nodes.

Binding: tools and slots. We intersect the candidate tools with the tools in $\mathcal{V}$. A node with no remaining candidate is marked unbound, recorded with a reason, and counted as a coverage deficit. Slots are then filled by a per-node interview. The interview first classifies each pending slot as a stated literal, a time, a measurement, or a value produced by another step. It then asks a question specific to that type. Each slot records an argument path, a value, and a provenance. The provenance is either a span or a label naming Σ, τ, or a policy table.

Entity naming. Before drawing edges, we name the entities that flow through the procedure. For each entity, we record the step that produces it and the steps that consume it. This stage makes implicit state explicit. Consider the instruction *"Delete my last email from kofi."* It is a single imperative, but executing it requires an identifier the prompt never mentions, namely the one returned by the search. Naming this entity introduces a dependency between the two steps. Downstream, it also yields a provenance obligation: the deleted identifier must be one that the search actually returned. This obligation separates an identifier that was looked up from one that was guessed.

Edge inference. Edges are derived deterministically, and each edge records the evidence used to create it. A data edge $u \to v$ at slot p is created in four cases: (i) u produces an entity that is consumed at p; (ii) p matches a handle shape under Σ and u's tool can produce such a handle; (iii) the slot explicitly refers to u; or (iv) a phrase in v's needs matches a phrase in u's produces, and u's role returns a value. A guard edge $u \to w$ connects a conditional node w to the node whose produces matches the guard's subject. If a guard has no such anchor, it cannot be evaluated from a trace. It therefore contributes only the observation it depends on and never creates a standalone obligation. An order edge is created from explicit textual sequencing. Every data and guard edge also creates an order edge.

D SPECIFICATION COMPILATION

This appendix details the compiler of §3.5: the constraint format, the rules that map nodes and edges to formulas, the invariants these formulas satisfy, and the validation of a compiled specification. Compilation is a deterministic function of the procedure graph $\mathcal{G}$ and the tool set. It uses neither a language model nor a reference execution.

D.1 CONSTRAINT FORMAT

Each emitted constraint consists of a formula and the metadata needed to report it, enforce it, and trace it to its source:

field	content
formula	the AGENTLTL formula κ
rationale	the reason for the requirement
repair	the instruction shown to the agent when a call is intercepted
weight	w_i in Definition 1
provenance	the node or edge of $\mathcal{G}$ that produced it, and the evidence for that edge

Weights. Compilation assigns $w_i = 1$ to all constraints. The weights in Definition 1 allow a hand-written procedure to encode relative importance, but a compiled procedure has no basis for such judgments. (§3.4) ignores weights: the runtime effect of a violation is determined by the constraint's disposition (§D.3).

D.2 EMISSION RULES

Each constraint is produced by one of the rules below. Node rules yield step-local requirements; edge rules yield the relational requirements that distinguish a procedure from a checklist.

Node rules.

node	emits
OBSERVE or ACT, unconditional and unguarded	presence of the bound tool
bound node, per grounded slot	an argument binding, guarded by the tool being called
ABSTAIN without a guard edge	an unconditional prohibition of the tool
n ACT nodes on one tool	at most n calls to that tool
write tool	the call contains only schema-declared fields
REPORT	provenance of the final answer; no tool requirement

Edge rules.

edge	emits
$\mathsf{data}(u \to v, p)$	a guarded ordering of u before v, and provenance requiring that v's argument at p contain a value returned by u
$\mathsf{guard}(u \to w, c)$, anchored	u is required on every branch; w is required on the branch selected by c
$\mathsf{guard}(u \to w, c)$, c is emptiness, w abstains	w is prohibited if u returned no result
guard, unanchored	u only; w is recorded as guard-only
$\mathsf{order}(u \to v)$	a guarded ordering

Table 3: 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.

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	$(13n + 3) \bmod 50$

D.3 Enforcement Taxonomy

Each formula is assigned two labels based on its structure.

Act. The error type the constraint targets: binding, abstention, sourcing, ordering, bounding, presence, or coverage. Feedback to the agent is ordered by this label, with binding errors first because they produce incorrect records.

Disposition. The point at which a violation becomes decidable, which determines whether interception is sound.

disposition	decidable	may block
at_call	when the call is proposed	yes
on_evidence	once a producer's output is recorded	yes
at_end	only at episode termination	no

A presence requirement is a liveness property: no finite prefix violates it, since the call may still occur, so blocking on it would penalize the agent for not having acted yet. Argument constraints are guarded by the call, so their violations are decidable when the call is proposed. Dispositions are computed by the same procedure used by the offline gates, so no runtime path can block on a constraint that the gates classify as non-blocking.

E ProcFlow and Data Generation

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

E.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 3 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 $\{\texttt{tool_a}, \texttt{tool_b}, \texttt{tool_c}, \texttt{tool_d}\}$. Both settings share the same procedural patterns; they differ only in vocabulary, prompt phrasing, and parameter ranges (Section E.5).

Table 4: The set of production ProcFlow templates, their description, and the subset of tools they require.

#	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`

E.2 Production Benchmark Templates

The production benchmark comprises 12 templates, grouped into the five structural types summarized in Table 5. Table 4 describes each template; tool abbreviations refer to the names in Table 3.

