

MENTOR: PROCEDURAL SHIELDING OF TOOL-USING LLM AGENTS

Ashwani Anand^{1,*}, Ivi Chatzi^{1,*}, Ritam Raha² & Anne-Kathrin Schmuck¹

¹Max Planck Institute for Software Systems, Kaiserslautern, Germany

²TNO-ESI, Eindhoven, The Netherlands

{ashwani, ichatzi, akschmuck}@mpi-sws.org, ritam.raha@tno.nl

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 auto-formalization pipeline. At its core, MENTOR utilizes a game-based technique to ensure robustness against runtime uncertainty without over-restricting 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.

*Equal contribution.

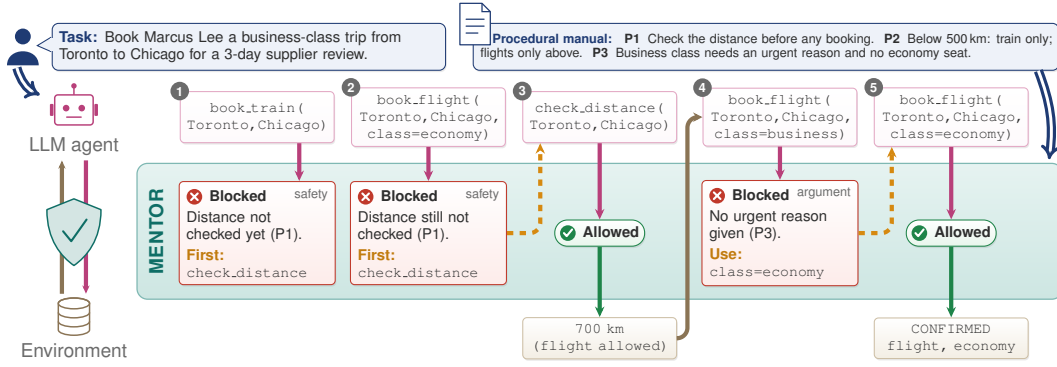


Figure 1: **Running example.** Our procedural shield MENTOR is generated from a procedural manual (top right) and applied to a travel-booking task (top left). MENTOR shields the interaction between an LLM agent (pink) and the environment (brown). MENTOR blocks tool calls (red), allows tool calls (green),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

(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* 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), whose evaluator accepts some incorrect traces because it does not check the relevant tool arguments (App. C.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. We provide all our code and parameters at <https://anonymous.4open.science/r/mentor-llm-shield/>.

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.

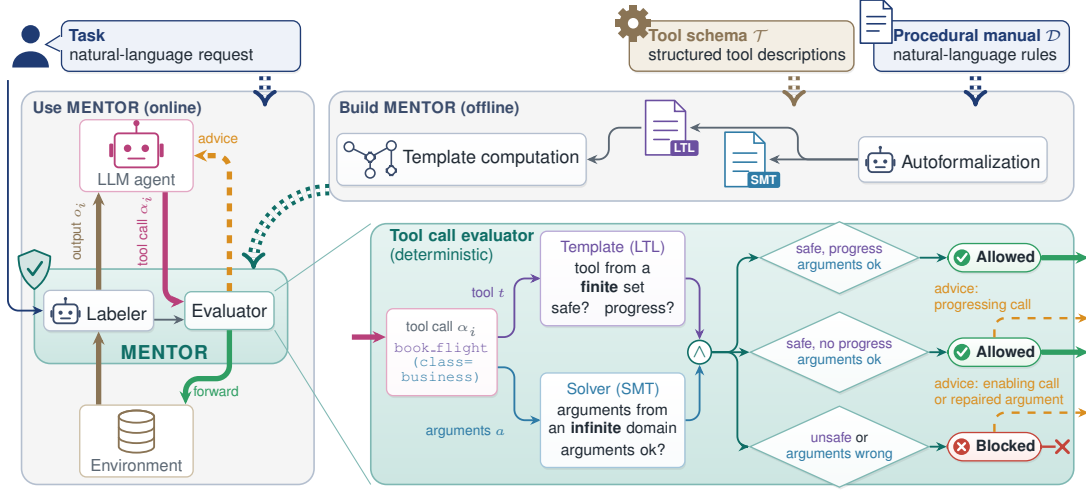


Figure 2: Overview of MENTOR. Manual and tool schema are formalized into an LTL-based template and an SMT-based solver (offline, top right), used to shield every tool call (online, left) by deterministically evaluating (bottom right) if tool calls are safe and progressing (violet, LTL), and use correct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

