MENTOR: PROCEDURAL SHIELDING OF TOOL-USING LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-using LLM agents are increasingly used to automate workflows governed by procedural manuals, such as company rules for booking travel or procuring hardware. These manuals are written in natural language for humans and describe general requirements over such complex workflows. Ensuring compliance of LLM agents with these manuals is difficult, and state-of-the-art techniques use either LLM judges or manually written rules to forbid problematic tool-calls locally at runtime. This is insufficient because (i) only blocking tool calls keeps the agent clueless about how to make correct progress towards fulfilling its task, and (ii) can lead to overly conservative behavior of the agent, as tool calls might only be problematic in certain contexts. To overcome these challenges, we introduce MENTOR, which (i) additionally provides guidance for the agent to make procedural-compliant progress on the task, and (ii) ensures that this compliance is context-aware. We achieve these goals by building a novel history-dependent runtime shield that is obtained from the procedural manual via a multi-step autoformalization pipeline. At its core, MENTOR utilizes a game-based technique to ensure robustness against runtime uncertainty without over-resticting agents' behavior. This yields a formally assisted framework for procedural shielding of LLM agents that improves compliance well beyond existing methods. In particular, MENTOR lifts the compliance of shielded light-weight open-source LLMs above that of non-shielded proprietary models. Since a shield is synthesized once per manual and then serves every task, a MENTOR-shielded agent becomes a viable choice for repeatedly executed workflows, potentially reducing deployment costs when the offline construction cost is amortized over repeated tasks.

1 INTRODUCTION

The abilities of tool-using large language model (LLM) agents allow them to be used to automate workflows such as hardware procurement (Zdravković et al., 2022; Ozman, 2025), patient diagnosis (Tu et al., 2025; Ghezloo et al., 2025) or travel bookings (Dwivedi et al., 2025). Such workflows are governed by *procedural manuals*, written in natural language for human operators, that specify *how* such tasks must be executed. For example, Fig. 1 (top right) shows a simple manual for booking a business trip.

While such workflows may seem natural for humans, LLM agents have been observed to struggle with them, both in existing work about evaluating compliance post-hoc (Li et al., 2025; Nandi et al., 2026; Anand et al., 2026a), and in our own experiments. For instance in the domain of Fig. 1, checking agents' tool call traces reveals they often attempt to book business class after inventing a fake "urgent reason", or never book anything at all.

To aid LLM agents in such tasks, we introduce MENTOR, a procedural shield that checks proposed tool calls against formalized requirements and guides agents towards fulfilling outstanding obligations. Shielding is a well-established technique for policies obtained by reinforcement learning (RL), where a shield monitors the actions a policy proposes at runtime and blocks those that violate a formal specification (Könighofer et al., 2022; Hsu et al., 2024). We develop this approach for tool-using LLM agents around four interconnected challenges.

⟨*C1*⟩ *Context-dependence & guidance.* A procedural manual, as e.g. the one in Fig. 1, induces a partial temporal order over tool calls, so whether a tool call complies is *context-dependent* (e.g., booking a flight is allowed in step 5 but not in step 2). This context-dependence introduces *obligations*

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.

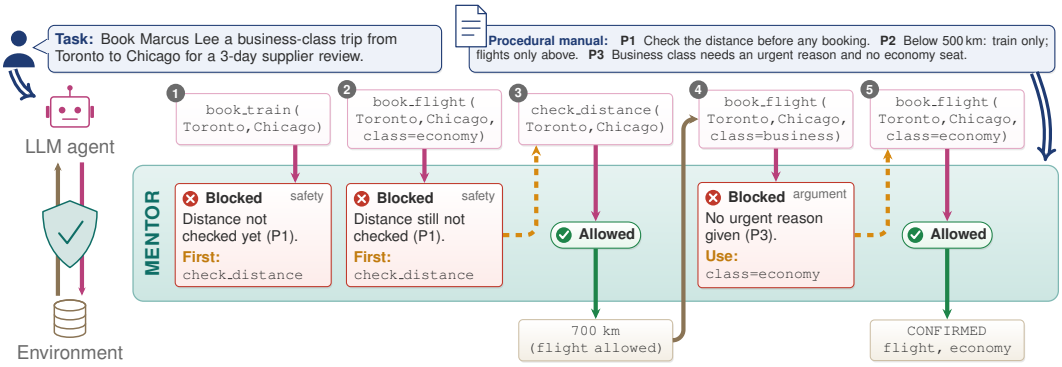

Figure 1: **Running example.** Our procedural shield MENTOR is generated from a procedural manual (top right) and applied to a travel-booking task (top left). MENTOR shields the interaction between an LLM agent (pink) and the environment (brown). MENTOR blocks tool calls (red), allows tool calls (green), and provides guidance towards task completion (orange).

an agent needs to fulfill before another necessary call would be allowed (e.g., checking the distance before any booking) and it must eventually make some specific tool calls to complete its task (e.g., the booking itself). Blocking a call without identifying a compliant next step can leave the agent repeatedly attempting blocked calls and failing to complete its task, as illustrated by our experiments (Fig. 5 in App. A.2). Shields that also incorporate context-dependent progress guidance were only recently introduced for RL as *strategy template based runtime shields (*STARS*)* (Anand et al., 2026b). MENTOR adapts this idea to LLM agents, by first auto-formalizing the manual into Linear Temporal Logic (LTL) and then synthesizing a strategy template from it.

⟨*C2*⟩ *Argument-based constraints.* Context-dependence does not only affect the order of tool calls, but also their arguments. For example, booking an economy class flight is allowed without an urgent reason in step 5, whereas booking business class is not in step 4. The values these arguments can take often range over unbounded domains, such as strings, and might only be revealed at runtime, making it challenging to express constraints over them. The importance of checking argument values is illustrated by our analysis of SOP-Bench (Nandi et al., 2026): its evaluator accepts some procedurally non-compliant traces because it does not check the relevant tool arguments (App. D.3). MENTOR addresses this problem with argument constraints using Satisfiability Modulo Theories (SMT) that are checked and, if violated, repaired at runtime.

⟨*C3*⟩ *Natural language.* Any formally assisted framework for procedural shielding of LLM agents needs to translate natural language texts within the manual, the task description, and tool outputs into formal artifacts. In our work, we automatically translate the manual into a formal specification offline. Then, at runtime, we observe that both cheap proprietary LLMs and fine-tuned small open-weights LLMs can effectively ground the task description and tool outputs.

⟨*C4*⟩ *Delivering the shield.* A classical shield for an RL policy intervenes directly on the action the policy proposes. In the context of LLM agents, however, actions are tool calls which are written token by token. Depending on the availability of the agent's next-token distribution, MENTOR can deliver the blocking and advice of tool calls to the agent in two ways: (i) via a prompt shield that injects textual feedback in the tool call outputs, and (ii) via a token shield, which masks the tokens of unsafe calls and boosts those of progressing calls during generation.

To show its impact on procedural compliance of shielded agents, we evaluate MENTOR on 3 benchmarks covering 8 domains using both proprietary and open-weights agents. The experiments show that MENTOR clearly outperforms unshielded agents and other guarding methods, and lifts the compliance of small open-weight models above those of unshielded proprietary ones.

Related Work. Shielding LLM agents has received increasing attention very recently, but none of the existing approaches addresses the outlined challenges appropriately. We compare them here and in Table 2 in App. A in more detail.

One line of work on LLM shielding techniques delegate compliance decisions to a guardrail LLM (Xiang et al., 2024; Hoover et al., 2026; Kang et al., 2026) which can interpret natural language (⟨C3⟩) within the manual, tasks, and tool outputs. However, the resulting judgements can be unreliable due to the stochastic nature of LLMs. Alternatively, deterministic guards hard-code procedural constraints,

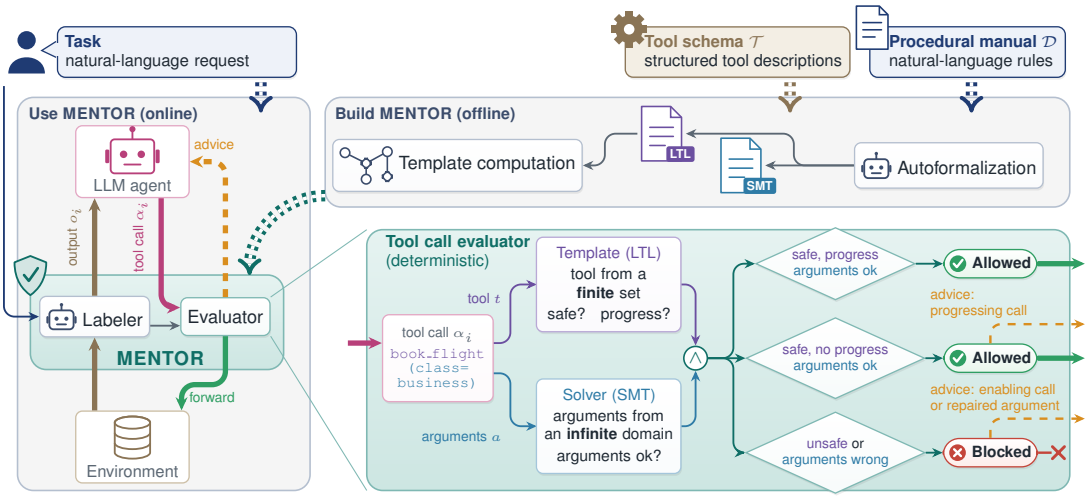

Figure 2: Overview of MENTOR. Manual and tool schema are formalized into an LTL-based template and an SMT-based solver (offline, top right), used to shield every tool call (online, left) by deterministically evaluating (bottom right) if tool calls are safe and progressing (violet, LTL), and use correct arguments (blue, SMT); unsafe and non-progressing calls receive advice (orange).

which are either written by hand in a dedicated language (Wang et al., 2026; Winston et al., 2026; Palumbo et al., 2026; Reddy et al., 2026; Kamath et al., 2025) or extracted by an LLM (Shi et al., 2025; Mondl et al., 2026). Such deterministic guards, however, cannot interpret natural language tool outputs during their execution. As a result, they cannot provide rich enough context-dependent guidance to the agent (⟨C1⟩). In addition, manually translating the procedural manual into formal constraints can make shield synthesis labor intensive and hard to extend to new domains, and this task is known to be difficult even for experts (Czepa et al., 2019; Greenman et al., 2023). MENTOR combines the strengths of both approaches by using a refinement-based auto-formalization technique to build a deterministic tool-call evaluator which is paired with a labeler to ground tool call outputs.

LTL was recently proposed as the right formalism to capture procedural dependencies (Anand et al., 2026a; Aminof et al., 2026), and translating natural language into LTL is well studied (Chen et al., 2023; Cosler et al., 2023; Fuggitti et al., 2023; Liu et al., 2023; Ma et al., 2025; Wang et al., 2025). However, none of the above work used LTL in the context of shielding LLM agents. In addition, LTL constraints are propositional and cannot express numeric ranges or unbounded argument domains (⟨C2⟩). Auto-generated SMT-based constraints have so far only served post-hoc verification (An et al., 2025; Anand et al., 2026a), not runtime shielding.

2 Overview

MENTOR (*Monitoring and ENforcement of Temporal Objectives at Runtime*) sits between a tool-using LLM agent and its environment (Fig. 2). We outline how its components address ⟨C1⟩–⟨C4⟩.

Setting. A tool schema $\mathcal{T}$ defines the set of tools available to the agent. Let $\mathcal{A}$ be the set of valid tool-calls $\alpha = (t, a)$, with tool $t \in \mathcal{T}$ and argument assignment a, including the terminating call `done` that ends the interaction. Let $\mathcal{H}$ denote the set of finite interaction traces $h = [o_0, \alpha_1, o_1, \ldots, \alpha_l, o_l]$, where o_0 is the natural-language task and o_i the environment's response to α_i. The agent $\pi : \mathcal{H} \to \Delta(\mathcal{A})$ repeatedly samples $\alpha_i \sim \pi(\cdot \mid h)$ and observes o_i, until it generates `done`. Each call is generated internally by an autoregressive LLM over a token vocabulary V. A natural-language procedural manual $\mathcal{D}$ specifies which traces are acceptable.

Deterministic tool call evaluator (Fig. 2, bottom). Our goal is that the trace generated by the agent completes the task o_0 while complying with $\mathcal{D}$. To this end, for each proposed call α_i and history h, the evaluator asks two questions: is α_i *safe*, i.e., whether a compliant complete trace under $\mathcal{D}$ still exists after α_i, and does α_i make *progress* towards completion (⟨C1⟩)? Also, are its *arguments* a_i permitted (⟨C2⟩)? Safe calls with permitted arguments are executed; non-progressing calls additionally receive advice identifying a progressing call. Other calls are blocked, returning

advice identifying a prerequisite call or repaired arguments instead. The call `done` is blocked while the task remains incomplete (Fig. 2, bottom).

Building MENTOR offline (Fig. 2, top). MENTOR is constructed once per manual $\mathcal{D}$ and tool schema $\mathcal{T}$ and then reused for every task. First, a multi-agent *autoformalization* loop (Sec. 3.2) translates $\mathcal{D}$ into a formal specification with two parts (Sec. 3.1): (i) *temporal requirements* in linear temporal logic (LTL) formulas to capture tool call order, restrictions that tool outputs impose, and obligations before `done`, and (ii) *argument constraints* in satisfiability modulo theories (SMT) encoding, which fix the permitted argument values over possibly unbounded argument domains ($\langle$C2$\rangle$). The LTL requirements are compiled into a game between agent and environment on a finite graph, and a *strategy template* (Sec. 3.2) marks, at every vertex of this graph, which calls are unsafe and which are progressive ($\langle$C1$\rangle$). The template and SMT solver form the deterministic evaluator.