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
```

E.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 6 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.

Table 5: The twelve production templates and the failure mode each one isolates.

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 6: Difficulty scaling per template.

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

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.

E.4 Instance Generation Procedure

Each benchmark run is uniquely identified by the triplet $(\text{template}, \text{seed}, \text{difficulty})$; the default grid contains 12 templates $\times$ 5 difficulties $\times$ 2 seeds $= 120$ instances. 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` and the ordered `tool_call_log` of $\{\text{name}, \text{args}, \text{result}\}$ triples; (iv) instantiates fresh tool objects per run; and (v) builds the constraint set from a *procedure graph*.

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 7.

Table 7: Fields stored with each generated instance of ProcFlow.

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.
`num_tool_calls_expected`	Total tool calls in the correct trajectory.
`difficulty`	Integer in $\{1, \ldots, 5\}$.

Table 8: Production vs. simplified template settings in ProcFlow.

Aspect	Production mode	Simplified mode
Tool vocabulary	8 tools (`a`/`b`/`c`/`d` + `a_list`, `iterations`, `expand`, `value`)	4-tool pool $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$
Trace length	D_1: $\sim$3–10 calls; D_5: up to 50 calls (filtered)	3–7 typical, hard-capped at 8
Parameter ranges	Wide: target $\leq$ 200, depth $\leq$ 6, list $\leq$ 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

E.5 Production vs. Simplified Mode

The two settings share procedural flows (loops, branches, fan-out, fan-in, conditional re-processing) but differ in vocabulary, prompt length, and parameter ranges. Table 8 summarises the differences.

The 3–8 call window of simplified mode provides a solid 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 FO-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).

F Benchmark Environment Adaptation

Appendix C describes how a task's natural-language description is converted into a procedure graph and compiled into a machine-checkable specification G_P. This appendix describes the modifications required to score such a specification against episodes from four existing benchmarks (BFCL, τ-bench, WorkBench, and MedAgentBench) and to enforce it during execution, together with the constraints we imposed on those modifications.

None of these benchmarks were designed for this form of evaluation. Each assumes a specific agent loop and output format, and each evaluates task correctness only.

F.1 Design Principles

We only adapted environments as follows. Task correctness is computed by each benchmark's original grader, unmodified and without access to the specification, since a grader derived from the enforced constraints would make any improvement circular. When a grader expects a transcript format our runner does not produce, we reconstruct the transcript rather than reimplement the grader. The agent receives exactly the inputs the benchmark specifies, and graph induction is restricted to the same inputs (Appendix C). AgentLTL interacts with the environment only between the agent's proposal of a tool call and its execution, and we never modify environment logic to make a constraint checkable. Finally, each episode record must contain everything needed to re-score the episode offline.

F.2 Tool-Call Interception

Runtime enforcement requires that agent actions be issued as discrete, inspectable tool calls. Benchmarks that do not satisfy this requirement must be adapted before enforcement can have any effect.

MedAgentBench is the clearest example. Its reference agent loop operates on plain text: the agent emits lines that the harness parses as either a `GET` request with query parameters or a `POST` request followed by a JSON body, and the harness executes the parsed request. Because no tool calls are issued, there is no point at which an action can be inspected before execution, and rejecting an action would require editing the model's generated text.

We therefore expose the benchmark's nine operations as tools, so that each action becomes a proposal that can be inspected and, if necessary, rejected. This change is incompatible with the original grader, which reads an upstream transcript object and matches specific literal strings that identify a write request and its acceptance. We reconstruct the transcript from the recorded trace in the original format, including these literal strings, and run the graders unchanged.