One line of work on LLM shielding techniques delegate compliance decisions to a guardrail LLM (Xi-ang et al., 2024; Hoover et al., 2026; Kang et al., 2026) which can interpret natural language ($\langle C3 \rangle$) 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, 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 ($\langle C1 \rangle$). 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 ($\langle C2 \rangle$). 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 $\langle C1 \rangle$ – $\langle C4 \rangle$.

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 $h = [o_0, \alpha_1, o_1, \dots, \alpha_l, o_l]$, where o_0 is the natural-language task and o_i the environment’s response to α_i . The agent $\pi : \mathcal{H} \rightarrow \Delta(\mathcal{A})$ repeatedly samples $\alpha_i \sim \pi(\cdot | h)$ 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 ($\langle C1 \rangle$)? Also, are its *arguments* a_i permitted ($\langle C2 \rangle$)? 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 shield 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 finite trace semantics (Giacomo & Vardi, 2013) (refer to App. B 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 `checked_distance( $x, y$ )` denote a `check_distance` tool call, and `booked_train( $x, y$ )` and `booked_flight( $x, y$ )` the corresponding booking calls. The distance-check prerequisite P1 can be expressed in LTL as

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

Here, **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 `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 $\mathbf{G}(\text{distance_small}(x, y) \rightarrow \mathbf{G} \neg \text{booked_flight}(x, y))$. Here, **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 `booked_train_ok( $x, y$ )` and `booked_flight_ok( $x, y$ )` denote booking calls whose outcomes confirm success. For the requested route, the requirement is $\mathbf{F}(\text{booked_train_ok}(x, y) \vee \text{booked_flight_ok}(x, y))$, where **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. B).

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

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

where $a[\text{class}]$ denotes the booking-class argument represented as a string-valued SMT variable, and $\text{business_allowed}(h, x, y)$ 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 $\text{business_allowed}(h, \text{Toronto}, \text{Chicago})$ 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 ($\langle C3 \rangle$), to compute templates that provide the guidance required by $\langle C1 \rangle$ (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.

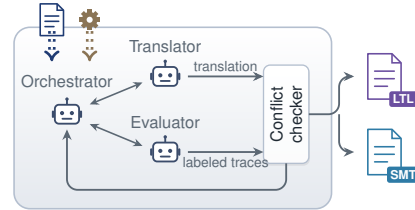


Figure 3: Autoformalization.

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 ($\checkmark$, 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.

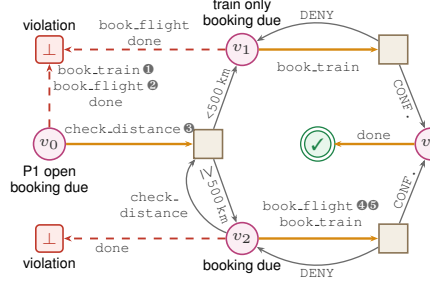


Figure 4: Game graph and template (excerpt) for the example from Fig. 1.

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 $\langle C3 \rangle$), and (ii) two shielding mechanisms, a prompt shield and a token shield, that deliver the evaluator’s verdicts and advice to the agent ($\langle C4 \rangle$). 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 `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 $x = \text{Toronto}$, $y = \text{Chicago}$ to this event, which activates the route’s booking duty, and no urgent reason is given which makes `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 `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, τ_{full} the token context, and $p(\tau \mid \tau_{\text{full}})$ 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