Applying MENTOR online (Fig. 2, left). Since the task o_0 and some tool outputs are in natural language, at runtime, a *labeler* LLM (Sec. 3.3) grounds them into formal, *labeled events* consistent with the specifications ($\langle$C3$\rangle$). For each proposed call, the evaluator advances through the game graph following h, consults the template and the solver, and produces a verdict and advice. These reach the agent in one of two ways ($\langle$C4$\rangle$): (i) via a *prompt shield* that delivers them as text in the tool output, or (ii) via a *token shield* that additionally masks the tokens of unsafe calls and boosts those of progressing calls during generation.

3 Methodology

This section formalizes the components in Fig. 2 and explains how each addresses challenges $\langle$C1$\rangle$–$\langle$C4$\rangle$. We introduce our specification language in Sec. 3.1, then describe the offline construction of MENTOR in Sec. 3.2 and its online use for procedural shielding in Sec. 3.3.

3.1 Specification Language

Temporal Requirements via LTL. LTL extends propositional logic with temporal operators that express requirements on the ordering of events over time. We use standard LTL syntax with finte trace semantics (Giacomo & Vardi, 2013) (refer to App. C for details). We interpret these requirements over interaction traces h. At each step, propositions are evaluated from predicates on the tool call $\alpha = (t, a)$ and, when needed, its observed outcome o. Formulas can thus relate proposed actions to preceding calls and results, and require successful completion before termination.

In our example in Fig. 1, booking travel requires a preceding distance check, the resulting distance restricts the permitted travel mode, and the requested booking must eventually succeed. For origin x and destination y, let $\mathsf{checked_distance}(x, y)$ denote a `check_distance` tool call, and $\mathsf{booked_train}(x, y)$ and $\mathsf{booked_flight}(x, y)$ the corresponding booking calls. The distance-check prerequisite P1 can be expressed in LTL as

$$\neg\big(\mathsf{booked_train}(x, y) \vee \mathsf{booked_flight}(x, y)\big)\ \mathbf{W}\ \mathsf{checked_distance}(x, y).$$

Here, $\mathbf{W}$ denotes 'weak until', which means that bookings are forbidden until the distance check occurs, and throughout the interaction if it never occurs. This blocks the bookings in steps 1 and 2. Crucially, each (x, y) is tracked separately, i.e., a Toronto–Chicago check cannot satisfy the prerequisite for Toronto–Montreal.

Tool results can further restrict the continuation. Let $\mathsf{distance_small}(x, y)$ hold if `check_distance` returns a distance below 500 km for that route. The prohibition on flights below 500 km in P2 can then be expressed in LTL as $\mathbf{G}\big(\mathsf{distance_small}(x, y) \rightarrow \mathbf{G}\,\neg\mathsf{booked_flight}(x, y)\big)$. Here, $\mathbf{G}$ denotes 'globally', i.e., a result below 500 km forbids *all* subsequent flight bookings for that route.

Both formulas above would permit an execution that checks the distance and then stops without making a booking, so we also express *obligations* to complete the task. Let $\mathsf{booked_train_ok}(x, y)$ and $\mathsf{booked_flight_ok}(x, y)$ denote booking calls whose outcomes confirm success. For the requested route, the requirement is $\mathbf{F}\big(\mathsf{booked_train_ok}(x, y) \vee \mathsf{booked_flight_ok}(x, y)\big)$, where $\mathbf{F}$ denotes 'eventually' and requires a successful booking before termination. An activated obligation that remains unfulfilled is called an *open duty*. Here, the requested booking is an open duty from the start, remains open after the distance check in step 3 and gets fulfilled by the confirmed booking in step 5.

These examples illustrate how our temporal specification captures both *safety*, by excluding forbidden behavior, and *progress*, by requiring obligations to be fulfilled before termination. Grounding

propositions in tool calls and their outcomes enables runtime monitoring. However, tool arguments may range over unbounded domains, so we parameterize the propositions arguments and bind them to concrete entities at runtime. Preserving these bindings is essential, as ignoring (x, y) would make a short-distance result for one route prohibit flights on every route. To mitigate this, we use SMT to check that the tool calls' arguments are consistent with the proposition bindings.

Argument Constraints via SMT. We express *argument constraints* as predicates on the arguments a of a call $\alpha = (t, a)$, which may also depend on information from the interaction trace h. Each constraint applies to calls identified by its associated proposition. In our example, satisfying the distance-check does not by itself permit a business-class booking, since P3 also requires the agent to establish an urgent reason and the absence of an economy seat for that route.

We encode argument constraints using SMT, which decides the satisfiability of first-order formulas modulo theories. In our implementation, we use Z3 (de Moura & Bjørner, 2008) with quantifier-free formulas over strings and linear integer arithmetic. SMT can directly express argument restrictions, enabling a solver to check whether multiple restrictions are jointly satisfiable and efficiently find satisfying values without fully enumerating the argument domain. Our argument constraints use a domain-specific language that is evaluated directly on proposed arguments and has an SMT encoding for satisfiability checks and symbolic argument search (App. C).

For the Toronto–Chicago flight in Fig. 1, let $\texttt{business_allowed}(h, x, y)$ indicate whether h establishes both an urgent reason and the absence of an economy seat for route (x, y). P3 constrains calls satisfying $\mathsf{booked_flight}(x, y)$ via the constraint

$$a[\texttt{class}] = \texttt{business} \rightarrow \texttt{business_allowed}(h, x, y),$$

where $a[\texttt{class}]$ denotes the booking-class argument represented as a string-valued SMT variable, and $\texttt{business_allowed}(h, x, y)$ is a fixed Boolean value for the given history and route. The distance check in step 3 satisfies P1, and its 700 km result does not trigger P2's flight prohibition. However, no urgent reason has been established, so $\texttt{business_allowed}(h, \texttt{Toronto}, \texttt{Chicago})$ is false. The business-class booking in step 4 therefore violates the constraint.

3.2 Building MENTOR Offline

To build MENTOR, we first translate natural-language manuals into the specifications of Sec. 3.1 (⟨C3⟩), to compute templates that provide the guidance required by ⟨C1⟩ (Fig. 2, top right).

Autoformalization. We use a multi-agent refinement loop with three LLM agents: a translator, an evaluator, and an orchestrator (Fig. 3). The *translator* takes the manual and the tool schema and formalizes the requirements in the specification language defined above. The *evaluator* generates test traces from the manual, consisting of tool calls α_i and their outcomes o_i, and labels them as compliant or non-compliant with the manual. A deterministic conflict checker then compares the proposed specification's verdicts on these traces with the evaluator's proposed labels and returns disagreements to first the translator to revise the specification. The translator can contest a label, by providing an explanation grounded in the manual, which the evaluator reviews. The *orchestrator* coordinates these revisions and resolves remaining disagreements referring to the manual. The loop ends when the specification agrees with the evaluated traces or reaches a fixed iteration limit.

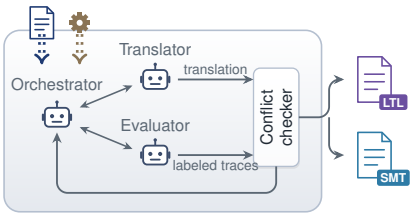

Figure 3: Autoformalization.

Note that if the loop terminates successfully, it provides some evidence of consistency between the generated specification and the manual, but it does not prove that it captures every requirement correctly. However, formalizing natural-language requirements is difficult even for human experts (Greenman et al., 2023; Czepa et al., 2019). Given that there exists no perfect method for this task, we believe that our loop provides a practical way to iteratively refine the specification and will become increasingly reliable as LLMs and autoformalization techniques improve.

Template Construction. The LTL specification φ defines compliance for interaction traces h. Shielding, however, requires determining which tool calls must be blocked or advised *during* interaction, before the trace is complete. Towards this, we adopt a standard game-theoretic approach, which automatically maps φ into a finite two-player game between the agent and the environment (see Fig. 4 for an example) (Pnueli & Rosner, 1989). At an agent vertex (circle), the agent chooses a call, which

leads to an environment vertex (square), at which the environment chooses its outcome. Each vertex abstracts the part of the history h relevant for the specification into residual LTL formulas, i.e., the requirements that the remaining interaction must still satisfy (black). Calls violating a requirement lead to a sink $\perp$ (red), from which no compliant continuation exists, and `done` leads to the accepting vertex (✓, green) if and only if no duty is open.

To obtain a shielding mechanism from the above translation, we employ a recent method (Anand et al., 2023; 2026b) that automatically computes a *strategy template* over a given game. At each agent vertex, the template marks (i) *unsafe* tools (red, dashed), which violate a requirement immediately or later unavoidably, and (ii) *progressing* tools (orange, solid), which advance the fulfillment of open duties. In Fig. 4, `done` is unsafe at v_0, v_1 and v_2 because the booking duty is still open, but it is progressing at v_3 because the confirmed booking fulfills it.

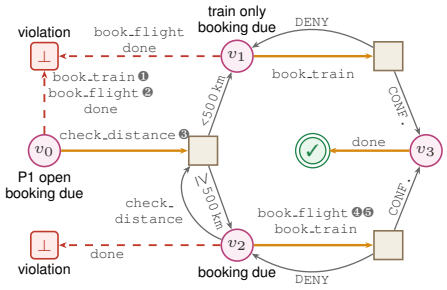

Figure 4: Game graph and template (excerpt) for the example from Fig. 1.

Intuitively, the template turns the global requirements into local constraints that let MENTOR block unsafe calls and advise progress toward compliance. It is also permissive, because the agent may choose freely any safe tool, including non-progressing calls. MENTOR interferes with the agent's preferences only where compliance requires it, and reminds of the open duties with its advice so that the agent can choose when to address them. However, the template only ranges over tools. The admissibility of their *arguments* requires additional SMT-based checks. In Fig. 4, booking a business-class flight (step 4 from Fig. 1) is progressing for the template in v_2 but blocked by the argument constraint P3. The next section discusses how this is handled by MENTOR at runtime.

3.3 Applying MENTOR at Runtime

The template-annotated game (Fig. 4) provides the *context-dependent* behavior of MENTOR. At runtime, the executed tool calls and outcomes are followed through the game, and the template of the current vertex determines unsafe and progressing calls. MENTOR additionally checks the *arguments* of proposed calls against the SMT constraints and, when a call is blocked, uses the solver to compute admissible values from the history h. Together, this yields the evaluator of Fig. 2 (bottom).

Two further components make this deterministic evaluator applicable to real agent interactions: (i) a labeler LLM that grounds propositions in the natural-language task description and tool outputs (the runtime part of ⟨C3⟩), and (ii) two shielding mechanisms, a prompt shield and a token shield, that deliver the evaluator's verdicts and advice to the agent (⟨C4⟩). We describe both below.

Labeler LLM. The tool-call evaluator operates on structured data, applying predicates to arguments a and outcomes o in h to evaluate propositions such as $\mathsf{booked_flight}(x, y)$ and argument constraints such as P3. However, the task description o_0 and free-text tool outputs, e.g. from `check_distance`, are natural language. MENTOR uses a *labeler LLM* to convert both into structured events according to the specification, on which the tool-call evaluator deterministically makes shielding decisions.

Before the first tool call, the labeler turns o_0 into one or more *pseudo tool calls*. The specification declares their names, arguments, and a plain-text extraction rule per argument. For Fig. 1, the task becomes `request_travel(Toronto, Chicago, class=business, urgent=false)`. The predicates of the temporal requirements bind x = `Toronto`, y = `Chicago` to this event, which activates the route's booking duty, and no urgent reason is given which makes $\texttt{business_allowed}(h, x, y)$ false.

Further, for each tool output marked as natural language, the specification declares a small label vocabulary. After a tool call, the labeler maps the returned text to exactly one label or none, and stores it in o_i for propositions to read like any other value. In Fig. 1, `check_distance` is labeled as ≥500 km, determining that $\mathsf{distance_small}(x, y)$ does not hold and the agent moves to v_2 in Fig. 4.

Prompt shield. The tool-call evaluator enforces its decision through the tool call output, and the prompt shield delivers the verdict and advice to the agent in the form of text along with the tool outcome. If a call $\alpha = (t, a)$ is blocked, it is not executed, and the agent receives as feedback the violated excerpt from the manual and a suggested tool call. The suggestion may be either the

template-indicated missing prerequisite call at the current vertex, or an SMT-repaired call for a violated argument constraint. Alternatively, if α is allowed, it is executed. However, if duties remain open, its result is augmented with a reminder listing the open duties and a progressing call. The call `done` is refused while duties remain open, or until a preset limit on the number of calls is reached.

As an example, consider that at v_0 of Fig. 4 the agent calls `book_train` (step 1 of Fig. 1). This call is unsafe at v_0 as `check_distance` is a prerequisite, so the agent receives the feedback:

```
[SHIELD] Blocked book_train(origin="Toronto", destination="Chicago").
- Policy rule for trip Toronto->Chicago: Check the distance before any booking. (P1)
  First: check_distance(origin="Toronto", destination="Chicago")
```

The prompt shield relies on agents following its feedback, which may be difficult for small open-weight models as shown in our experiments in Sec. 4. If the next-token distribution is accessible, MENTOR can additionally apply a token shield to enforce the same constraints at the token level.

Token shield. LLM agents express tool calls as token sequences, so we modify the next-token distribution during generation of the tool name t and arguments a. We do this by adapting the technique from STARS (Anand et al., 2026b), which uses a strategy template to modify an RL policy's action distribution, assigning unsafe actions probability zero and increasing the probability of progressing actions.

Before each model call, MENTOR derives unsafe and progressing tool names and argument values from the current tool-call evaluation. Since tool names precede arguments, a name is considered unsafe only if every call with that tool name is unsafe. At v_0 in Fig. 4, `book_train`, `book_flight`, and `done` are unsafe, and `check_distance` is progressing. For arguments, SMT identifies unsafe candidate values from the history h and suggested calls from prior advice. At v_2, `business` from the task description is unsafe under P3, while `economy` from the advised call is progressing. Each name and value is tokenized according to the model's tool-call format to identify its token sequence, so that we do not change the token distribution outside of tool calls.