More generally, we convert the agent's interaction protocol and reconstruct the grader's expected input, but we never reimplement the grader.

G Training Data and Patterns

G.1 Simplified Training Templates

The training corpus is built from 15 simplified templates over the four-tool simplified-mode pool $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$. Trace lengths range from 3 to 8 calls with augmentation. Table 9 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.

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.

G.2 Training Data Construction Pipeline

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

Pool construction. We first enumerate every pattern variant, each composed of a template and a transform choice. Of the 15 templates, 11 have a single fixed transform and 4 admit two transform variants, 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.

Table 9: 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`.

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 2×2 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`

Trace augmentation. To discourage memorization of fixed call sequences, every core trace is expanded into up to four augmentation variants:

`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 8-call hard cap, traces with ≥ 7 core calls receive only the `none` and `prefix` variants. The resulting pool contains 1,988 traces (570 `none`, 570 `prefix`, 424 `suffix`, 424 `both`).

Constraint construction. Every trace in the pool receives its own constraint specification, built by the same graph procedure as the benchmark (Section E.4): the gold trace is aligned with the template's procedure description to form a procedure graph, and constraints are emitted from the graph by fixed rules, each assigned a category (`do`, `bind`, `bound`, `order`, `abstain`, `source`) derived from its formula. A training example carries 18.0 constraints on average.

Effect on constraints. The extra `tool_a` call is functionally independent of the main computation, and the graph reflects this: it shares no data edge with the core procedure, so no ordering or provenance constraint links it to the other calls. It does change the constraint profile. Each extra call adds a `bind` constraint fixing its seed to the value stated in the prompt, and it shifts the `tool_a` call-count floor and ceiling (`bound`). When the core procedure does not otherwise call `tool_a`, it also adds a `do` constraint and a new floor. When the core already calls `tool_a`, ordering constraints that refer to `tool_a` are restated over specific occurrences of the call rather than its first occurrence.

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

Table 10: Alias pool sizes and example mappings for tools in ProcFlow.

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`

	Compliance $C(\tau, G_P)$					Blocks / episode			Killed (%)
Model	Base	soft	b&w	pers.	rev.	soft	b&w	pers.	soft
Qwen3.6	.985	.998	**1.000**	.998	.989	0.21	0.27	0.15	2
Qwen3.8	.955	.993	.994	**.998**	.989	0.46	0.41	0.41	2
Gemma31B	.964	**1.000**	.974	**1.000**	.989	0.40	5.92	0.44	10
Gemma12B	.914	.960	.933	**.965**	.928	0.37	1.18	0.50	3
DeepSeek	.995	**1.000**	**1.000**	**1.000**	.992	0.05	0.07	0.04	0
Qwen4B	.839	**.886**	.884	.883	.832	0.92	3.56	0.53	9
Mean	.942	.973	.964	**.974**	.953	0.40	1.90	0.35	4.3

Table 11: ProcFlow: compliance, blocks per episode and share of episodes terminated by the harness, per model and setting (temperature 0.7, n=360 per cell). *Base* = Baseline (unconstrained); *soft* = Soft-block; *b&w* = Block-and-warn; *pers.* = Persistent-block; *rev.* = Reviewed, the constraint-free Reinforced Agent of Ta et al. (2026).

evenly (12 per template), for a final target of 20 examples per template. The resulting training set contains 85 `none`, 85 `prefix`, 65 `suffix` and 65 `both` examples.

G.3 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 10 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)` → `correlate_signals(channel_a, channel_b)` → `normalize_reading(reading)` → `accumulate_readings(readings)`. This split tests whether the model has learned procedure structure versus surface-level pattern matching on tool names. On this split we report task success only; the underlying computation, and hence the expected answer, is invariant under renaming.

H Extended Harness Results

This appendix expands Table 1. Section H.1 gives ProcFlow scores per model and per template, and Section H.2 the four established benchmarks. Per-model accuracy for all five environments is in Table 12.

H.1 ProcFlow

Compliance. Table 11 gives the absolute values behind ProcFlow row of Table 1. All three enforcement settings raise compliance for all six models (18 of 18 cells). Two models reach 1.000 under at least two settings. The reviewer raises compliance for four models and lowers it for Qwen4B (.839 → .832) and DeepSeek (.995 → .992). Block-and-warn issues far more blocks than the