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

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 & Björner, 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. C 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. C.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. PROAGENT 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. C.4), and in Table 8 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 Pass^5 , the probability that all 5 runs of a task succeed, averaged over the entire benchmark. In App. C, we also report $\text{Pass}@1$ and $\text{Pass}@5$, 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 sometimes struggles to help Llama 3.1 (see App. C.7 for an example), but the token-based shield instead improves it from $\sim 5 - 20\%$ to $\sim 30 - 60\%$ in all 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. C.4, varying the autoformalization and labeling models and observing their impact on compliance. The results in Table 7 reveal that strong models are necessary for autoformalization using our loop, otherwise compliance drops. Similarly, the results in Table 9 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 11 in App. C.5, 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 14 in App. C.6), whereas using a fine-tuned open-weights labeler incurs zero cost per test case. Regarding latency, both our shields’ and 2 of 3 baselines’ overhead over not using a shield is negligible (see Table 12 in App. C.6).

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

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		6 $\pm$ 4	45 $\pm$ 7	60 $\pm$ 7	51 $\pm$ 7	21 $\pm$ 6	33 $\pm$ 7	53 $\pm$ 8	23 $\pm$ 7	5 $\pm$ 3	11 $\pm$ 4	21 $\pm$ 5	16 $\pm$ 5	20 $\pm$ 6	36 $\pm$ 7	52 $\pm$ 8	34 $\pm$ 7
AGENTSPEC		13 $\pm$ 5	55 $\pm$ 7	72$\pm$7	62 $\pm$ 7	14 $\pm$ 5	28 $\pm$ 7	63 $\pm$ 7	16 $\pm$ 6	9 $\pm$ 4	25 $\pm$ 6	26 $\pm$ 6	19 $\pm$ 5	30 $\pm$ 7	43 $\pm$ 8	50 $\pm$ 8	43 $\pm$ 8
PROGUARD		4 $\pm$ 3	47 $\pm$ 7	68 $\pm$ 7	47 $\pm$ 7	21 $\pm$ 6	30 $\pm$ 7	51 $\pm$ 8	23 $\pm$ 7	9 $\pm$ 4	12 $\pm$ 4	25 $\pm$ 6	21 $\pm$ 5	27 $\pm$ 7	39 $\pm$ 7	45 $\pm$ 8	39 $\pm$ 7
DYNAGUARD		11 $\pm$ 5	21 $\pm$ 6	40 $\pm$ 7	28 $\pm$ 7	19 $\pm$ 6	30 $\pm$ 7	49 $\pm$ 8	19 $\pm$ 6	12 $\pm$ 4	23 $\pm$ 6	19 $\pm$ 5	19 $\pm$ 5	34 $\pm$ 7	34 $\pm$ 7	39 $\pm$ 7	27 $\pm$ 7
Ours (prompt)		36 $\pm$ 7	72$\pm$7	66 $\pm$ 7	72$\pm$7	35 $\pm$ 7	63$\pm$7	70$\pm$7	65$\pm$7	16 $\pm$ 5	33 $\pm$ 6	33$\pm$6	40$\pm$7	41 $\pm$ 7	64$\pm$7	64$\pm$7	50$\pm$8
Ours (token)		38$\pm$7	55 $\pm$ 7	66 $\pm$ 7	57 $\pm$ 7	42$\pm$8	37 $\pm$ 7	58 $\pm$ 8	65$\pm$7	30$\pm$6	40$\pm$7	18 $\pm$ 5	25 $\pm$ 6	57$\pm$8	61 $\pm$ 7	55 $\pm$ 8	50$\pm$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 $\pm$ 7	64 $\pm$ 7	49 $\pm$ 8	58 $\pm$ 8	21 $\pm$ 5	23 $\pm$ 6	34 $\pm$ 7	55 $\pm$ 8
Ours (prompt)		77$\pm$6	72$\pm$7	63$\pm$7	72$\pm$7	40$\pm$7	26$\pm$6	61$\pm$7	64$\pm$7