Let V be the token vocabulary, $\boldsymbol{\tau}_{\text{full}}$ the token context, and $p(\tau \mid \boldsymbol{\tau}_{\text{full}})$ the next-token probability for $\tau \in V$. Since tool names may span several tokens and share prefixes, e.g., `book_flight` and `book_train`, we mask a token if the tokens written so far followed by τ can continue into unsafe names alone and into no safe one. For argument values, we only mask tokens completing an identified unsafe value, to avoid overrestriction. Tokens that continue a progressing call are *boosted*, including prefixes shared with safe, non-progressing alternatives. The modified distribution is

$$\widetilde{p}(\tau \mid \boldsymbol{\tau}_{\text{full}}) = \begin{cases} 0, & \text{if } \tau \text{ is masked,} \\ p(\tau \mid \boldsymbol{\tau}_{\text{full}}) + \eta b_\tau, & \text{if } \tau \text{ is boosted,} \\ p(\tau \mid \boldsymbol{\tau}_{\text{full}}), & \text{otherwise,} \end{cases}$$

renormalized before sampling. Here, $\eta > 0$ controls the boost strength and b_τ a counter that starts at 1 for each progressing tool call and increases for each step it is not called; if several progressing tools share τ, the largest counter is used. At v_2, the agent may still call `check_distance`, but each such call further boosts `book_flight`.

After the tool call has been generated, it is still evaluated through the deterministic tool call evaluator and its advice returned to the LLM agent, because the modified distribution does not guarantee compliance on its own. That is because each argument is constrained separately during generation, so unsafe combinations of arguments may still be generated.

4 EXPERIMENTS

We implemented MENTOR and evaluated its performance on three compliance benchmarks. We used Claude Opus 5 for the autoformalization loop, and tried several open-weights and proprietary models as LLM agents and as labelers. For the SMT solver, we used Z3 (de Moura & Bjørner, 2008). All experiments were run on a cluster with nodes of 2x Intel Xeon Gold 5317 CPUs, 2x Nvidia A100 GPUs and 1024 GB RAM. See App. D for the exact model versions and all hyperparameters used.

Benchmarks. We evaluated MENTOR on MANTRA (Anand et al., 2026a), SOP-Bench (Nandi et al., 2026) and AGENTORCA (Li et al., 2025). Each benchmark contains several domains with a procedural manual, tool schema, tasks, and a compliance evaluator. We evaluated MENTOR on 4 domains from MANTRA (2 original and 2 adapted from τ^2-bench), 2 domains from SOP-Bench, and 2 domains from AGENTORCA. However, we observed that SOP-Bench's evaluator does not read

the arguments of tool calls and misses argument-based constraint violations. For this reason, we did not include SOP-bench in our main results, and refer to App. D.3 for further details.

Baselines. We compare the compliance of agents shielded with both versions of MENTOR (prompt and token shield) against unshielded agents and, on MANTRA, three other baselines. DYNAGUARD provides a specialized LLM that judges the compliance of tool calls at runtime. PROGENT provides a specialized LLM that dynamically creates constraints during the interaction, which are then deterministically checked against the tool calls. AGENTSPEC provides a DSL in which humans write constraints that are then enforced deterministically at runtime, and a manually written labeler that uses fixed keywords and regex patterns. We write the constraints and labeling rules for AGENTSPEC using MENTOR's autoformalization loop (see App. D.4), and in Table 14 we show that it is comparable with AGENTSPEC's own LLM-based rule generator that uses 10% of the test cases as a validation set.

Metric. We ran 5 independent runs of each task per agent and report $\mathbf{Pass\hat{}5}$, the probability that all 5 runs of a task succeed, averaged over the entire benchmark. In App. D, we also report **Pass@1** and **Pass@5**, the average single-run success rate and the probability that at least one of 5 runs of a task succeeds, respectively.

Results. Table 1 summarizes the results for both benchmarks. MENTOR consistently improves compliance for every agent, proprietary and open-weights, in comparison to unshielded agents, and outperforms the baselines. We observe that the prompt-based shield is most effective for stronger agents that can act on its feedback. On MANTRA, it barely helps Llama 3.1 (see App. D.6 for an example), but the token-based shield instead improves it from below 0.1 to 0.30–0.42 in three domains. The baselines are less effective, often due to their lack of guidance towards progress, resulting in agents repeating blocked calls over and over (see App. A.2 for examples).

Impact of labeler and autoformalization. In Table 1 the autoformalization was done by Claude Opus 5, and the labeler was gpt-4.1 for the open-weights models and gpt-4.1-mini for the proprietary models. To understand the role of these choices, we conducted ablation studies on MANTRA in App. D.4, varying the autoformalization and labeling models and observing their impact on compliance. The results in Table 12 reveal that strong models are necessary for autoformalization using our loop, otherwise compliance drops. Similarly, the results in Table 10 show that open-weights models struggle to label successfully, but even small proprietary models such as gpt-4.1-mini are satisfactory. However, a small open-weights model can be easily finetuned to match this performance and act as a generalizable labeler, as we demonstrate with Qwen3-4B in Table 3 in App. B, where we trained it on gpt-4.1 labels from one domain and tested it on others.

Cost and latency. The autoformalization is performed once per manual and incurs a one-time cost, after which the generated constraints can be reused for all tasks within that domain. For this step, we used a 100 USD per month Claude subscription. The most expensive labeler, gpt-4.1, cost between 0.3 and 1.2 US cents per MANTRA test case (see Table 17 in App. D.5), whereas using a fine-tuned open-weights labeler incurs zero cost per test case. Regarding latency, the prompt shield's overhead over not using a shield is negligible, while our token shield, implemented as a vllm logit processor, incurs varied overhead per model but is still consistently lower than 2 out of 3 baselines that it outperforms (see Table 15 in App. D.5).

5 DISCUSSION, LIMITATIONS AND FUTURE WORK

Here, we discuss the limitations of MENTOR and propose directions for future work.

Autoformalization. MENTOR requires the natural language manual to be translated into formal specifications. This process is difficult both for humans and LLMs, and our loop requires strong LLMs to produce satisfactory results and does not guarantee correctness, especially for ambiguous rules that may require careful interpretation. Investigating better autoformalization techniques is a promising direction for future work that will improve MENTOR.

Labeler. MENTOR requires a labeler LLM to ground the natural-language entities mentioned in the task prompt and tool outputs into formal variables to be used by the shield. While we saw that this is a simple task for modern LLMs, even small finetuned open-weights models, it still introduces a layer of variability to the shield. It would be interesting to explore how well deterministic labelling techniques can replace the labeler LLM.

Agents: L3.1 = Llama 3.1, Q3 = Qwen3, Q3.6 = Qwen3.6, G4 = Gemma 4

(a) MANTRA	Operations (N=47)				Sneaky Sasquatch (N=43)				τ^2-retail (N=57)				τ^2-airline (N=44)			
Guard	L3.1	Q3	Q3.6	G4	L3.1	Q3	Q3.6	G4	L3.1	Q3	Q3.6	G4	L3.1	Q3	Q3.6	G4
None	0±0	28±7	62±7	55±7	7±4	44±8	56±8	42±8	5±3	16±5	37±6	18±5	34±7	30±7	50±8	41±7
AGENTSPEC	0±0	19±6	**77±6**	57±7	0±0	42±8	49±8	40±8	9±4	33±6	32±6	18±5	34±7	36±7	43±8	48±8
PROGENT	6±4	23±6	70±7	57±7	7±4	37±7	49±8	49±8	7±3	16±5	37±6	16±5	36±7	39±7	48±8	34±7
DYNAGUARD	4±3	19±6	40±7	40±7	5±3	42±8	53±8	47±8	9±4	21±5	23±6	23±6	36±7	25±7	39±7	34±7
Ours (prompt)	21±6	**64±7**	70±7	**77±6**	9±4	**77±7**	**65±7**	**79±6**	18±5	26±6	**39±7**	**25±6**	36±7	57±8	**64±7**	**66±7**
Ours (token)	**38±7**	55±7	66±7	57±7	**42±8**	37±7	58±8	65±7	**30±6**	**40±7**	18±5	**25±6**	**57±8**	**61±7**	55±8	50±8

(b) MANTRA	Operations		Sneaky Sasquatch		τ^2-retail		τ^2-airline	
Guard	GPT-5.4 mini	Haiku 4.5	GPT-5.4 mini	Haiku 4.5	GPT-5.4 mini	Haiku 4.5	GPT-5.4 mini	Haiku 4.5
None	62±7	64±7	49±8	58±8	21±5	23±6	34±7	55±8
Ours (prompt)	**77±6**	**72±7**	**63±7**	**72±7**	**40±7**	**26±6**	**61±7**	**64±7**

(c) AGENTORCA	Bank (N=51)				Healthcare (N=39)			
Guard	L3.1	Q3	Q3.6	G4	L3.1	Q3	Q3.6	G4
None	16±5	24±6	45±7	29±6	31±7	21±7	72±7	46±8
Ours (prompt)	**59±7**	**84±5**	**84±5**	**82±5**	**67±8**	77±7	87±5	**85±6**
Ours (token)	43±7	63±7	**84±5**	67±7	59±8	**79±7**	**90±5**	74±7

Table 1: **Results: Pass^5** ± standard error (%) over the N test cases of a domain. (a) Open-weights agents on MANTRA against the baselines. (b) Proprietary agents on MANTRA domains. (c) Open-weights agents on AGENTORCA. None denotes no shield, and the highest value in each column is bolded.

Natural language. MENTOR acts only on the tool calls the agents generate, and cannot ensure correctness of their natural language outputs. As such, it is not applicable to settings requiring interaction with humans (for instance, Barres et al. (2025)). Extending our shield to such settings is an interesting avenue for future work.

Token shield. The token shield must be reimplemented for different LLMs, tokenizers, or tool-call harnesses, as it relies on the specific tool calling format used during generation. Otherwise, it may harm performance by causing unexpected token generations.

6 CONCLUSION

We present MENTOR, a runtime shield for tool-using LLM agents to guide them through complex workflows described in procedural manuals. By combining strategy templates from temporal requirements with argument constraints encoded in SMT, MENTOR can act during inference to block unsafe tool calls that violate the manual and advise LLMs on how to make progress towards their assigned tasks. Using MENTOR on several open-weights and proprietary agents, we show that it effectively boosts their performance and makes them more reliable in compliance-critical settings.

AI USE STATEMENT

In this work, we used generative AI tools for implementation, for the autoformalization as per the pipeline, to support qualitative and thematic data analysis, and interpret results. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments. Furthermore, formulation of mathematical claims, assistance with critical ingredients for proving mathematical claims, assistance in the writing of proofs, and cleaning and reformatting datasets are not applicable to this work. Additionally, we used generative AI tools to create or modify scientific figures or images, create or edit software code, draft parts of the supplementary material, and identify related literature. We have reviewed all AI-assisted work. We checked critical parts of implemented code, did testing on the code, and checked the scripts that were used to produce the experimental results. 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 illustrates how verification-based shields can complement LLM agents to increase their reliability when following procedural manuals. However, MENTOR cannot guarantee compliant behavior at all times, so it should not substitute caution and additional safety measures in safety-critical domains. We hope that our work encourages further research into shielding increasingly stronger LLM agents for trustworthiness and reliability.

REPRODUCIBILITY STATEMENT

We provide all our code and parameters at https://anonymous.4open.science/r/mentor-llm-shield/. While the exact outputs are not reproducible due to the use of stochastic, proprietary LLMs, we believe one can obtain qualitatively similar results to us by running our code with the same parameters.

REFERENCES

Benjamin Aminof, Giuseppe De Giacomo, Aniello Murano, and Sasha Rubin. Specifying agent strategy spaces via ltl synthesis. In *Proceedings of the 23rd International Conference on Principles of Knowledge Representation and Reasoning*, pp. 36–45, 2026.

Chenyang An, Sam Bayless, Stefano Buliani, Darion Cassel, Byron Cook, Duncan Clough, Rémi Delmas, Nafi Diallo, Ferhat Erata, Nick Feng, et al. A neurosymbolic approach to natural language formalization and verification. *arXiv preprint arXiv:2511.09008*, 2025.

Ashwani Anand, Satya Prakash Nayak, and Anne-Kathrin Schmuck. Synthesizing permissive winning strategy templates for parity games. In *International Conference on Computer Aided Verification*, pp. 436–458. Springer, 2023.

Ashwani Anand, Ivi Chatzi, Ritam Raha, and Anne-Kathrin Schmuck. Mantra: Synthesizing smt-validated compliance benchmarks for tool-using llm agents. *arXiv preprint arXiv:2605.06334*, 2026a.

Ashwani Anand, Satya Prakash Nayak, Ritam Raha, and Anne-Kathrin Schmuck. Follow the stars: Dynamic ω-regular shielding of learned probabilistic policies. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems*, pp. 2978–2986, 2026b.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. tau2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. Nl2tl: Transforming natural languages to temporal logics using large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 15880–15903, 2023.

Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pp. 383–396. Springer, 2023.

Christoph Czepa, Amirali Amiri, Evangelos Ntentos, and Uwe Zdun. Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a

controlled experiment on understandability. *Software and Systems Modeling*, 18(6):3331–3371, 2019.

Giuseppe De Giacomo, Antonio Di Stasio, Francesco Fuggitti, and Sasha Rubin. Pure-past linear temporal and dynamic logic on finite traces. In *IJCAI*, pp. 4959–4965, 2020.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Yogesh K Dwivedi, Mohamed Youssef Ibrahim Helal, Ibrahim A Elgendy, Mousa Ahmed Albashrawi, Laurie Hughes, Mazen Shawosh, Vincent Dutot, and Il Jeon. Artificial intelligence agents and agentic systems in hospitality and tourism: challenges, opportunities and research agenda. *International Journal of Contemporary Hospitality Management*, 2025.

Francesco Fuggitti, Tathagata Chakraborti, et al. Nl2ltl-a python package for converting natural language (nl) instructions to linear temporal logic (ltl) formulas. In *AAAI*, pp. 16428–16430, 2023.

Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 163–173, 1980.

Fatemeh Ghezloo, Mehmet Saygin Seyfioglu, Rustin Soraki, Wisdom O Ikezogwo, Beibin Li, Tejoram Vivekanandan, Joann G Elmore, Ranjay Krishna, and Linda Shapiro. Pathfinder: A multi-modal multi-agent system for medical diagnostic decision-making applied to histopathology. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 23431–23441, 2025.

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pp. 854–860. IJCAI/AAAI, 2013.

Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. Little tricky logic: Misconceptions in the understanding of LTL. *The Art, Science, and Engineering of Programming*, 7(2): 7:1–7:37, 2023.

Monte Hoover, Vatsal Baherwani, Neel Jain, Khalid Saifullah, Joseph J Vincent, Chirag Jain, Melissa Rad, C Bruss, Ashwinee Panda, and Tom Goldstein. Dynaguard: A dynamic guardian model with user-defined policies. In *International Conference on Learning Representations*, volume 2026, pp. 87822–87859, 2026.

Kai-Chieh Hsu, Haimin Hu, and Jaime F. Fisac. The safety filter: A unified view of safety-critical control in autonomous systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 7(Volume 7, 2024):47–72, 2024. ISSN 2573-5144. doi: https://doi.org/10.1146/annurev-control-071723-102940.

Adharsh Kamath, Sishen Zhang, Calvin Xu, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Enforcing temporal constraints for llm agents. *arXiv preprint arXiv:2512.23738*, 2025.

Seongjae Kang, Taehyung Yu, and Sung Ju Hwang. Policyguide: From guarding one action to guiding the whole workflow for policy-compliant llm agents. *arXiv preprint arXiv:2608.19861*, 2026.

Bettina Könighofer, Roderick Bloem, Rüdiger Ehlers, and Christian Pek. Correct-by-construction runtime enforcement in ai–a survey. In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, pp. 650–663. Springer, 2022.

Zekun Li, Shinda Huang, Jiangtian Wang, Nathan Zhang, Antonis Antoniades, Wenyue Hua, Kaijie Zhu, Sirui Zeng, Chi Wang, William Yang Wang, et al. Sopbench: Evaluating language agents at following standard operating procedures and constraints. *arXiv preprint arXiv:2503.08669*, 2025.

Jason Xinyu Liu, Ziyi Yang, Ifrah Idrees, Sam Liang, Benjamin Schornstein, Stefanie Tellex, and Ankit Shah. Grounding complex natural language commands for temporal tasks in unseen environments. In *Conference on Robot Learning*, pp. 1084–1110. Pmlr, 2023.

Zhi Ma, Cheng Wen, Zhexin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification–automated translation of software requirements to ltl via hierarchical semantics decomposition using llms. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1208–1220. IEEE, 2025.

Adam Mondl, Matthew Maisel, and John H Brock. Autoformalization of agent instructions into policy-as-code. *arXiv preprint arXiv:2606.26649*, 2026.

Subhrangshu Nandi, Arghya Datta, Rohith Nama, Udita Patel, Nikhil Vichare, Indranil Bhattacharya, Shivam Asija, Arushi Gupta, Giuseppe Carenini, Jing Xu, et al. Sop-bench: Complex industrial sops for evaluating llm agents. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 9604–9615, 2026.

Firoz Mohammed Ozman. Systematic literature review on the rise of agentic ai in enterprise operations. *International Journal of Frontiers in Science and Technology Research*, 8(2):001–015, 2025.

Nils Palumbo, Sarthak Choudhary, Jihye Choi, Guy Amir, Prasad Chalasani, and Somesh Jha. Formal policy enforcement for real-world agentic systems. *arXiv preprint arXiv:2602.16708*, 2026.

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 179–190, 1989. doi: 10.1145/75277.75293.

Vikas Reddy, Sumanth Reddy Challaram, and Abhishek Basu. Reason less, verify more: Deterministic gates recover a silent policy-violation failure mode in tool-using llm agents. *arXiv preprint arXiv:2607.07405*, 2026.

Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Securing AI agents with privilege control. *arXiv preprint arXiv:2504.11703v3*, 2025. URL `https://arxiv.org/abs/2504.11703v3`.

Tao Tu, Mike Schaekermann, Anil Palepu, Khaled Saab, Jan Freyberg, Ryutaro Tanno, Amy Wang, Brenna Li, Mohamed Amin, Yong Cheng, et al. Towards conversational diagnostic artificial intelligence. *Nature*, 642(8067):442–450, 2025.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pp. 2938–2950, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720253. doi: 10.1145/3744916.3764546.

Jun Wang, David Smith Sundarsingh, Jyotirmoy V Deshmukh, and Yiannis Kantaros. Conformalnl2ltl: Translating natural language instructions into temporal logic formulas with conformal correctness guarantees. *arXiv preprint arXiv:2504.21022*, 2025.

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

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, et al. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning. *arXiv preprint arXiv:2406.09187*, 2024.

Milan Zdravković, Hervé Panetto, and Georg Weichhart. Ai-enabled enterprise information systems for manufacturing. *Enterprise Information Systems*, 16(4):668–720, 2022.

A Extended Related Work

A.1 Comparison of Shielding Features

We summarize the discussion of related shielding approaches for LLM agents from Sec. 1 in Tab. 2 using the following criteria:

Auto-extracted: ● if an LLM produces the checked constraints without hand-written rules, ◐ LLM-assisted constraint creation, – if the shield checks no explicit constraints.

Table 2: Comparison of MENTOR with existing LLM guarding approaches: ● supported, ◐ partially supported, ○ not supported, – not applicable.

	Specification	Runtime check					Agent support	
Shield	Auto-extracted	Determ-inistic	Temporal order	Argument values	Oblig-ations	Free-text outputs	Guidance	Token-level
LLM-based								
GuardAgent (Xiang et al., 2024)	●	○	○	◐	○	●	◐	○
DynaGuard (Hoover et al., 2026)	–	○	◐	◐	○	●	◐	○
PolicyGuide (Kang et al., 2026)	●	○	◐	◐	◐	●	●	○
Rule-based								
AgentSpec (Wang et al., 2026)	◐	●	◐	◐	◐	◐	◐	○
Progent (Shi et al., 2025)	◐	◐	○	●	○	◐	◐	○
Agent-C (Kamath et al., 2025)	◐	●	●	●	◐	◐	◐	●
MENTOR (ours)	●	●	●	●	●	●	●	●

Deterministic: ● if the compliance decisions are deterministic (without an LLM judge), ◐ if an LLM edits constraints during the run.

Temporal order, argument values: ◐ if left to an LLM judge or hand-coded as general-purpose code.

Obligations: ● if the shield tracks open duties and does not let the agent end the task while one is open, ◐ if open duties can only be detected at the end (by hand-coded checks or an error flag).

Free-text outputs: ● if the shield can decide based on natural-language tool results, ◐ if this needs hand-written string checks or only changes the policy indirectly.

Guidance: ● if a blocked call is answered with a compliant alternative or the action that unlocks it, ◐ if only the reason for the block is given.

Token-level: enforcement during decoding, for open-source models.

Tab. 2 shows that MENTOR covers all criteria, while every existing technique lacks several of them. Furthermore, PolicyGuide and Agent-C are not publicly available, so we cannot compare to them in our experiments. In the following, we show how the lack of these features results in non-compliant traces that slip through these shields, as observed in our experiments.

A.2 Failure Traces of other Shielding Techniques w.r.t. ⟨C1⟩–⟨C2⟩

This section shows, on traces from our experiments, how the missing features in Tab. 2 let non-compliant traces slip through existing shields, and why these shields often prevent the agent from completing its task at all. We group the observed failures by their cause: the shield has no access to the history of the task (⟨C1⟩, context-dependence), it blocks a call without telling the agent how to make progress (⟨C1⟩, guidance), or it does not constrain the arguments of a call (⟨C2⟩).

Most examples come from one Operations task, the running example of Fig. 1: the user asks for a 1.5-hour flight within North America. The policy forbids business class on North American flights under 6 hours. If the agent requests business class anyway, the booking tool does not book; it returns a policy violation and suggests to give the reason "Urgent Business Need", with which the violation becomes only a warning. Unshielded agents request business class in 11 of 20 runs and are then tempted to *invent* such an urgent reason and report it in a `flag_policy_violation` call, although the user never stated an urgent need. Such traces are clearly non-compliant, but hard to catch for existing shielding methods. We ran each shield 20 times on this task (4 models, 5 runs each).

No access to the history (⟨C1⟩, context-dependence). As outlined in the introduction, whether a tool call complies depends on the history of the task, e.g., installing software is allowed only after its tier has been checked. A shield must therefore take this history into account.

PROGENT checks each call only against its own arguments; a rule cannot refer to an earlier call or its result. Its only memory is an LLM that may rewrite the whole policy after each allowed call, and

```
Task:  for U-88104, book a one-way economy trip RDU ->
BOI on 2026-10-03; reuse passenger, bags and payment of
reservation R-551208.

    [0] get_reservation_details(R-551208)      BLOCKED
    [1] search_direct_flight(10-03,RDU,BOI)    BLOCKED
    [2] search_onestop_flight(10-03,RDU,BOI)   BLOCKED
    [3] search_direct_flight(...)              BLOCKED
    [4] get_reservation_details(...)           BLOCKED
 [5-44] the same three calls round-robin       BLOCKED
   [45] search_direct_flight(...)              BLOCKED
        [HARNESS] 45 tool calls made; call done now.
[46-49] the same three calls round-robin       BLOCKED
        Step cap reached:  no book_reservation, no done.

Guard's reason at step 1:  "The agent violated the rule
by not listing the action details and obtaining explicit
user confirmation before making a tool call.  The rule
states 'Before taking any actions that update the booking
database (...)  you must list the action details and
obtain explicit user confirmation'."
```

Figure 5: Agent stalling: DYNAGUARD on Gemma 4 executing τ^2-airline, case 39. All 50 calls, including read-only lookups, are blocked with the same reason, although the cited rule only applies to calls that update the booking database.

these rewrites mostly damage the policy. Over all Operations tasks (4 models, 5 runs each), 273 of the 902 rewrites removed a tool, so every later call to that tool was denied, and 364 loosened a constraint. Only once did the same call go from blocked to allowed, and only because a rewrite deleted the policy's rule that three verification IDs are required. In our setup, PROGENT also always allows `done`, so duties that are still open at the end are never checked.

AGENTSPEC can use the history, since its rules are Python functions that read the earlier calls and their results (99 of its 132 rules over all domains do so); e.g., one rule blocks `install_software` until `check_software_tier` has returned a result for that software.

DYNAGUARD's judge sees the history of executed calls, so its verdict can correctly change once the agent makes progress: in one Llama 3.1 run, `install_software` is blocked because no vulnerability scan was run, the agent runs the scan, and the same call is then allowed. However, the judge cannot reliably tell from this history whether a duty is already done: in one Qwen3 run, it blocks `done` 157 times because "the scan report was not archived", although `archive_scan_report` succeeded at step 3.

MENTOR, in contrast, tracks in its strategy template which duties are open, and does not allow `done` while a duty is still open.

No guidance, so the agent stalls ($\langle$C1$\rangle$, guidance). The most common failure mode we observed is *stalling*: the shield blocks a call but offers the agent no admissible alternative, so the agent repeats the blocked call, or tries other blocked calls, until its step budget is exhausted and the task ends without the required booking.

AGENTSPEC blocks the request to book business class in 13 of 20 runs on the Operations task, but a block does not tell the agent what to do instead. In one Qwen3 run, the business request is blocked; the agent then flags and reviews a booking ID that does not exist, both calls are blocked, and the agent ends the task without booking economy. Only 8 of 20 runs pass.

DYNAGUARD explains why a call was blocked, but its judge sees only the executed calls, not its own earlier blocks. A blocked call therefore leaves no trace in the history: when the agent repeats it, the judge receives the identical prompt and, at temperature 0, returns the identical verdict, so a wrong block can never be undone by the agent. In one Gemma 4 run on the Operations task, the correct economy booking is blocked 35 times in a row with the reason "booking a business-class flight instead of economy class", and the run ends without a booking. Stalling is not specific to this task, nor to argument constraints. Figure 5 shows the same loop on τ^2-airline: DYNAGUARD blocks all 50 calls of Gemma 4, even read-only lookups, because the agent has not asked the user for confirmation. The policy requires this confirmation only before calls that change the booking database, so the verdict itself is wrong; no compliant action of the agent can unlock a lookup, the identical reason is repeated 50 times, and the task ends at the step limit without a booking.

MENTOR, in contrast, tells the agent which call unlocks a blocked call or, as detailed next, suggests the blocked call with repaired arguments, so the agent always has a compliant call that makes progress.

No argument constraints (⟨C2⟩). Blocking the business request and the invented urgent reason on the Operations task requires constraints on the arguments of a call, not only on its position in the trace.

PROGENT does not stop this behavior, because its policy only checks argument types (e.g., `travel_class` must be economy, business or first). In fact, PROGENT even amplifies it: the agent requests business class in 16 of 20 runs, against 11 of 20 without a shield. For example, in one Qwen3 run, the agent requests business class, receives the violation and flags it with the reason "Urgent Business Need"; no call is blocked.

AGENTSPEC has rules for both steps and blocks the business request in 13 of 20 runs, but, as discussed above, its blocks leave the agent without a way to complete the task.

DYNAGUARD blocks the business request in 12 of 20 runs and the invented reason in 10 runs, but its LLM judge is inconsistent: in 2 runs the invented reason is accepted, and in 2 Llama 3.1 runs the business request goes through. Only 4 of 20 runs pass.

MENTOR, in contrast, blocks the business request and suggests the same call with `travel_class="economy"`, so the agent does not receive the suggestion to give an urgent reason. In the one run where the agent still writes an urgent reason, our shield blocks the flag, because this reason is allowed only if the user's request states an urgent need.