Table 12: Task accuracy per enforcement setting, by model. *Base* = Baseline (unconstrained); *soft* = Soft-block; *b&w* = Block-and-warn; *pers.* = Persistent-block; *rev.* = Reviewed, the constraint-free Reinforced Agent of Ta et al. (2026).

Benchmark (n)	Base	soft	b&w	pers.	rev.
Qwen3.6 (Acc %)					
BFCL (800)	**52.5**	49.8	52.4	51.2	51.9
τ-bench (150)	**43.3**	33.3	36.0	41.3	**43.3**
WorkBench (568)	73.8	69.0	75.2	68.8	**76.6**
MedAgentBench (150)	22.7	58.7	58.7	**62.0**	26.7
ProcFlow (360)	91.9	90.8	91.9	91.9	**92.5**
Qwen3.8 (Acc %)					
BFCL (800)	40.8	40.2	41.8	40.9	**46.0**
τ-bench (150)	27.3	22.0	**28.0**	24.7	23.3
WorkBench (568)	81.3	75.6	80.4	76.3	**82.9**
MedAgentBench (150)	10.0	34.0	**34.7**	33.3	23.3
ProcFlow (360)	98.1	95.6	**98.9**	98.3	98.6
Gemma31B (Acc %)					
BFCL (800)	55.0	52.6	55.0	52.9	**57.4**
τ-bench (150)	45.3	34.0	45.3	44.0	**52.0**
WorkBench (568)	79.8	74.3	82.2	74.1	**82.9**
MedAgentBench (150)	26.0	**49.3**	**49.3**	**49.3**	24.7
ProcFlow (360)	52.2	52.2	53.3	57.2	**60.3**
Gemma12B (Acc %)					
BFCL (800)	44.2	42.5	43.4	42.5	**45.1**
τ-bench (150)	42.7	40.0	40.7	**43.3**	38.0
WorkBench (568)	66.2	60.2	65.3	59.7	**71.7**
MedAgentBench (150)	17.3	37.3	**38.0**	32.7	24.7
ProcFlow (360)	33.6	46.4	43.1	46.4	**50.6**
DeepSeek (Acc %)					
BFCL (800)	43.8	**47.1**	46.2	45.1	42.4
τ-bench (150)	68.0	53.3	64.0	**70.0**	67.3
WorkBench (568)	**88.6**	82.7	84.9	81.2	87.3
MedAgentBench (150)	26.7	57.3	**62.7**	61.3	26.0
ProcFlow (360)	99.2	**100.0**	99.7	**100.0**	98.1
Qwen4B (Acc %)					
BFCL (800)	23.5	23.0	**23.8**	22.6	20.1
τ-bench (150)	22.7	22.0	22.7	22.0	**26.0**
WorkBench (568)	25.2	31.0	**32.0**	29.2	23.8
MedAgentBench (150)	10.7	10.7	10.7	10.7	**18.0**
ProcFlow (360)	14.2	**20.0**	19.7	17.2	10.3

other two enforcement settings (1.90 vs. 0.40 and 0.35 per episode) for a smaller compliance gain. Overriding a block in that setting requires re-issuing the call, and Gemma31B does this systematically (5.92 blocks per episode against 0.40 under soft-block, for +.010 compliance against +.036). Soft-block terminates at most 10% of episodes on any model.

Constraint-specific blocking. Table 13 gives two measurements for each constraint family. Discrimination is measured on baseline episodes, which compile no constraints, so it is a property of the specification alone. Blocking is measured under the enforcement settings. Over all families, a constraint is 9.5× more likely to be violated on a wrong episode than on a correct one, and only 1.26% of constraints are violated on correct episodes. `order` never fires on a correct episode: sibling iterations of a fan-out receive no ordering constraint (Section E.4), so a valid reordering is never penalised. `source` is the exception. It is violated only on correct episodes, where it has the highest rate of any family, but it never blocks. In practice, blocking is almost entirely call-count ceilings: `bound` issues 95% of block-and-warn blocks and 78% of soft-block blocks. `bind`, `order` and `source` never block. `do` blocks three times in 10,787 episodes, consistent with the absence of gains on missing-only instances.

Table 13: Constraint families on ProcFlow, pooled over six models. */inst.*: constraints per instance. *Fires*: share of the family's constraints violated on wrong and on correct baseline episodes. *Blocks*: blocks issued, from the enforcer's per-constraint counters. Persistent-block does not record per-constraint counters and is omitted.

		Fires on baseline (%)			Blocks	
Family	/inst.	wrong	correct	ratio	soft	b&w
`bound`	5.75	22.44	2.80	8.0×	675	3,892
`do`	3.21	6.81	0.41	16.6×	2	3
`abstain`	2.75	9.41	0.10	89.8×	190	210
`order`	2.05	0.41	0.00	∞	0	0
`bind`	1.83	7.41	0.34	21.5×	0	0
`source`	0.46	0.00	3.69	0.0×	0	0
All	16.05	12.00	1.26	9.5×	867	4,105

H.2 Established benchmarks

Protocol. We use four established benchmarks: BFCL multi-turn (800 instances: the `base`, `miss_func`, `miss_param` and `long_context` categories, 200 each), τ-bench (150: 45 airline and 105 retail tasks), WorkBench (568) and the five write groups of MedAgentBench (150: 30 tasks each). The retrieval groups of MedAgentBench contain no action a constraint could act on. Each instance has one specification, generated once from what the agent sees and identical across models and settings. Each (model, instance, setting) is run once, with the same seed in every setting. This gives 1,668 episodes per model and setting and 50,040 in total. All runs use the same agent loop and the benchmark's own step budget: 8 steps per user turn on BFCL, 30 on τ-bench, 20 on WorkBench and 15 on MedAgentBench. On BFCL, WorkBench and MedAgentBench, a turn in which every proposed call was blocked draws on a separate allowance of the same size rather than on the step budget. τ-bench's loop does not separate the two.