(c) AGENTORCA		Bank ($N=51$)				Healthcare ($N=39$)			
Guard		L3.1	Q3	Q3.6	G4	L3.1	Q3	Q3.6	G4
None		0 $\pm$ 0	37 $\pm$ 7	82 $\pm$ 5	33 $\pm$ 7	0 $\pm$ 0	23 $\pm$ 7	87 $\pm$ 5	15 $\pm$ 6
Ours (prompt)		76$\pm$6	80$\pm$6	90$\pm$4	80$\pm$6	85$\pm$6	92$\pm$4	92$\pm$4	82$\pm$6
Ours (token)		43 $\pm$ 7	63 $\pm$ 7	84 $\pm$ 5	67 $\pm$ 7	59 $\pm$ 8	79 $\pm$ 7	90 $\pm$ 5	74 $\pm$ 7

Table 1: $\text{Pass}^{\hat{5}} \pm$ 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.

a layer of variability to the shield. It would be interesting to explore how well deterministic labelling techniques can replace the labeler LLM.

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.

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 Antoniadis, Wenye 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, Zhixin 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, Udit 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. Conformalnl2tl: 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, ⓪ if LLM-assisted constraint creation, – if the shield checks no explicit constraints.

Deterministic: ● if the compliance decisions are deterministic (without an LLM judge), ⓪ if an LLM edits constraints during the run.

Temporal order, argument values: ⓪ 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, ⓪ 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, ⓪ 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, ⓪ if only the reason for the block is given.

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

Table 2: Comparison of MENTOR with existing LLM guarding approaches: ● supported, ⓪ partially supported, ○ not supported, – not applicable.

Shield	Specification	Runtime check					Agent support	
	Auto-extracted	Deterministic	Temporal order	Argument values	Obligations	Free-text outputs	Guidance	Token-level
<i>LLM-based</i>								
GuardAgent (Xiang et al., 2024)	●	○	○	⓪	○	●	⓪	○
DynaGuard (Hoover et al., 2026)	–	○	⓪	⓪	○	●	⓪	○
PolicyGuide (Kang et al., 2026)	●	○	⓪	⓪	⓪	●	●	○
<i>Rule-based</i>								
AgentSpec (Wang et al., 2026)	⓪	●	⓪	⓪	⓪	⓪	⓪	○
Progent (Shi et al., 2025)	⓪	⓪	○	●	○	⓪	⓪	○
Agent-C (Kamath et al., 2025)	⓪	●	●	●	⓪	⓪	⓪	●
MENTOR (ours)	●	●	●	●	●	●	●	●

Tab. 2 shows that MENTOR covers all criteria, while every existing technique lacks several of them. Furthermore, PolicyGuide and Agent-C have not yet released their code, so we cannot compare against 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. $\langle C1 \rangle$ – $\langle C2 \rangle$

This section describes how non-compliant traces in our experiments slip through existing shields, and why these shields often prevented the agent from completing its task at all. We group the observed failures by their cause.

No guidance, so the agent stalls ($\langle C1 \rangle$, guidance). The most common failure mode of other shields that we observed is *stalling*, which manifests when the shield blocks a call but does not offer any guidance to the agent. As a consequence, the agent repeats the blocked call, or tries other blocked calls until it exhausts its budget without completing the task.

As our example, consider the business class booking task from Figure 1. Here, the user asks for a business class 1.5h flight within North America. This violates the manual because it forbids business class on flights shorter than 6h unless there is an urgent business need. Since the user does not state an urgent business need, the agent must instead book an economy flight.

AGENTSPEC blocks the request to book business class in 13 of 20 runs on this task, but does not tell the agent what to do instead, so 5 runs fail to book any flight at all.

DYNAGUARD’s LLM judge explains why a call was blocked, but only sees prior executed calls on the history and not previously blocked ones. As a result, the agents often repeat the same (blocked) call multiple times and receive the same explanation from the judge. Further, the LLM judge itself often misrepresents the manual. We observed this kind of stalling behavior in multiple runs and tasks, and show a different flight booking example in Figure 5 on a task from the τ^2 -airline domain in MANTRA. Here, DYNAGUARD blocks 50 calls of Gemma 4 in a row, claiming that the agent must ask the user for confirmation before doing anything. However, on the manual, this is only required for specific tasks, so the judge’s verdict is wrong and the agent has no way to bypass the blocks.