B FINE TUNING THE LABELER

In the main experiments reported in Table 1 the labeler is GPT-4.1. Here we replace it with Qwen3-4B-Instruct-2507, a 4-billion-parameter open-weight model that runs on the same GPU as the agent. We trained it by *distillation*: the small model (the *student*) learns to reproduce the labels of GPT-4.1 (the *teacher*). All training targets are GPT-4.1 outputs; we use no human labels. In this section, "MANTRA domains" means the two synthetic domains of MANTRA, Operations and Sneaky Sasquatch, and "τ^2 domains" means the two MANTRA τ^2 domains: retail and airline.

Task format. Each training example is one labeler call as the shield issues it (Sec. 3.3). For the task description, the prompt contains the request schema of the specification (pseudo tools, their arguments, types, defaults and extraction rules) and the text; the target is a JSON array of pseudo tool calls. For a tool output, the prompt contains the label vocabulary (a closed list of allowed answers), a short guidance text and the output; the target is one vocabulary item or `NONE`. On τ^2, the text holds the user's messages so far and the assistant's last message, so each user turn gives one call. Since the schema or vocabulary is always part of the prompt, the student has to learn to read and fill a schema, not to memorize a domain.

Data. We use three sources. (i) *GPT-4.1 labels from past runs.* Every shield run caches the GPT-4.1 labels. For each cached label we rebuild the prompt and the text and keep the pair only if its recomputed cache key (a hash of prompt and text) equals the cache file name. We then remove duplicates and prompts that received two different labels in different runs. This gives 239 examples from the MANTRA domains, 6,408 from the τ^2 domains and 70 from SOP-Bench. (ii) *Synthetic domains.* We wrote a list of 48 themes that avoid all benchmark themes (e.g., pet boarding reservations). For each theme, Codex (OpenAI's coding agent), run in an empty read-only sandbox without access to any benchmark file, writes a specification; the 48 specifications have 188 pseudo tools in total. It then writes 3,934 texts together with the label it intends, including hard cases such as no request, missing fields, typos and distractors. GPT-4.1 labels each text with the shield's own prompt, and we keep the 3,133 examples on which its label matches the intended one (strings match if their word sets overlap by at least 0.6 Jaccard similarity). (iii) *Paraphrases.* Codex rewrites 160 real messages from MANTRA and SOP-Bench three times each and must keep every fact. We keep the 441 rewrites whose GPT-4.1 label equals the label of the original message. In (ii) and (iii) the target is always the GPT-4.1 label; the agreement check removes texts whose meaning is unclear, but also hard cases on which GPT-4.1 is wrong.

Splits. We test on a benchmark family that the student never saw in training. The families are MANTRA, τ^2 and SOP-Bench. When family F is held out, all real examples of F form the test set; training and validation use the real examples of the other families, their paraphrases and all synthetic

Table 3: $\mathbf{Pass\hat{}5}$ on the MANTRA synthetic domains (top) and the MANTRA τ^2 domains (bottom), without a guard (None) and with our prompt shield, with gpt-4.1 or the fine-tuned Qwen3-4B as the labeler. Best arm per row in bold (ties: all bold).

		$\mathbf{Pass\hat{}5}$		
Domain	Model	None	Ours (gpt-4.1)	Ours (Qwen3-4B FT)
Operations	Llama 3.1	0.00	0.21	**0.30**
	Qwen3	0.28	**0.64**	0.60
	Qwen3.6	0.62	**0.72**	**0.72**
	Gemma 4	0.55	**0.77**	0.70
Sneaky Sasquatch	Llama 3.1	0.07	0.09	**0.14**
	Qwen3	0.44	**0.77**	0.65
	Qwen3.6	0.56	**0.65**	0.58
	Gemma 4	0.42	**0.79**	0.58
τ^2-retail	Llama 3.1	0.05	0.18	**0.28**
	Qwen3	0.16	0.28	**0.37**
	Qwen3.6	0.39	0.40	**0.46**
	Gemma 4	0.18	**0.25**	0.23
τ^2-airline	Llama 3.1	0.34	0.36	**0.43**
	Qwen3	0.32	0.57	**0.66**
	Qwen3.6	0.50	**0.64**	**0.64**
	Gemma 4	0.41	**0.66**	0.64

data, but no paraphrase of F. Validation takes 10% of the groups (a case, a task, or a synthetic batch), so that related examples stay in one split. We cap each (family, domain, labeling task) at 1,200 training and 60 validation examples. Each training example on the task description gets, with probability 0.3, a copy in which all tool and argument names are in camelCase, so that the student takes names from the prompt (examples whose rules refer to name patterns are not copied). We trained two *folds* (a fold is one split with one family held out), each giving one LoRA adapter: τ^2 held out (4,133 train / 278 validation / 6,408 test) and MANTRA held out (7,814 / 540 / 239). In the shield, each domain uses the adapter that did not see its family.

Training. We train LoRA adapters (rank 32, $\alpha = 64$, dropout 0.05) on all linear layers, i.e., 66M trainable parameters (1.6% of the model). We train for 2 epochs with learning rate 10^{-4}, a cosine schedule with 3% warm-up, effective batch size 16, bfloat16, and a maximum length of 16,384 tokens (no example was longer). The loss is computed on the target tokens only, and thinking mode is off. We evaluate the validation loss every 200 steps and keep the checkpoint with the lowest value. We use AdamW with the Hugging Face Trainer defaults, batch size 1 per device with 16 gradient accumulation steps, and seed 42 (PEFT 0.21.0, Transformers 4.57.6, PyTorch 2.11.0). Training takes 1.6 h for the τ^2 fold and 7.4 h for the MANTRA fold on one GPU.

Shield-level results. We run the prompt shield with the fine-tuned labeler in the same setup as with GPT-4.1 (4 agents, 5 trials per task); Table 3 gives $\mathbf{Pass\hat{}5}$ per domain and agent, for the MANTRA domains (top) and the τ^2 domains (bottom). The fine-tuned labeler is served by vLLM on the same A100 GPU as the agent. Averaged over all 16 (domain, agent) pairs, $\mathbf{Pass\hat{}5}$ is 0.50 with either labeler, and 0.33 without a shield. Per domain, the fine-tuned labeler is better on τ^2-retail (0.33 vs. 0.28) and τ^2-airline (0.59 vs. 0.56), worse on Sneaky Sasquatch (0.49 vs. 0.58), and within 0.01 on Operations. Thus, in these runs, the low label-level agreement on τ^2 did not lower $\mathbf{Pass\hat{}5}$; we did not analyse which labels the shield's rules depend on. With 5 trials and 43–57 tasks per domain, differences of a few points are within run-to-run noise. Each fold was trained once with one seed.

C Specification Language

This section gives the syntax and semantics of the specification language introduced in Sec. 3.1. A specification contains temporal requirements and argument constraints, together with the declarations that connect them to tool calls and their outcomes. Figure 6 summarizes its structure.

(a) Specification structure

$$
\begin{aligned}
\langle spec \rangle &::= (\{\langle type \rangle\}, \{\langle module \rangle\}, \{\langle request \rangle\}, \{\langle observation \rangle\}) \\
\langle type \rangle &::= (S, \kappa) \\
\langle module \rangle &::= (S, \{\langle proposition \rangle\}, \{\langle requirement \rangle\}, \{\langle constraint \rangle\}) \\
\langle proposition \rangle &::= (p(v), t, e) \\
\langle requirement \rangle &::= (\mathsf{kind}, \varphi, m) \\
\langle constraint \rangle &::= (p(v), e, m) \\
\langle request \rangle &::= (Q, \Sigma, d, I) \\
\langle observation \rangle &::= (t, \pi_{\text{text}}, \pi_{\text{label}}, \Lambda, I)
\end{aligned}
$$

(b) Temporal requirements via LTL

$$
\begin{aligned}
\varphi ::= {} & \top \mid \bot \mid p(v) \mid \mathbf{O}p(v) \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \\
& \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid (\varphi \,\mathbf{U}\, \varphi) \mid (\varphi \,\mathbf{W}\, \varphi)
\end{aligned}
$$

(c) Core expressions for proposition predicates and argument constraints

$$
\begin{aligned}
e ::= {} & c \mid v \mid a \mid o \mid b_v \mid e.\pi \mid e[e] \mid [e_1, \ldots, e_r] \\
& \mid \neg e \mid -e \mid (e \wedge e) \mid (e \vee e) \mid (e \bowtie e) \mid (e \odot e) \mid \mathrm{if}(e, e, e) \\
& \mid \mathrm{last}(p(v)) \mid \mathrm{exists}(p(v)) \mid \mathrm{count}(p(v)) \mid \mathrm{hist}(p(v), \pi) \\
& \mid \mathrm{present}(e) \mid \mathrm{len}(e) \mid \mathrm{contains}(e, e) \mid \mathrm{startswith}(e, e) \\
\bowtie \in {} & \{==, !=, <, \leq, >, \geq, \in, \notin\} \\
\odot \in {} & \{+, -, \times, /\}
\end{aligned}
$$

Figure 6: Abstract structure and syntax of the specification language. Braces denote finite collections. In (a), S is an entity type and κ its entity-identification rules; kind classifies a temporal requirement as a safety requirement or an obligation, and m is its source sentence. A request declaration gives a pseudo tool name Q, a typed argument schema Σ, defaults d, and extraction instructions I. An observation declaration identifies the free-text and label fields of an outcome and its label vocabulary Λ. In (c), c is a constant, π a field path, and b_v the event introducing the entity. The function meanings are given in Table 4. Auxiliary string, list, and conversion helpers are omitted from the core expression grammar.

Requirements are parameterized by concrete entities. An entity type specifies how tool arguments and outcomes identify entities; a call may identify several entities. Requirements concerning the same entity type are grouped into a *module*, which declares its propositions, temporal requirements, and argument constraints. Several modules may concern the same entity type. Within a module, the parameter v denotes the entity being considered. It may represent a tuple, such as the route (x, y) in Sec. 3.1. Requirements concerning the whole task use a common task identifier.

Each entity-identification rule associates a tool with an expression that computes an entity key from the tool's arguments and, when available, its outcome. The rule is evaluated on proposed arguments before the call is checked and again on the completed event after execution. A scalar result identifies one entity, a collection identifies each of its elements, and an undefined result identifies no entity. Entity keys are compared after conversion to strings, removal of surrounding whitespace, and case folding. Tuple-valued parameters are represented by a composite key whose components preserve the corresponding argument values.

The specification also declares the information to be extracted from natural language: schemas for pseudo tool calls representing requests, and label vocabularies for free-text tool outcomes. These declarations determine the labeler's inputs and outputs (Sec. 3.3). Each temporal requirement and argument constraint is associated with its source sentence in the manual, which can subsequently be included in the shield's feedback.

Temporal Requirements via LTL. A proposition declaration $(p(v), t, e)$ associates $p(v)$ with tool t and a Boolean predicate e. The proposition holds at an event precisely when the event calls t and e evaluates to true under the entity binding; an omitted predicate is interpreted as true. Matching the event to the entity must be expressed in e: the parameter v alone does not restrict a proposition to calls concerning that entity. For example, a proposition for route (x, y) compares the call's origin and destination with x and y.

An *action proposition* can be evaluated from the proposed call, the entity binding, and recorded history. An *observation proposition* additionally depends on the current outcome. Earlier outcomes are available through the history, whereas the current outcome becomes available only after execution. This distinction determines which requirements can be checked before a tool call.

Fix a module and an entity binding, and let $\mathcal{P}$ be the resulting set of propositions. Each recorded event induces a letter $\ell_i \subseteq \mathcal{P}$ containing precisely the propositions that hold at that event. The resulting finite word is $\sigma = \ell_0 \cdots \ell_{n-1} \in (2^{\mathcal{P}})^+$. Events at which no proposition holds retain their positions and contribute the empty letter. Formulas follow standard LTL syntax (see Fig. 6(b)). Using finite-trace semantics (Giacomo & Vardi, 2013), extended with the past operator $\mathbf{O}$, satisfaction at position $0 \le i < n$ is defined by

$$\begin{aligned}
\sigma, i \models p &\iff p \in \ell_i, \\
\sigma, i \models \neg\varphi &\iff \sigma, i \not\models \varphi, \\
\sigma, i \models \varphi \wedge \psi &\iff (\sigma, i \models \varphi) \wedge (\sigma, i \models \psi), \\
\sigma, i \models \mathbf{X}\varphi &\iff i + 1 < n \wedge (\sigma, i+1 \models \varphi), \\
\sigma, i \models \varphi\, \mathbf{U}\, \psi &\iff \exists j,\ i \le j < n : (\sigma, j \models \psi) \\
&\qquad \wedge \big(\forall k,\ i \le k < j : \sigma, k \models \varphi\big), \\
\sigma, i \models \mathbf{O}p &\iff \exists j,\ 0 \le j \le i :\ p \in \ell_j.
\end{aligned}$$

Here, the bound entity parameter is suppressed in proposition names. Boolean constants, disjunction, and implication have their usual meanings, and

$$\begin{aligned}
\mathbf{F}\varphi \equiv \top\, \mathbf{U}\, \varphi, \qquad\qquad \mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi, \\
\varphi\, \mathbf{W}\, \psi \equiv (\varphi\, \mathbf{U}\, \psi) \vee \mathbf{G}\varphi.
\end{aligned}$$

We write $\sigma \models \varphi$ iff $\sigma, 0 \models \varphi$. In particular, $\mathbf{X}\varphi$ is false at the final position: it requires a further event.

The past operator $\mathbf{O}$ expresses that a proposition holds at the current event or held earlier. It permits prerequisites to be written directly without increasing the expressive power of LTL (Gabbay et al., 1980; De Giacomo et al., 2020).