Correctness is each benchmark's own grader: BFCL's state and response checker, τ-bench's reward, WorkBench's outcome match and MedAgentBench's reference solution. None of them sees the specification. Episodes ended by the harness (soft-block termination) are scored as failures, as on ProcFlow. Eight episodes died on an endpoint transport error, all Qwen3.8 on WorkBench (three instances; 3, 2 and 3 episodes in the three enforcement settings). They are treated as undecided and excluded from every pairing. They are the † cells in Table 12. Compliance is computed post hoc on the final trace against the same compiled specification in every setting, including *Base* and *Reviewed*.

Compliance. Table 14 gives the absolute values behind the four external rows of Table 1. Soft-block and block-and-warn raise compliance for every model on every benchmark (24 of 24 cells), and persistent-block in 21. Its three exceptions are all on τ-bench, where its mean is .880 against a baseline of .882. The largest gains are on MedAgentBench, where soft-block and persistent-block bring five of six models to .984 or above. The exception is Qwen4B (.944 → .967), which never emits a well-formed write request. Its JSON payloads are malformed, the tool-call parser drops them, and the graded action never reaches the harness. The reviewer changes mean compliance by +.000 to +.009 per benchmark. Blocking density differs by an order of magnitude across benchmarks, from 0.14–0.18 blocks per episode on BFCL to 1.6–2.4 on MedAgentBench. BFCL is low because its quoted-value constraints are not enforced (see Protocol). Unlike on ProcFlow, persistent-block rather than block-and-warn issues the most blocks on τ-bench, WorkBench and MedAgentBench. With no override available, the agent re-proposes a refused call and is refused again. Soft-block terminates 0.8% (BFCL) to 6.6% (MedAgentBench) of episodes on average, and at most 18% on any model (Gemma12B on MedAgentBench).

Constraint blocking statistics. Table 17 is the counterpart of Table 13. The specifications discriminate far less on the established benchmarks than on ProcFlow. Pooled, a constraint is 1.9× more likely to be violated on a wrong baseline than on a correct one (ProcFlow: 9.5×), and 5.9% of constraints are violated on correct episodes (ProcFlow: 1.26%). MedAgentBench is the exception at 9.9×. Its `bind` constraints reach 8.0× because the values they assert are prescribed by the tool

Table 14: Established benchmarks: compliance, interventions per episode and share of episodes terminated by the harness, per model and setting. There is one run per instance, so n=800, 150, 568 and 150 per cell. Blocks are counted from the enforcer's record, so an episode that soft-block terminates still contributes its blocks. *Rewrites* are provisional calls the reviewer changed before execution, which is the reviewer's form of intervention. *Base* = Baseline (unconstrained); *soft* = Soft-block; *b&w* = Block-and-warn; *pers.* = Persistent-block; *rev.* = Reviewed, the constraint-free Reinforced Agent of Ta et al. (2026).

	Compliance $C(\tau, G_P)$					Blocks / episode			Rewrites / ep.	Killed (%)
Model	Base	soft	b&w	pers.	rev.	soft	b&w	pers.	rev.	soft
BFCL										
Qwen3.6	.915	**.923**	**.923**	**.923**	.914	0.13	0.18	0.13	0.90	0.9
Qwen3.8	.880	**.904**	.903	.903	.892	0.16	0.28	0.16	1.32	0.6
Gemma31B	.921	.927	.926	**.928**	.923	0.16	0.21	0.15	0.80	1.4
Gemma12B	.914	.917	.918	**.919**	.908	0.14	0.17	0.14	0.87	1.0
DeepSeek	.905	**.913**	**.913**	.910	.904	0.14	0.14	0.13	0.55	0.4
Qwen4B	.885	**.894**	.893	.890	.875	0.14	0.13	0.14	1.83	0.4
Mean	.903	**.913**	**.913**	.912	.903	0.14	0.18	0.14	1.04	0.8
τ-bench										
Qwen3.6	.879	**.903**	.892	.879	.888	0.53	0.61	0.60	0.02	4.0
Qwen3.8	.844	**.855**	.847	.833	.846	0.21	0.33	0.35	0.11	0.7
Gemma31B	.890	**.906**	.894	.891	.894	0.71	0.77	0.85	0.11	12.0
Gemma12B	.889	**.923**	.896	.888	.895	0.48	0.77	0.93	0.19	1.3
DeepSeek	.898	**.929**	.919	.904	.901	0.55	0.78	0.87	0.07	0.7
Qwen4B	.892	**.905**	.901	.882	.871	0.76	0.71	0.89	0.95	6.7
Mean	.882	**.904**	.892	.880	.882	0.54	0.66	0.75	0.24	4.2
WorkBench										
Qwen3.6	.886	**.928**	.909	**.928**	.891	0.54	0.60	1.24	0.23	8.8
Qwen3.8	.933	**.974**	.959	**.974**	.935	0.44	1.02	1.20	0.22	3.7
Gemma31B	.941	.988	.968	**.989**	.947	0.64	0.89	1.09	0.10	9.5
Gemma12B	.898	.937	.925	**.938**	.901	0.47	0.71	0.83	0.39	5.5
DeepSeek	.945	**.988**	.974	**.988**	.946	0.43	0.57	0.53	0.07	1.6
Qwen4B	.846	**.936**	.918	.932	.825	0.85	1.00	1.27	1.50	7.4
Mean	.908	**.959**	.942	.958	.908	0.56	0.80	1.03	0.42	6.1
MedAgentBench										
Qwen3.6	.860	.996	.987	**1.000**	.866	1.61	1.55	1.50	0.29	5.3