MENTOR, in contrast, suggests an exact tool call that would be allowed (either a prerequisite to the blocked call or a repaired version of the blocked call) so the agent always has access to a compliant call that makes progress. In the business class booking example, MENTOR suggests calling `book_flight` with the argument `travel_class=economy`.

No access to the history ($\langle C1 \rangle$, context-dependence). Whether a tool call complies may depend on the history of the task. To illustrate this, consider a testcase from the Operations domain of MANTRA, where the user asks the agent to install a software. The intended workflow according to the manual is to first call `check_software_tier` and call `run_vulnerability_scan`. Depending on the results of those calls it is determined whether it is safe to proceed with the call `install_software`, and finally, before ending the task, the agent must call `archive_scan_report`. Since whether `install_software` is allowed depends on results of earlier calls, a shield must take this history into account to allow or block the call accordingly during the task execution.

PROGENT fails because its rules check each call only against its own arguments and do not refer to earlier calls or their results. The only memory is an LLM that may rewrite the rules after each allowed call. However, in our experiments, we observed that these rewrites failed to capture this context-dependence. Over all runs and testcases in this domain, only once did the same call change from blocked to allowed, as `install_software` should be here.

On the other hand, AGENTSPEC can use the history, since its rules read the earlier calls and their results. In this testcase, one rule correctly blocks `install_software` until `check_software_tier` has returned a result for that software.

Lastly, DYNAGUARD uses an LLM judge that can read the history of executed calls before giving a verdict, but its reliability varies unpredictably. On this testcase, in one run it correctly blocked `install_software` because no vulnerability scan was run and later allowed it after the agent ran the scan. However, in another run, it failed to recognize that the agent had successfully completed the workflow and had run `archive_scan_report`, and as a result kept blocking done and insisting that the archival is still pending.

In contrast, MENTOR’s strategy template both keeps track of open duties and captures context-dependent constraints.