For example, with p denoting a distance check and q a booking for the same route, $\mathbf{G}(q \to \mathbf{O}p)$ expresses the prerequisite in P1. Since checking and booking are distinct tool calls, the check must precede the booking. Obligations may instead require a successful outcome: a failed booking attempt does not satisfy $\mathbf{F}\,\mathsf{booked_flight_ok}(x, y)$. An activated obligation that remains unfulfilled is an *open duty*, as in Sec. 3.1. If the manual permits the work to be waived under a condition r, that alternative must be explicit in the formula, for example $\mathbf{F}(q \vee r)$. A failed tool call does not by itself discharge an obligation.

Argument Constraints via SMT. An argument constraint associates an action proposition $p(v)$ with a Boolean expression e. Whenever a proposed call satisfies $p(v)$, its arguments must satisfy e under the current entity binding and recorded history. The condition may use outcomes of earlier calls, but cannot depend on the unavailable outcome of the proposed call. This allows constraints to express, for example, that an identifier was returned by an earlier lookup, that a required argument is present, or that an argument value is permitted by previously established facts.

The DSL in Fig. 6(c) provides field and list access, Boolean and arithmetic expressions, comparisons, conditionals, and operations on recorded history. Here, a and o denote the current call's arguments and outcome, v the bound entity, and b_v the event that introduced it. Access to o is restricted to predicates evaluated after execution. History operations inspect earlier executed events recorded for the entity and module, excluding the current event, and are initialized by the labeler LLM. Table 4 defines the functions shown in the grammar.

Expression evaluation accounts for incomplete tool outputs. A missing field or list element evaluates to null. Numeric strings are interpreted as numbers in arithmetic and numeric comparisons; ordering comparisons with non-numeric values are false. String equality ignores case and surrounding whitespace, and membership in a string is a case-insensitive substring test. Addition is numeric when both operands are numbers; otherwise, a string operand causes concatenation, with null treated as the empty string. Other invalid arithmetic operations, including division by zero, yield null. When a rule requires an earlier lookup to exist,that prerequisite must be expressed explicitly, for example using $\mathrm{exists}(p(v))$. If an applicable constraint evaluates to false, the proposed arguments do not satisfy it.

Concrete proposed arguments are checked by evaluating these expressions. For symbolic argument search, the SMT encoding introduces variables for the arguments and evaluates history-dependent

Table 4: Functions in the core expression language. History operations use earlier recorded events with the entity parameter v bound. A field path π selects information from an event's tool arguments or outcome.

Expression	Meaning
$\mathrm{last}(p(v))$	The latest earlier event satisfying $p(v)$; if none exists, its fields are undefined (null).
$\mathrm{exists}(p(v))$, $\mathrm{count}(p(v))$	Whether an earlier event satisfies $p(v)$, and the number of such events, respectively.
$\mathrm{hist}(p(v), \pi)$	The non-null values at path π from earlier events satisfying $p(v)$, in chronological order.
$\mathrm{present}(e)$	Whether e is neither null, an empty string, nor an empty list.
$\mathrm{len}(e)$	The length of a string or list; zero for an unsuitable value.
$\mathrm{contains}(e_1, e_2)$	For a list e_1, whether it contains an element equal to e_2; for a string, a case-insensitive substring test.
$\mathrm{startswith}(e_1, e_2)$	A case-insensitive string-prefix test.
$\mathrm{if}(e_1, e_2, e_3)$	e_2 when the Boolean condition e_1 holds, and e_3 otherwise.

subexpressions to fixed values. String, numeric, and Boolean arguments are represented by string, real, and Boolean variables, respectively. The supported Boolean, arithmetic, and string operations are then encoded using the corresponding SMT operations. This representation is used by Z3 (de Moura & Bjørner, 2008) to find argument values satisfying the proposition predicates and applicable constraints at runtime.

The SMT encoding does not represent every DSL operation exactly. List-valued arguments are represented by their lengths, and symbolic string equality is exact, unlike the normalized comparison used in concrete evaluation. Presence is encoded as nonemptiness for strings, positive length for lists, and true for other types. Case and type conversions are translated using their operands without applying the conversion, and a constraint that cannot be translated is omitted from the SMT query. These restrictions limit symbolic argument search; the proposed tool call is still subject to the concrete argument checks.

For the running example, let $B_{x,y} = \texttt{business_allowed}(h, x, y)$ be the history-dependent Boolean from Sec. 3.1. It records both an urgent reason and the absence of an economy seat for the route. For a flight booking, the symbolic conditions expressing the entity binding and P3 are

$$a[\texttt{origin}] = x \ \wedge \ a[\texttt{destination}] = y \\ \wedge \big(a[\texttt{class}] \neq \texttt{business} \ \vee \ B_{x,y}\big).$$

The route and $B_{x,y}$ are fixed by the current binding and history; the arguments are symbolic. For Toronto–Chicago in Fig. 1, no urgent reason has been established, so $B_{x,y}$ is false. The resulting constraint excludes business class while preserving the route.

D Additional Experimental Results

D.1 Setup

Metrics. For a benchmark suite with N tasks, we run each task $n = 5$ times independently and let $\alpha_i \in \{0, 1, \ldots, n\}$ denote the number of successful runs on task i. The tables below report three standard success rates:

- **Pass@1**— the average single-attempt success rate, $\frac{1}{N}\sum_{i=1}^{N} \frac{\alpha_i}{n}$. This is the standard accuracy of one independent rollout.
- **Pass@k**— the probability that at least one of k independent attempts on a task succeeds, averaged over the suite. Following the unbiased estimator of Chen et al. (**?**), Pass@k $= \frac{1}{N}\sum_{i=1}^{N}\big[1 - \binom{n-\alpha_i}{k}\big/\binom{n}{k}\big]$ for $n \geq k$. We report **Pass@5**.
- **Pass^k**— the probability that *all* k independent attempts on a task succeed, averaged over the suite (Barres et al., 2025): **Pass^k**$= \frac{1}{N}\sum_{i=1}^{N} \binom{\alpha_i}{k}\big/\binom{n}{k}$. This is a stricter consistency/reliability measure than Pass@k. We report **Pass^5**.

Table 5: Shield and harness settings per benchmark. Budgets are in tool calls. "T" is the sampling temperature and "p" the nucleus parameter; "default" is the provider's default. The token shield's decoding settings apply to all benchmarks except AgentOrca; MANTRA additionally masks `done` without a refusal limit and uses the growing boost counter b_τ.

	MANTRA	SOP-Bench	AgentOrca
Step budget / block budget	40 / 25	200 / 60	200 / 200
Harness call cap	45	–	–
`done` tool available	yes	no	no
`done` refusals before forced end	4	n/a	n/a
Continuation turns	3	1	–
Request labels	labeler	labeler	from task data
Labeler, prompt shield	GPT-4.1	GPT-4.1 mini	n/a
Labeler, token shield	GPT-4.1	GPT-4.1	n/a
Agent sampling, prompt shield	default	T 0.5	T 0, p 0.01
Agent sampling, token shield	T 0.7, p 1.0, at most 8,192 tokens		T 0, p 0.01
Boost strength η / counter b_τ	0.5 / $\min(4, 1+k)$	0.5 / 1	0.5 / 1
Trials per task	5	5	5

Table 6: Models used in the evaluation. "Parameters" is the number of stored parameters; for the mixture-of-experts (MoE) model and for Gemma 4 E4B, the number of parameters used per token is in brackets. API vendors do not publish model sizes.

Model	Parameters	Weights / serving	Role
Agents, prompt shield and baselines			
Llama 3.1 8B Instruct	8.0B	Q4_K_M, Ollama	agent
Qwen3 8B	8.2B	Q4_K_M, Ollama	agent
Qwen3.6-35B-A3B (MoE)	36.0B (3B active)	Q4_K_M, Ollama	agent
Gemma 4 E4B Instruct	8.0B (4B effective)	Q4_K_M, Ollama	agent
GPT-5.4 mini	n/a	OpenAI API	agent (MANTRA suites only)
Claude Haiku 4.5	n/a	Anthropic API	agent (MANTRA suites only)
Agents, token shield			
Llama 3.1 8B Instruct	8.0B	bf16, vLLM 0.26.0	agent
Qwen3 8B	8.2B	bf16, vLLM 0.26.0	agent
Qwen3.6-27B	27.8B	bf16, vLLM 0.26.0	agent
Gemma 4 E4B Instruct	8.0B (4B effective)	bf16, vLLM 0.26.0	agent (not on τ^2-bench)
Other roles			
Claude Opus 5	n/a	Claude Code	translator, judge, orchestrator
Claude Sonnet, GPT-5.5, GPT-5.6-luna	n/a	Claude Code / Codex CLI	translator ablation
Qwen3.8 27B	27.3B	Q4_K_M, Ollama	translator ablation; Progent policy model
GPT-4.1 mini	n/a	OpenAI API	labeler (prompt-shield ablation); τ^2-bench user simulator
GPT-4.1	n/a	OpenAI API	labeler (prompt and token shield; prompt-shield ablation); teacher for fine-tuning
Qwen3-4B-Instruct-2507 + LoRA	4.0B	bf16, vLLM 0.23.0	fine-tuned labeler
DynaGuard-8B	8.2B	F16, Ollama	baseline guard

Models. Table 6 lists all models and their roles. The prompt shield runs the open-weight agents through Ollama with 4-bit weights (Q4_K_M). The token shield needs the logits, so it runs the agents through vLLM with bf16 weights; the two shields therefore do not use the same weights. For Qwen3.6, the prompt shield uses the mixture-of-experts model Qwen3.6-35B-A3B, and the token shield uses the dense model Qwen3.6-27B. The two shields also differ in sampling settings and in the labeler (Table 5).

D.2 FULL RESULTS ON MANTRA

Table 7: **Pass@1** on the four MANTRA domains for open-weight (white) and proprietary (gray) agents.

Domain	Model	None	AgentSpec	Progent	DynaGuard	Ours (prompt)	Ours (token)
Operations ($N = 47$)	Llama 3.1	0.05±0.02	0.16±0.04	0.18±0.04	0.14±0.04	0.57±0.05	**0.58±0.06**
	Qwen3	0.52±0.06	0.54±0.05	0.55±0.05	0.38±0.06	**0.75±0.06**	0.70±0.06
	Qwen3.6	0.77±0.05	**0.85±0.05**	0.78±0.06	0.51±0.07	0.82±0.05	0.74±0.06
	Gemma 4	0.74±0.05	0.78±0.05	0.72±0.05	0.53±0.06	**0.83±0.05**	0.73±0.06
	GPT-5.4 mini	0.72±0.06	–	–	–	**0.82±0.05**	–
	Claude Haiku 4.5	0.76±0.05	–	–	–	**0.78±0.06**	–
Sneaky Sasquatch ($N = 43$)	Llama 3.1	0.20±0.05	0.11±0.03	0.21±0.05	0.22±0.05	0.45±0.05	**0.61±0.06**
	Qwen3	0.63±0.06	0.60±0.06	0.57±0.06	0.63±0.06	**0.84±0.05**	0.68±0.05
	Qwen3.6	0.76±0.05	0.69±0.06	0.70±0.06	0.74±0.06	**0.81±0.05**	0.77±0.05
	Gemma 4	0.61±0.06	0.60±0.06	0.61±0.06	0.63±0.06	**0.80±0.06**	0.74±0.06
	GPT-5.4 mini	0.71±0.05	–	–	–	**0.77±0.06**	–
	Claude Haiku 4.5	0.71±0.06	–	–	–	**0.80±0.06**	–
τ^2-retail ($N = 57$)	Llama 3.1	0.13±0.04	0.12±0.04	0.13±0.04	0.12±0.04	0.36±0.05	**0.46±0.06**
	Qwen3	0.37±0.05	0.53±0.06	0.34±0.05	0.29±0.06	0.49±0.06	**0.57±0.06**
	Qwen3.6	0.58±0.05	0.58±0.05	**0.60±0.05**	0.27±0.06	0.56±0.06	0.45±0.05
	Gemma 4	0.36±0.05	0.44±0.05	0.34±0.05	0.31±0.06	**0.48±0.05**	**0.48±0.06**
	GPT-5.4 mini	0.39±0.05	–	–	–	**0.53±0.06**	–
	Claude Haiku 4.5	0.29±0.06	–	–	–	**0.41±0.06**	–
τ^2-airline ($N = 44$)	Llama 3.1	0.42±0.07	0.43±0.07	0.42±0.07	0.38±0.07	0.47±0.07	**0.63±0.07**
	Qwen3	0.43±0.07	0.47±0.07	0.44±0.07	0.32±0.07	**0.73±0.06**	0.70±0.07
	Qwen3.6	0.61±0.07	0.63±0.06	0.57±0.07	0.39±0.07	**0.74±0.06**	0.63±0.07
	Gemma 4	0.55±0.07	0.55±0.07	0.50±0.07	0.37±0.07	**0.73±0.06**	0.55±0.07
	GPT-5.4 mini	0.47±0.06	–	–	–	**0.65±0.07**	–
	Claude Haiku 4.5	**0.67±0.07**	–	–	–	**0.67±0.07**	–

Table 8: **Pass@5** on the four MANTRA domains for open-weight (white) and proprietary (gray) agents.