Qwen3.8	.738	**1.000**	.989	**1.000**	.754	2.07	3.41	3.50	1.07	5.3
Gemma31B	.862	**.991**	.983	.984	.865	1.80	1.60	2.41	0.48	10.7
Gemma12B	.805	**.996**	.969	.991	.824	2.31	3.07	5.21	0.45	18.0
DeepSeek	.861	**1.000**	.999	**1.000**	.870	1.34	1.46	1.35	0.16	0.0
Qwen4B	.944	**.967**	.954	.966	.942	0.45	0.44	0.46	0.93	0.0
Mean	.845	**.992**	.980	.990	.853	1.60	1.92	2.40	0.57	6.6

schema, and a correct agent sends them verbatim. On the other benchmarks `bind` asserts values quoted from the task text and sits at 0.7–3.8×. `bind` is also where blocking concentrates, with 48–56% of the blocks in each enforcing setting. On ProcFlow, by contrast, 95% of block-and-warn's blocks were call-count ceilings. Presence constraints never block: no block in any setting comes from a requirement to make an additional call. `abstain` is the only inverted family (0.3×). These are prohibitions whose negation was meant to restrict an argument or the content of a call, not the call itself.

Blocks on solved episodes. For each block issued on an instance whose baseline was correct, we check whether the blocking constraint is also violated by that correct baseline's complete trace. This separates a specification that is stricter than the benchmark's grader from a block that merely came at the wrong moment. Of the 3,146 such blocks with a baseline trace to check, 84.3% come from a constraint that the correct solution itself violates. By benchmark this is 81.1% on BFCL, 88.6% on τ-bench, 93.9% on WorkBench and 49.9% on MedAgentBench. A further 2,512 WorkBench blocks fall on baselines stored without a trace and are not counted. Outside MedAgentBench, most of the

Table 15: Intervention-to-failure ratio and its consequences, pooled over six models. *ratio*: interventions per episode on instances whose baseline was wrong, divided by the same on instances whose baseline was correct. An intervention is a block for the enforcement setting and a rewritten call for the reviewer. *fix*: share (%) of baseline-wrong instances with at least one intervention that the setting solves. *break*: share (%) of baseline-correct instances with at least one intervention that the setting fails.

	soft			b&w			pers.			rev.		
Benchmark	ratio	fix	break	ratio	fix	break	ratio	fix	break	ratio	fix	break
BFCL	1.63×	5.2	48.0	1.74×	5.9	23.9	1.54×	4.1	48.0	2.09×	18.3	38.1
τ-bench	1.04×	7.8	66.7	0.94×	20.6	39.6	1.07×	18.5	18.2	2.17×	18.6	50.0
WorkBench	2.22×	21.3	60.6	3.03×	25.1	18.6	2.35×	19.9	62.3	**7.11**×	24.8	29.8
MedAgentBench	3.01×	**38.2**	15.2	2.40×	**39.4**	20.3	**3.45**×	**37.6**	16.9	1.11×	18.0	22.8

Table 16: Accuracy change (points) against Baseline by the baseline failure class of each (model, instance) pair, pooled over six models. *soft* = Soft-block; *b&w* = Block-and-warn; *pers.* = Persistent-block; *rev.* = Reviewed, the constraint-free Reinforced Agent of Ta et al. (2026). n is for all four benchmarks; excluding MedAgentBench leaves 718 refusable-only, 1,257 missing-only and 1,150 silent pairs. *Unclassified*: baseline stored without a trace. **Bold**: best per row within each half.

		All four				Excl. MedAgentBench			
Failure class	n	soft	b&w	pers.	rev.	soft	b&w	pers.	rev.
Refusable only	1244	+22.7	**+23.0**	+22.6	+11.2	+9.3	+8.9	+9.3	**+13.2**
Mixed	746	+12.3	+11.8	+10.9	**+12.9**	+12.3	+11.8	+10.9	**+12.9**
Missing only	1317	+7.4	+7.7	+7.2	**+15.3**	+7.7	+8.1	+7.6	**+16.1**
Silent	1294	+7.5	+10.4	+9.1	**+18.8**	+7.9	+10.6	+9.3	**+19.0**
Unclassified	427	+10.6	+15.5	+10.3	**+18.6**	+10.6	+15.5	+10.3	**+18.6**
Solved	4978	−12.9	**−8.7**	−12.1	−12.7	−12.9	**−8.5**	−12.1	−12.6

harm on solved episodes therefore comes from the induced specification being stricter than the grader, not from the enforcement mechanism. Pooled over the four benchmarks, solved (instance, setting) pairs with at least one block lose 41.4 points, against 6.3 for pairs without one.

I Extended Finetuning Results

I.1 Established benchmark per-template accuracy

Table 18 breaks the Full row of Table 2 down by template (30 episodes per cell). Templates are sorted by their mean over the seven finetuned models.

The loop and nesting templates remain the hardest after finetuning. The $c{+}a$, c and g models share the same five lowest templates (`nested_loops+branching`, `branch_selection`, `nested_loops`, `dynamic_expansion`, `loop+expansion`); averaged over these three models they reach 54–80% on these templates against 97–100% on the other seven. `branch_selection` is the one template that no GRPO variant takes above 57%. The five-lowest set differs for $c{+}a{+}g$, which instead scores lowest on `fan_out+loop` (16.7%) and `tool_condition_branching` (50.0%), and for the SFT models, which are at or near 0% on all three loop and nesting templates (`nested_loops`, `nested_loops+branching`, `loop+expansion`) while keeping 100% on `forall_processing`. The SFT degradation is therefore concentrated in the loop family rather than spread uniformly.