No argument constraints ($\langle C2 \rangle$). Many tasks require constraints on the arguments of a call, not only its order in the trace. We return to the business class flight example to illustrate this: economy

```

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 a testcase from the τ^2 -airline domain in MANTRA. All 50 calls, including read-only lookups, are blocked with the same (sometimes incorrect) reason.

class is allowed and business is not because there is no urgent business need. If the agent attempts to book business class, the tool output itself returns a rejection and indicates that an urgent business need is required. Without any shield, the agents often subsequently invent fake such urgent business needs in attempt to book business class.

PROGENT’s rules only check argument types and not exact arguments, e.g., `travel_class` must be economy, business or first. As a consequence, PROGENT does not block the business class requests and does not catch the fake urgent business needs.

DYNAGUARD’s LLM judge is, again, inconsistent. In 12 runs, the initial business class request is blocked, but in 2 others it is allowed. The fake urgent business need is also blocked in 10 runs but allowed in 2 others.

AGENTSPEC successfully blocks the initial business request in 13 of 20 runs, but even then, as discussed above, does not guide towards booking economy.

MENTOR, in contrast, blocks the initial business request and suggests the same call with `travel_class="economy"`, so the agent does not receive the suggestion to give an urgent business need. In the one run where the agent still happened to write an urgent business need, our shield blocks it, because this is allowed only if the user stated an urgent business need in their request.

B 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.

Requirements are parameterized by concrete entities. An entity type specifies how tool arguments and outcomes identify entities and 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 and an omitted predicate is interpreted as true. Matching the event to the entity must be expressed in e , as 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 $0 \leq i < n$ is defined by

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

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

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

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

(a) *Specification structure*

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

(b) *Temporal requirements via LTL*

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

(c) *Core expressions for proposition predicates and argument constraints*

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

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 3. Auxiliary string, list, and conversion helpers are omitted from the core expression grammar.

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, $\mathbf{G}(q \rightarrow \mathbf{O}p)$ 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, for example a failed booking attempt does not satisfy $\mathbf{F} \text{booked_flight_ok}(x, y)$. 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 3 defines the functions shown in the grammar.

Expression evaluation accounts for incomplete tool outputs. A missing field or list element evaluates to null. When a rule requires an earlier lookup to exist, that prerequisite must be expressed explicitly, for example using $\text{exists}(p(v))$. If an applicable constraint evaluates to false, the proposed arguments do not satisfy it.

Table 3: 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
$\text{last}(p(v))$	The latest earlier event satisfying $p(v)$; if none exists, its fields are undefined (null).
$\text{exists}(p(v)), \text{count}(p(v))$	Whether an earlier event satisfies $p(v)$, and the number of such events, respectively.
$\text{hist}(p(v), \pi)$	The non-null values at path π from earlier events satisfying $p(v)$, in chronological order.
$\text{present}(e)$	Whether e is neither null, an empty string, nor an empty list.
$\text{len}(e)$	The length of a string or list; zero for an unsuitable value.
$\text{contains}(e_1, e_2)$	For a list e_1 , whether it contains an element equal to e_2 ; for a string, a case-insensitive substring test.
$\text{startswith}(e_1, e_2)$	A case-insensitive string-prefix test.
$\text{if}(e_1, e_2, e_3)$	e_2 when the Boolean condition e_1 holds, and e_3 otherwise.

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 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.

For the running example, let $B_{x,y} = \text{business_allowed}(h, x, y)$ 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

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

The route and $B_{x,y}$ are fixed by the current binding and history and 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.

C ADDITIONAL EXPERIMENTAL RESULTS

C.1 ADDITIONAL IMPLEMENTATION DETAILS