Domain	Model	None	AgentSpec	Progent	DynaGuard	Ours (prompt)	Ours (token)
Operations ($N = 47$)	Llama 3.1	0.17±0.06	0.40±0.07	0.36±0.07	0.30±0.07	**0.81±0.06**	**0.81±0.06**
	Qwen3	0.81±0.06	0.81±0.06	**0.83±0.06**	0.53±0.07	**0.83±0.06**	0.77±0.06
	Qwen3.6	0.85±0.05	**0.87±0.05**	0.85±0.05	0.68±0.07	**0.87±0.05**	0.85±0.05
	Gemma 4	0.87±0.05	**0.91±0.04**	0.89±0.05	0.70±0.07	0.87±0.05	0.83±0.06
	GPT-5.4 mini	0.81±0.06	–	–	–	**0.85±0.05**	–
	Claude Haiku 4.5	**0.89±0.05**	–	–	–	0.85±0.05	–
Sneaky Sasquatch ($N = 43$)	Llama 3.1	0.37±0.07	0.30±0.07	0.49±0.08	0.37±0.07	**0.79±0.06**	**0.79±0.06**
	Qwen3	0.84±0.06	0.77±0.07	0.86±0.05	0.84±0.06	0.88±0.05	**0.91±0.04**
	Qwen3.6	**0.91±0.04**	0.79±0.06	0.84±0.06	0.86±0.05	**0.91±0.04**	**0.91±0.04**
	Gemma 4	0.79±0.06	**0.86±0.05**	0.79±0.06	0.79±0.06	0.81±0.06	0.79±0.06
	GPT-5.4 mini	**0.95±0.03**	–	–	–	0.86±0.05	–
	Claude Haiku 4.5	**0.88±0.05**	–	–	–	0.86±0.05	–
τ^2-retail ($N = 57$)	Llama 3.1	0.28±0.06	0.18±0.05	0.23±0.06	0.14±0.05	0.54±0.07	**0.58±0.07**
	Qwen3	0.56±0.07	0.67±0.06	0.54±0.07	0.33±0.06	0.63±0.06	**0.68±0.06**
	Qwen3.6	**0.79±0.05**	**0.79±0.05**	**0.79±0.05**	0.33±0.06	0.72±0.06	0.65±0.06
	Gemma 4	0.58±0.07	0.67±0.06	0.58±0.07	0.35±0.06	**0.70±0.06**	0.67±0.06
	GPT-5.4 mini	0.61±0.07	–	–	–	**0.63±0.06**	–
	Claude Haiku 4.5	0.42±0.07	–	–	–	**0.60±0.07**	–
τ^2-airline ($N = 44$)	Llama 3.1	0.55±0.08	0.50±0.08	0.50±0.08	0.39±0.07	0.59±0.07	**0.68±0.07**
	Qwen3	0.50±0.08	0.64±0.07	0.50±0.08	0.36±0.07	**0.86±0.05**	0.73±0.07
	Qwen3.6	0.73±0.07	0.77±0.06	0.73±0.07	0.39±0.07	**0.80±0.06**	0.68±0.07
	Gemma 4	0.64±0.07	0.59±0.07	0.66±0.07	0.41±0.07	**0.77±0.06**	0.59±0.07
	GPT-5.4 mini	**0.73±0.07**	–	–	–	0.70±0.07	–
	Claude Haiku 4.5	**0.75±0.07**	–	–	–	0.70±0.07	–

D.3 EVALUATION ON SOP-BENCH

In addtion to the MANTRA and the AgentOrca benchmark reported on in Table 1 in Sec. 4, we also ran both shields on two SOP-Bench domains, patient intake and customer service, with the same four open-weight agents and 5 runs per task. While analysing the runs, we found that the benchmark's scorer does not read the arguments of tool calls and therefore accepts traces that violate the SOP. We first document this flaw and then report the results.

The scorer ignores tool arguments. In the patient-intake domain, SOP-Bench can score a task as passed even when the agent breaks the SOP. The agent must call five lookup tools (e.g., `validateInsurance`) and then `registerPatient`, passing it the five lookup results. The SOP allows registration only if, among other conditions, the insurance status is `valid`. Two parts of the benchmark let wrong calls pass. First, the mock `registerPatient` ignores the status values it receives and returns the stored outcome for the patient. Second, when the agent gives no JSON answer block (none of our agents did), the scorer takes its prediction from the tool results in the trace; it never reads the arguments of a call or the agent's final answer. For example, for patient P100008 the correct behaviour is to run the lookups, which return insurance status `invalid`, prescription status `invalid` and pharmacy check `no`, and then to call `registerPatient` with these values; the tool then returns `failure`, the ground truth. Qwen3 without a shield instead put all six calls into one reply, so it had to fill in the statuses of `registerPatient` before any lookup had run. It guessed `valid`, `valid` and `yes`, which tells the system that the patient is eligible. The task still passed: the lookups returned the true values, `registerPatient` ignored the wrong statuses and returned the stored `failure`, and the scorer compared only these tool results with the ground truth. In another passed task (patient P100017), Llama 3.1 sent the tool names as the status values (e.g., `insurance_validation="validateInsurance"`). Over the 5 runs of the 35 tasks without a shield, 88 of Qwen3's 129 passes (68%) and 22 of Llama 3.1's 46 passes (48%) contain a `registerPatient` call whose statuses differ from the lookup results; for Gemma 4 and Qwen3.6 this holds for 1 of 169 and 0 of 152 passes. If we count only passes whose arguments match the lookup results, the TSR of Qwen3 without a shield falls from 0.74 to 0.23 (41 of 175), below its TSR with our prompt shield (0.43, Table 9). With the prompt shield, none of the 428 passes (over all four models) has this problem, because one of the shield's rules blocks any `registerPatient` call whose statuses differ from the lookup results.

Results. Table 9 reports SOP-Bench's own task success rate and $\mathbf{Pass\hat{}5}$ on the fixed sample of 35 tasks per domain, keeping in mind that the unshielded numbers on patient intake are inflated by the scorer flaw described above. Both shields bring Qwen3.6 and Gemma 4 to a perfect score on patient intake, and the prompt shield lifts Qwen3 on customer service from 0.25 to 0.70 ($\mathbf{Pass\hat{}5}$ from 0.20 to 0.40). Llama 3.1, however, loses under the prompt shield on both domains: on patient intake it ends with an open duty in 34 of the 35 tasks, and on customer service it proposes on average 68 calls per task, 61 of which are blocked, because it repeats the blocked call instead of following the advice. This is the stalling behaviour of App. D.6; the token shield, which removes the blocked call from the vocabulary, brings the same agent to 0.38 on patient intake, above its unshielded score. Llama 3.1's successes are, however, not repeatable: under the token shield it solves 83% of the patient-intake tasks in at least one run but no task in all five, and its $\mathbf{Pass\hat{}5}$ is 0 in five of its six cells, which the standard deviation of 0.20 over runs also reflects. The one regression of the token shield is Gemma 4 on customer service (0.94 to 0.47, with a standard deviation of 0.19 over runs).

Table 9: compliance scores on two SOP-Bench domains for open-weight agents. TSR is SOP-Bench's own task success rate which is flawed. $\mathbf{Pass\hat{}5}$ results are the companion to Table 1 in Sec. 4.

		TSR			**Pass@5**			$\mathbf{Pass\hat{}5}$		
Domain	Model	None	Ours (prompt)	Ours (token)	None	Ours (prompt)	Ours (token)	None	Ours (prompt)	Ours (token)
Patient intake ($N = 35$)	Llama 3.1	0.26±0.04	0.02±0.03	**0.38±0.20**	0.77	0.09	**0.83**	**0.00**	**0.00**	**0.00**
	Qwen3	**0.74±0.09**	0.43±0.05	0.46±0.07	**1.00**	0.66	0.77	**0.20**	0.06	0.11
	Qwen3.6	0.87±0.03	**1.00±0.00**	**1.00±0.00**	**1.00**	**1.00**	**1.00**	0.71	**1.00**	**1.00**
	Gemma 4	0.97±0.01	**1.00±0.00**	**1.00±0.00**	**1.00**	**1.00**	**1.00**	0.86	**1.00**	**1.00**
Customer service ($N = 35$)	Llama 3.1	**0.21±0.02**	0.09±0.02	0.18±0.03	0.37	0.29	**0.46**	**0.11**	0.00	0.00
	Qwen3	0.25±0.05	**0.70±0.04**	0.59±0.04	0.37	0.83	**0.86**	0.20	**0.40**	0.34
	Qwen3.6	0.81±0.04	**0.94±0.03**	0.90±0.05	0.97	**1.00**	0.97	0.46	**0.77**	0.74
	Gemma 4	**0.94±0.04**	0.86±0.03	0.47±0.19	**1.00**	**1.00**	0.74	**0.80**	0.63	0.14

D.4 Ablation Studies

We vary the three LLM-dependent parts of MENTOR one at a time: the labeler that grounds the request and the free-text tool outputs at runtime, the model that autoformalizes the manual, and, for the AgentSpec baseline, the technique that writes its rules.

Table 10: Labeler ablation: **Pass^3** of the prompt shield on two MANTRA domains and two agents; agent and tasks fixed and only labeler varied. Open-weight labelers run through Ollama, GPT-4.1 and GPT-4.1 mini through the OpenAI API, the fine-tuned Qwen3-4B through vLLM (App. B).

	Pass^3			
	Sneaky Sasquatch ($N = 43$)		τ^2-retail ($N = 57$)	
Labeler	Qwen3	Gemma 4	Qwen3	Gemma 4
gpt-4.1-mini (OpenAI)	0.70±0.07	0.62±0.07	0.39±0.06	**0.33±0.06**
Qwen3-4B (fine-tuned, vLLM)	0.68±0.07	0.60±0.07	**0.44±0.06**	0.28±0.05
Gemma 4 (Ollama)	0.07±0.04	0.07±0.04	0.42±0.07	0.28±0.06
gpt-oss (Ollama)	0.02±0.02	0.07±0.04	0.35±0.06	0.26±0.06
Llama 3.1 (Ollama)	0.02±0.02	0.05±0.03	**0.44±0.07**	0.28±0.06
Qwen3 (Ollama)	0.05±0.03	0.07±0.04	0.35±0.06	0.26±0.06
Qwen3.6 (Ollama)	0.02±0.02	0.09±0.04	0.40±0.07	0.25±0.06
gpt-4.1 (OpenAI)	**0.80±0.06**	**0.79±0.06**	0.35±0.06	0.31±0.06

Table 11: Labeler ablation: **Pass^5** on all MANTRA with the GPT-4.1 mini and the GPT-4.1 labeler.

		Prompt shield (gpt-4.1-mini)	Prompt shield (gpt-4.1)	Token shield (gpt-4.1)
Domain	Model	**Pass^k5**	**Pass^k5**	**Pass^k5**
Operations ($N = 47$)	Llama 3.1	0.28±0.07	0.21±0.06	**0.38±0.07**
	Qwen3	0.62±0.07	**0.64±0.07**	0.55±0.07
	Qwen3.6	**0.72±0.07**	0.70±0.07	0.66±0.07
	Gemma 4	0.72±0.07	**0.77±0.06**	0.57±0.07
Sneaky Sasquatch ($N = 43$)	Llama 3.1	0.14±0.05	0.09±0.04	**0.42±0.08**
	Qwen3	0.67±0.07	**0.77±0.07**	0.37±0.07
	Qwen3.6	0.63±0.07	**0.65±0.07**	0.58±0.08
	Gemma 4	0.58±0.08	**0.79±0.06**	0.65±0.07
τ^2-retail ($N = 57$)	Llama 3.1	0.21±0.05	0.18±0.05	**0.30±0.06**
	Qwen3	0.32±0.06	0.26±0.06	**0.40±0.07**
	Qwen3.6	**0.42±0.07**	0.39±0.07	0.18±0.05
	Gemma 4	**0.26±0.06**	0.25±0.06	0.25±0.06
τ^2-airline ($N = 44$)	Llama 3.1	0.41±0.07	0.36±0.07	**0.57±0.08**
	Qwen3	0.59±0.07	0.57±0.08	**0.61±0.07**
	Qwen3.6	0.61±0.07	**0.64±0.07**	0.55±0.08
	Gemma 4	0.64±0.07	**0.66±0.07**	0.50±0.08

Table 12: Autoformalization-model ablation: **Pass^3** of the prompt shield on two MANTRA domains and three agents, with the specification written by different translator and evaluator models.

Domain	Model	Claude Opus 5	Claude Sonnet	gpt-5.5	gpt-5.6-luna	Qwen3.8
Operations ($N = 47$)	Llama 3.1	0.38±0.06	0.09±0.04	**0.47±0.07**	0.13±0.05	0.11±0.05
	Qwen3	0.64±0.07	0.34±0.07	**0.68±0.07**	0.23±0.06	0.34±0.07
	Gemma 4	0.75±0.06	0.68±0.07	**0.79±0.06**	0.36±0.07	0.55±0.07
τ^2-retail ($N = 57$)	Llama 3.1	**0.28±0.05**	0.25±0.06	0.09±0.04	0.16±0.05	0.14±0.05
	Qwen3	0.39±0.06	**0.46±0.07**	0.39±0.07	0.28±0.06	0.42±0.07
	Gemma 4	0.33±0.06	0.32±0.06	**0.35±0.06**	0.30±0.06	**0.35±0.06**

Translation Techniques. AGENTSPEC rules are written by hand, so the AGENTSPEC baseline is only as good as the person or method that writes its rules. We therefore write the rules with two techniques: our own autoformalization loop, and the LLM-based rule generator that AGENTSPEC proposes (Wang et al., 2026). Both produce the same two files per domain. The first is a rule file in AGENTSPEC's language. Each rule names a *trigger* (the tool that is about to be called), a *check* (a list of predicates, each possibly negated) and an *enforcement* action, which runs when all predicates of the check hold. The second is a Python module that defines the predicates. Each predicate receives the user's message, the arguments of the planned call and the earlier calls with their results. The released rule language has no *or*, no comparison operators such as $=$ or $<$, and cannot refer to a tool argument, so all thresholds, state and ordering logic live in the Python module. Rules are checked only before a call. Hence the only way to require that something happens by the end of a task is to block `done`, the tool the agent calls to end the task. The τ^2 domains have no `done` tool, so there such requirements cannot be expressed at all.