I.2 Gains by base failure shape

To relate finetuning gains to the kind of failure the base model makes, we score every base-model episode against the constraint specification and classify each violated constraint as *refusable* (a call-count ceiling, a wrong argument, or a forbidden call; these can be refused at the moment the call is made) or *missing* (a required call that never happened, or a call-count floor not reached). An

Table 17: Constraint families on the established benchmarks. */inst.*: constraints per instance. *Fires*: share of constraints violated on wrong and correct baseline episodes (8,645 baselines with a trace). *Ratio*: wrong over correct; ∞: never violated on a correct baseline; –: never violated or absent. τ = τ-bench, WB = WorkBench, MAB = MedAgentBench. *Blocks*: blocks issued over six models by Soft-block (*soft*), Block-and-warn (*b&w*) and Persistent-block (*pers.*). `do` (value flow): a consumed value must come from a producer's output, which makes it decidable from the trace. **Bold**: highest finite ratio per column.

		Fires on baseline (%), pooled			Ratio, per benchmark				Blocks		
Family	/inst.	wrong	correct	ratio	BFCL	τ	WB	MAB	soft	b&w	pers.
`bind`	5.31	10.66	5.40	2.0×	1.1	0.7	3.8	**8.0**	2,203	3,344	3,402
`do` (presence)	3.02	16.43	7.28	**2.3**×	**3.1**	**1.8**	1.7	∞	0	0	0
`do` (value flow)	0.79	5.05	2.99	1.7×	1.6	0.8	**7.7**	–	647	834	747
`order`	1.76	8.76	5.97	1.5×	2.0	1.1	0.8	–	878	852	1,286
`bound`	0.59	9.82	8.22	1.2×	0.9	0.8	1.3	∞	737	783	1,494
`source`	0.11	4.99	0.00	∞	–	∞	–	∞	38	96	70
`abstain`	0.04	2.37	7.89	0.3×	0.2	0.0	–	–	17	20	18
All	11.62	11.36	5.93	1.9×	1.8	1.2	1.6	9.9	4,520	5,929	7,017

Table 18: Accuracy (%) per ProcFlow template, 30 episodes per cell (Base: 31 on `nested_loops` and 34 on `loop+expansion`, from duplicate runs). GRPO columns give the reward terms: c = compliance score, a = accuracy, g = gold distance. SFT_k: supervised fine-tuning for k epochs. †Held out from training. **Bold**: best per row, ties included.

		GRPO				SFT		
Template	Base	c+a+g	c+a	c	g	SFT_1	SFT_2	SFT_3
`nested_loops+branching`	0.0	0.0	**83.3**	63.3	**83.3**	3.3	0.0	0.0
`branch_selection`†	3.3	53.3	53.3	53.3	**56.7**	23.3	3.3	6.7
`nested_loops`	0.0	20.0	76.7	**83.3**	80.0	0.0	0.0	0.0
`dynamic_expansion`	13.3	46.7	**66.7**	53.3	56.7	26.7	16.7	23.3
`loop+expansion`	0.0	70.0	73.3	**76.7**	73.3	0.0	0.0	0.0
`fan_out+loop`	0.0	16.7	**100.0**	96.7	93.3	33.3	6.7	6.7
`tool_condition_branching`	0.0	50.0	**100.0**	**100.0**	**100.0**	10.0	3.3	6.7
`loop_termination`†	13.3	70.0	**100.0**	**100.0**	**100.0**	13.3	16.7	10.0
`forall_exists_combined`	13.3	86.7	**100.0**	**100.0**	**100.0**	60.0	36.7	36.7
`exists_selection`	0.0	83.3	**100.0**	**100.0**	**100.0**	93.3	40.0	50.0
`fan_out_fan_in`	20.0	96.7	90.0	**100.0**	**100.0**	36.7	63.3	80.0
`forall_processing`	**100.0**	93.3	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

episode is *refusable-only*, *missing-only* or *mixed* according to which kinds it violates, and *silent* if it is wrong without violating either. Ordering and provenance constraints are not used. An instance's base shape is the majority shape among the base model's wrong rollouts (ties count as mixed); instances the base model always solves are *solved*. The gain is the finetuned model's accuracy minus the base model's accuracy on the same instance, averaged over instances of that shape.

Table 19: Accuracy gain over the base model (points) by the base model's failure shape; n = instances. GRPO reward terms: c = compliance score, a = accuracy, g = gold distance. SFT_k: supervised fine-tuning for k epochs. Rows with $n < 10$ are not interpreted. **Bold**: best per row, ties included.

Base shape	n	c+a+g	c+a	c	g	SFT_1	SFT_2	SFT_3
refusable-only	46	+57	+88	**+89**	+88	+26	+15	+14
mixed	56	+42	+77	+74	**+79**	+19	+10	+14
missing-only	5	+40	+40	+33	+33	+0	+7	+20
silent	2	+67	+100	+100	+100	+100	+17	+67
solved	11	−6	**+0**	**+0**	**+0**	−9	−9	−9

Nearly all instances the base model fails are refusable-only or mixed (102 of 109). On both, the three fully trained GRPO ablations recover most of the gap ($+74$ to $+89$ points), while SFT recovers at most 26 points. On the 11 instances the base model always solves, the three GRPO ablations lose nothing, while $c{+}a{+}g$ and the SFT models lose 6–9 points.

I.3 Constraint categories after finetuning

Table 20 reports, per model, the pass rate of each constraint category of the specification on the model's own full-benchmark episodes. Episodes are scored offline.

Table 20: Pass rate per constraint category on ProcFlow; n = episodes (aborted Base runs excluded). GRPO reward terms: c = compliance score, a = accuracy, g = gold distance. SFT_k: supervised fine-tuning for k epochs. **Bold**: best per column, ties included.

Model	n	`do`	`bind`	`bound`	`order`	`abstain`	`source`
Base	354	.881	.974	.701	.796	.793	.963
$c{+}a{+}g$	360	.979	.796	.770	**1.00**	**.999**	.994
$c{+}a$	360	**.999**	.979	**.943**	.997	.990	.982
c	360	**.999**	**.985**	**.943**	**1.00**	.986	**1.00**
g	360	**.999**	.977	.938	.996	.987	.976
SFT_1	360	.997	.974	.736	.997	.838	.952
SFT_2	360	.994	.982	.774	.992	.763	.970
SFT_3	360	.993	.980	.777	.992	.790	.945

Three patterns stand out. First, call-count bounds are the most frequently violated category for every model; only the fully trained GRPO ablations reach about 0.94. Second, $c{+}a{+}g$ is the only finetune whose argument-binding pass rate falls below the base model (0.796 vs. 0.974): its errors are wrong literal arguments rather than wrong procedures. Third, the SFT models violate `abstain` constraints (0.76–0.84), i.e. they call the tool that the branch condition rules out, which the GRPO models almost never do (0.986–0.999). Ordering is essentially satisfied by every finetuned model.

I.4 Established benchmarks

Each model is run four times on every established benchmark, in the baseline setting (no constraints shown or enforced), with the same sampling seed; replicate differences therefore reflect pipeline nondeterminism only.

Replicate spread. Table 21 gives the mean and standard deviation over the four runs. On BFCL, WorkBench and MedAgentBench the standard deviation is at most 1.3 points, well below most finetuning effects. τ-bench is noisy: the base model alone ranges from 19.4% to 27.9% across its four runs, so differences of a few points on τ-bench are not interpretable.

Per-domain breakdown. Table 22 splits τ-bench, WorkBench and MedAgentBench by domain or task group.

The WorkBench gain of $c{+}a{+}g$ is broad: it improves on all six domains, by 7 to 14 points. Its MedAgentBench gain comes mostly from task group 9 ($3.3\% \rightarrow 25.0\%$). The SFT losses on MedAgentBench are concentrated in two task groups: group 4, where accuracy drops from 63.3% to 5% or below, and group 2 for SFT_2 and SFT_3. Task groups 3, 8 and 10 are unsolved by every model.

Table 21: Accuracy (%) on the established benchmarks, mean$_{\pm \text{sd}}$ over four replicate runs; n = instances. GRPO reward terms: c = compliance score, a = accuracy, g = gold distance. SFT$_k$: supervised fine-tuning for k epochs. **Bold**: best mean per row.

		GRPO				SFT		
Benchmark (n)	Base	c+a+g	c+a	c	g	SFT$_1$	SFT$_2$	SFT$_3$
BFCL (800)	$24.7_{\pm 0.6}$	$27.6_{\pm 0.2}$	$27.8_{\pm 0.7}$	$28.2_{\pm 0.5}$	$27.5_{\pm 0.6}$	$25.6_{\pm 0.8}$	$27.6_{\pm 0.3}$	$\mathbf{28.6}_{\pm 0.7}$
τ-bench (165)	$24.1_{\pm 3.5}$	$25.0_{\pm 2.4}$	$\mathbf{28.0}_{\pm 0.8}$	$27.3_{\pm 4.1}$	$25.0_{\pm 0.6}$	$25.5_{\pm 2.6}$	$27.4_{\pm 1.5}$	$26.2_{\pm 1.2}$
WorkBench (568)	$26.8_{\pm 0.5}$	$\mathbf{36.0}_{\pm 0.4}$	$30.0_{\pm 0.3}$	$30.6_{\pm 0.7}$	$30.4_{\pm 1.0}$	$26.1_{\pm 0.5}$	$23.9_{\pm 0.5}$	$23.4_{\pm 0.4}$
MedAgentBench (300)	$34.9_{\pm 0.8}$	$37.1_{\pm 1.3}$	$36.2_{\pm 1.1}$	$\mathbf{37.6}_{\pm 0.6}$	$36.6_{\pm 1.3}$	$27.2_{\pm 0.6}$	$22.5_{\pm 0.3}$	$22.1_{\pm 0.4}$

Table 22: Accuracy (%) by domain or task group, mean over four runs; n = instances. τ = τ-bench, WB = WorkBench, MAB = MedAgentBench. GRPO reward terms: c = compliance score, a = accuracy, g = gold distance. SFT$_k$: supervised fine-tuning for k epochs. **Bold**: best per row, ties included (none where all are 0).

		GRPO				SFT		
Domain (n)	Base	c+a+g	c+a	c	g	SFT$_1$	SFT$_2$	SFT$_3$
τ airline (50)	25.5	25.5	**31.0**	26.0	25.0	24.5	23.5	22.0
τ retail (115)	23.5	24.8	26.7	27.8	25.0	25.9	**29.1**	28.0
WB analytics (80)	32.2	**40.9**	33.8	34.4	36.2	33.1	30.3	30.9
WB calendar (99)	38.4	**47.0**	43.4	42.2	39.9	30.8	29.0	31.3
WB CRM (72)	17.7	**31.9**	25.3	28.5	28.1	13.9	13.2	11.8
WB email (89)	35.1	**42.4**	34.0	36.0	33.4	36.0	34.0	30.3
WB project mgmt. (59)	28.4	35.2	33.9	34.7	32.6	**39.4**	26.3	25.8
WB multi-domain (169)	16.4	**25.9**	18.8	18.8	20.7	15.4	16.4	15.5
MAB task 1 (30)	**100.0**	99.2	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
MAB task 2 (30)	75.8	68.3	75.8	**76.7**	71.7	70.0	30.0	26.7
MAB task 3 (30)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MAB task 4 (30)	63.3	**67.5**	63.3	63.3	63.3	5.0	0.8	0.8
MAB task 5 (30)	60.0	56.7	66.7	**79.2**	75.8	53.3	53.3	53.3
MAB task 6 (30)	6.7	**12.5**	7.5	6.7	6.7	0.0	0.0	0.0
MAB task 7 (30)	40.0	41.7	40.0	**42.5**	41.7	40.8	40.0	39.2
MAB task 8 (30)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MAB task 9 (30)	3.3	**25.0**	9.2	7.5	6.7	2.5	0.8	0.8
MAB task 10 (30)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0