Our loop (column "AgentSpec (our loop)"). We use the same two-agent loop and the same guidelines that write MENTOR's specifications (Sec. 3.2), with checks for the new target language: the translator checks that the rule file parses and the module imports, and the judge runs its traces through the rules. Both agents are Claude Opus 5. They receive the policy document, the tool reference and the source code of the tools, the initial database with the fields removed that no tool returns, and AGENTSPEC's grammar, interpreter and published rule examples. They see no test cases, results or MENTOR specifications. The judge writes its own labelled example traces from the document, runs them through the rules, and answers ACCEPT or REVISE with a list of defects. The loop stops at ACCEPT or after 6 rounds. All four domains ended with ACCEPT, after 2 to 4 rounds, with 35 to 59 rules and 19 to 38 predicates per domain (Table 13). The rules cover fewer policy sentences in the τ^2 domains, mainly because of the missing `done` tool. We checked the transcripts of all agents: no agent read a file outside its inputs. We then froze the rule files; no rule was changed after results were seen.

AGENTSPEC*'s generator (column "AgentSpec (LLM-gen)").* We reimplement the generator from the released AGENTSPEC code. It asks the model for one rule per requirement sentence, with the authors' query *"Help to generate agentspec rule for the requirement: ⟨sentence⟩"*. The system prompt is the authors' template, with small formatting changes, their three example predicates and their answer format. Each answer is one JSON record with a rule name, a trigger, the predicates as Python code and an enforcement action. As in the paper, there is no refinement loop, and the rules were not run on any trace; we only checked that the rule file parses and the module imports, and fixed only names and missing imports. Our benchmarks do not come with requirement lists, so one GPT-4.1 call per domain copies the requirement sentences from the policy document (22 to 87 per domain), and we checked the result against the document. The method also shows the model 10% of the risky test cases as examples. We draw 10% of the cases of each domain, at least 3, with a fixed seed. For each case we show the user request, the case's grading checks in plain words, and one passing and one failing trajectory of the unshielded agent, when the unshielded runs contain one (3 of the 20 examples lack one of the two). We remove these cases from every column when scoring, so all columns use the same held-out cases. We made five changes to the method. (i) The paper uses OpenAI o1; we use Claude Opus 5, as for our loop, so the authoring model is not what differs between the two techniques. (ii) One agent per domain answers all queries in one session and is told to treat them independently; the released code makes a separate API call per sentence. (iii) The model also receives the tool signatures, descriptions and source code, because a predicate must know the names of result fields, and a short description of the rule format with some advice on writing rules. (iv) The examples include the grading checks in words, which the paper's examples do not. (v) The released function that prints a rule produces text that does not parse, so we assemble the rule file ourselves, and we expand a trigger of `any`, which the grammar rejects, into one rule per tool.

Everything after authoring is the same for both columns: the AGENTSPEC interpreter, the agents, the 5 trials per task and the scoring. Both enforcement actions that the generator uses (`stop` and `user_inspection`) block the call, as does our loop's `skip`. The two columns thus differ only in how the rules were written. Table 13 summarizes the two rule sets and Table 14 their compliance rates on the held-out cases.[1] Neither technique is consistently better for AGENTSPEC: the generator's

[1]The cases shown to AGENTSPEC's generator and removed from every arm are case_009, case_020, case_040 and case_042 on Operations, Sneaky Sasquatch and τ^2-airline; and case_009, case_040, case_042, case_049 and case_057 on τ^2-retail.

rules win in 8 rows, ours in 9, with 3 ties, and both stay far below MENTOR, which is the best arm in 18 of the 20 rows. The generator's rule sets on the τ^2 domains are large (125–231 rules) because a trigger of `any` is expanded into one rule per tool.

Table 13: The AGENTSPEC rule sets. Our loop: rounds of the judge until ACCEPT, rules, predicates, and policy sentences covered out of those the translator identified. AGENTSPEC's generator: requirement sentences (one query each), rules after expanding the trigger `any` into one rule per tool, and cases shown to the model as examples (Shown).

	Our loop				AGENTSPEC's generator		
Domain	Rounds	Rules	Predicates	Sentences	Requirements	Rules	Shown
Operations	4	35	38	21/27	22	22	4
Sneaky Sasquatch	4	44	19	37/41	30	30	4
τ^2-retail	4	59	27	22/36	39	174	5
τ^2-airline	2	46	33	31/51	47	125	4

Table 14: **Pass^5** with AGENTSPEC rules written by AGENTSPEC's LLM rule generator (LLM-gen) and by our autoformalization loop (our loop), next to the unshielded agent (None) and our shield (Ours), on the held-out cases (N per domain). Best arm per row in bold (ties: all bold).

Domain	Model	None	AgentSpec (LLM-gen)	AgentSpec (our loop)	Ours
Operations ($N = 43$)	Llama 3.1	0.00	0.05	0.00	**0.26**
	Qwen3	0.28	0.26	0.19	**0.58**
	Qwen3.6	0.60	**0.77**	**0.77**	0.70
	Gemma 4	0.56	0.63	0.56	**0.70**
Sneaky Sasquatch ($N = 39$)	Llama 3.1	0.08	0.03	0.00	**0.15**
	Qwen3	0.44	0.23	0.41	**0.69**
	Qwen3.6	0.54	0.51	0.51	**0.62**
	Gemma 4	0.44	0.26	0.41	**0.56**
τ^2-retail ($N = 52$)	Llama 3.1	0.04	0.08	0.08	**0.19**
	Qwen3	0.15	0.27	**0.33**	0.31
	Qwen3.6	0.33	0.38	0.31	**0.40**
	Gemma 4	0.15	**0.25**	0.15	**0.25**
τ^2-airline ($N = 40$)	Llama 3.1	0.33	0.35	0.33	**0.40**
	Qwen3	0.30	0.28	0.35	**0.55**
	Qwen3.6	0.47	0.35	0.42	**0.57**
	Gemma 4	0.42	0.35	0.45	**0.60**

D.5 COST AND LATENCY

D.6 CASE STUDY: PROMPT SHIELD VERSUS TOKEN SHIELD

The prompt-based shield only works if the agent acts on its message. We show one MANTRA task (Sneaky Sasquatch, case 018), in which the message is correct but Llama 3.1 8B does not follow it. With the token-based shield, the same agent solves the task in 3 of 5 runs.

Rule and task. Rule `Inv_S1` requires checking the inventory before giving food to a patient. The user asks to treat patient PT-6620, whose magnesium is low. The task has 9 checks; among them, `check_inventory(item_type="foods")` must be called, and one `give_food` with a suitable food (Milk, Relish or Cheese) must follow it.

No shield. The agent gives Milk without checking the inventory. The run passes 7 of 9 checks.

Prompt shield. We describe the single run with temperature 0. The agent first calls `give_food(patient_id="PT-6620", food_name="Milk")`. The shield blocks it and tells the agent which calls to make first, among them `check_inventory(item_type="foods")`:

```
[SHIELD] Blocked give_food(patient_id="PT-6620",
food_name="Milk").
```

Table 15: Seconds per proposed tool call (blocked calls included), mean ± standard deviation over the 5 trials. †Served by vLLM, comparable only within the column.

		Seconds per proposed tool call					
Domain	Model	None	AgentSpec	Progent	DynaGuard	Ours (prompt)	Ours (token)†
Operations ($N = 47$)	Llama 3.1	1.56±0.08	1.30±0.15	21.04±4.00	3.90±1.55	1.26±0.05	1.57±0.06
	Qwen3	8.01±0.80	7.85±1.60	27.79±7.43	10.41±1.18	7.54±1.63	7.86±0.16
	Qwen3.6	6.15±0.15	6.18±1.00	34.08±8.00	8.87±0.36	7.07±0.28	17.60±2.85
	Gemma 4	7.39±0.24	7.26±0.99	17.49±4.67	10.79±0.41	8.00±0.32	0.71±0.02
Sneaky Sasquatch ($N = 43$)	Llama 3.1	1.24±0.05	0.82±0.22	12.22±1.95	2.21±0.39	1.14±0.13	1.49±0.08
	Qwen3	6.98±0.23	6.72±1.12	18.12±1.66	8.51±0.46	7.09±0.37	9.90±1.37
	Qwen3.6	6.64±0.62	6.33±1.47	24.28±4.35	6.99±0.40	6.54±0.25	11.45±1.89
	Gemma 4	5.83±0.42	4.70±0.85	12.95±3.15	6.63±0.31	5.83±0.21	0.59±0.04
τ^2-retail ($N = 57$)	Llama 3.1	1.36±0.15	1.08±0.17	15.01±2.91	3.33±0.48	2.05±0.12	1.87±0.04
	Qwen3	16.12±1.79	12.17±2.69	29.14±5.46	16.36±0.84	14.39±1.82	12.73±1.03
	Qwen3.6	7.09±0.45	5.82±1.36	34.97±5.74	8.58±0.28	9.93±0.37	35.91±2.69
	Gemma 4	7.94±0.19	7.72±0.97	25.16±5.68	9.34±0.20	9.01±0.25	1.08±0.06
τ^2-airline ($N = 44$)	Llama 3.1	1.95±0.49	2.70±0.48	17.63±2.20	4.18±0.18	1.96±0.17	1.59±0.00
	Qwen3	17.69±1.81	18.24±3.65	31.70±2.97	28.41±4.39	16.96±1.39	13.94±1.23
	Qwen3.6	11.88±1.00	10.31±2.06	42.00±3.33	13.57±0.53	13.91±0.42	33.36±3.39
	Gemma 4	12.48±0.50	12.19±1.75	24.81±4.97	11.64±0.32	11.51±0.20	0.97±0.06

Table 16: Tool calls the agent proposed per case, blocked calls included, mean ± standard deviation over the 5 trials.

		Proposed tool calls per case					
Domain	Model	None	AgentSpec	Progent	DynaGuard	Ours (prompt)	Ours (token)
Operations ($N = 47$)	Llama 3.1	5.9±0.3	7.2±0.7	5.6±0.4	9.6±2.5	12.3±2.0	15.9±0.6
	Qwen3	6.0±0.5	7.1±0.2	6.4±0.8	29.7±8.7	7.2±0.2	11.1±1.4
	Qwen3.6	4.9±0.1	5.4±0.3	5.2±0.1	21.9±5.0	6.4±0.4	14.8±1.0
	Gemma 4	6.8±1.2	7.4±0.8	10.1±2.3	22.9±0.5	8.8±1.1	14.7±1.5
Sneaky Sasquatch ($N = 43$)	Llama 3.1	6.6±0.1	7.6±1.3	6.2±1.0	8.4±3.0	9.6±0.9	9.8±0.7
	Qwen3	5.2±0.2	7.3±0.3	6.9±0.9	10.6±0.5	6.0±0.2	6.9±0.4
	Qwen3.6	6.3±0.3	7.4±0.6	6.5±0.4	10.7±2.2	6.3±0.2	7.0±1.1
	Gemma 4	6.0±0.4	16.6±2.0	8.6±1.2	8.2±0.9	9.1±1.5	8.2±0.7
τ^2-retail ($N = 57$)	Llama 3.1	3.9±0.3	2.0±0.4	2.4±0.7	2.2±0.6	7.0±0.5	27.0±2.2
	Qwen3	3.1±0.2	3.7±0.2	3.3±0.1	6.2±0.7	7.2±0.8	14.3±2.3
	Qwen3.6	5.6±0.4	5.7±0.4	5.6±0.2	17.0±0.9	7.0±0.5	17.1±0.9
	Gemma 4	2.7±0.1	3.2±0.1	2.8±0.1	13.0±1.6	5.5±0.0	10.8±2.1
τ^2-airline ($N = 44$)	Llama 3.1	3.6±1.1	1.3±0.1	2.2±0.5	1.2±0.0	3.8±0.6	31.9±1.0
	Qwen3	2.5±0.0	2.8±0.1	3.9±0.5	3.7±0.6	4.8±0.3	14.7±1.3
	Qwen3.6	2.5±0.1	2.7±0.2	3.0±0.2	4.7±0.4	3.7±0.4	9.5±1.2
	Gemma 4	3.1±0.2	2.9±0.3	4.4±0.9	13.8±0.2	4.4±0.4	6.2±0.7

Table 17: Cost of the labeler per domain if every label were a live API call, in thousandths of a USD per case (mUSD/case) and in USD per run over all cases of the domain, at list prices of GPT-4.1 mini (0.40/1.60 USD per 1M input/output tokens) and GPT-4.1 (2.00/8.00).

		USD, gpt-4.1-mini		USD, gpt-4.1	
Domain	cases	mUSD/case	per run	mUSD/case	per run
Operations	47	0.61	0.029	3.04	0.143
Sneaky Sasquatch	43	0.70	0.030	3.49	0.150
τ^2-retail	57	2.47	0.141	12.37	0.705
τ^2-airline	44	1.95	0.086	9.74	0.429

```
- Policy rule for patient PT-6620:  ... check what the
 inventory holds before you hand a food over.
   First:  check_inventory(item_type="foods")
```

This advice is correct: the inventory check would unblock `give_food`. The agent calls `listen_to_patient`, which the same message also suggests, and then repeats the blocked `give_food` call. The shield blocks it again, says that this exact call was already rejected, and repeats the advice. The agent then calls `talk_to_head_doctor`, which is also blocked, and makes no further tool call, even after 3 continuation turns. It never checks the inventory and never gives food. The run passes 7 of 9 checks, the same as without a shield: no rule is broken, but the patient is not treated.

Token shield. At the first call, `give_food` is already masked and `listen_to_patient` is boosted, and the agent calls `listen_to_patient`. At the second call, the shield state is the same as at the prompt-based shield's second call (only `listen_to_patient` has run). The shield masks the tool names `give_food`, `talk_to_head_doctor` and `done`, among others. The agent therefore cannot generate either of the two calls it made under the prompt-based shield, and it generates `check_inventory(item_type="foods")`. It then gives Cheese (the environment reports the deficiency as treated), sends the patient home, and calls `done`. The run passes 9 of 9 checks, with no blocked call and no shield message; the only shield text the agent sees is the fixed note in the system prompt. In all five runs (seeds), the agent calls the same sequence of tools, with no blocked call. Three runs pass all checks. The other two fail 2 checks because the agent calls `check_inventory(item_type="")`. The shield does not mask this value, because rule `Inv_S1` accepts an inventory check with any `item_type`; the task's checks require `"foods"`.