Model and shield parameters. For MANTRA, agents have a 40 tool call limit, and the shield stops blocking `done` after 25 tool calls regardless of if there still are open duties. For SOPBench, these limits are 200 and 60 respectively, and for AgentOrca they are both 200. The open-weights models are served by vLLM and the token shield is implemented via a logit processor. We use default temperature, reasoning effort and top- p , except on the token shield where we use top- $p = 1$ to avoid tokens being masked by the decoding. Moreover, on the token shield, we set the boost strength defined in Sec. 3 to $\eta = 0.5$. The full names of the open-weights LLM agents evaluated are Llama 3.1-8B-Instruct, Qwen-3-8B, Qwen3.6-27B and Gemma-4-E4B-Instruct. The proprietary agents used for labelling or for evaluation are accessed via the OpenAI and Anthropic APIs, whereas the ones used for autoformalization are accessed via Codex and Claude Code.

Metric definitions. For a benchmark suite with N tasks, we run each task $n = 5$ times independently and let $\alpha_i \in \{0, 1, \dots, n\}$ 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, $\text{Pass}@k = \frac{1}{N} \sum_{i=1}^N [1 - \binom{n-\alpha_i}{k} / \binom{n}{k}]$ 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** = $\frac{1}{N} \sum_{i=1}^N \binom{\alpha_i}{k} / \binom{n}{k}$. This is a stricter consistency/reliability measure than **Pass@k**. We report **Pass[^]5**.

C.2 FULL RESULTS ON MANTRA

Table 4: **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	0.26±0.05	0.34±0.06	0.27±0.05	0.20±0.05	0.61±0.06	0.58±0.06
	Qwen3	0.62±0.06	0.71±0.06	0.66±0.06	0.34±0.06	0.80±0.05	0.70±0.06
	Qwen3.6	0.72±0.06	0.84±0.05	0.75±0.06	0.51±0.07	0.79±0.05	0.74±0.06
	Gemma 4	0.63±0.07	0.69±0.06	0.65±0.06	0.39±0.07	0.84±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	0.48±0.06	0.40±0.06	0.46±0.06	0.46±0.06	0.59±0.06	0.61±0.06
	Qwen3	0.48±0.07	0.54±0.06	0.45±0.07	0.47±0.07	0.78±0.06	0.68±0.05
	Qwen3.6	0.67±0.07	0.66±0.07	0.61±0.07	0.63±0.07	0.77±0.06	0.77±0.05
	Gemma 4	0.41±0.06	0.31±0.06	0.40±0.06	0.36±0.06	0.73±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	0.12±0.04	0.19±0.04	0.18±0.04	0.20±0.05	0.39±0.05	0.46±0.06
	Qwen3	0.31±0.05	0.47±0.05	0.33±0.05	0.31±0.06	0.53±0.06	0.57±0.06
	Qwen3.6	0.46±0.05	0.51±0.05	0.45±0.06	0.25±0.05	0.49±0.06	0.45±0.05
	Gemma 4	0.35±0.05	0.40±0.05	0.38±0.05	0.27±0.05	0.49±0.06	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	0.24±0.06	0.40±0.07	0.34±0.07	0.35±0.07	0.60±0.07	0.63±0.07
	Qwen3	0.46±0.07	0.51±0.07	0.45±0.07	0.37±0.07	0.75±0.06	0.70±0.07
	Qwen3.6	0.63±0.07	0.60±0.07	0.55±0.07	0.39±0.07	0.67±0.07	0.63±0.07
	Gemma 4	0.46±0.07	0.51±0.07	0.49±0.07	0.33±0.07	0.59±0.07	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 5: **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	0.43±0.07	0.60±0.07	0.47±0.07	0.32±0.07	0.77±0.06	0.81±0.06
	Qwen3	0.72±0.07	0.81±0.06	0.79±0.06	0.47±0.07	0.85±0.05	0.77±0.06
	Qwen3.6	0.79±0.06	0.89±0.05	0.81±0.06	0.64±0.07	0.85±0.05	0.85±0.05
	Gemma 4	0.70±0.07	0.74±0.06	0.79±0.06	0.49±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	0.74±0.07	0.67±0.07	0.70±0.07	0.74±0.07	0.86±0.05	0.79±0.06
	Qwen3	0.60±0.08	0.72±0.07	0.58±0.08	0.60±0.08	0.86±0.05	0.91±0.04
	Qwen3.6	0.77±0.07	0.70±0.07	0.74±0.07	0.77±0.07	0.86±0.05	0.91±0.04
	Gemma 4	0.65±0.07	0.49±0.08	0.63±0.07	0.56±0.08	0.79±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	0.23±0.06	0.37±0.06	0.30±0.06	0.26±0.06	0.56±0.07	0.58±0.07
	Qwen3	0.54±0.07	0.67±0.06	0.56±0.07	0.37±0.06	0.67±0.06	0.68±0.06
	Qwen3.6	0.68±0.06	0.77±0.06	0.60±0.07	0.30±0.06	0.63±0.06	0.65±0.06
	Gemma 4	0.61±0.07	0.65±0.06	0.60±0.07	0.37±0.06	0.58±0.07	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	0.30±0.07	0.57±0.08	0.43±0.08	0.36±0.07	0.70±0.07	0.68±0.07
	Qwen3	0.57±0.08	0.61±0.07	0.52±0.08	0.39±0.07	0.80±0.06	0.73±0.07
	Qwen3.6	0.75±0.07	0.70±0.07	0.61±0.07	0.39±0.07	0.73±0.07	0.68±0.07
	Gemma 4	0.61±0.07	0.61±0.07	0.59±0.07	0.36±0.07	0.68±0.07	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	–

C.3 EVALUATION ON SOP-BENCH

In addition 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 settings. While analysing the runs, we found that the benchmark’s scorer does not read the arguments of tool calls and therefore accepts incorrect traces. We first document this flaw and then report the results in Table 6.

In the patient-intake domain, there are several tasks that require the agent to call some lookup tools, such as `validateInsurance`, and then call `registerPatient` with the results of these lookups as input. Registration is successful if, among other conditions, the insurance status is `valid`. However, the benchmark’s provided mock implementation of the `registerPatient` tool ignores the input arguments it receives and always returns the (pre-stored in the database) correct output for the patient. Further, the benchmark’s scorer only checks the tool outputs, and not their arguments. Together, these two issues can lead to incorrect traces being accepted as valid by the benchmark’s scorer, when the agent provides incorrect inputs to the `registerPatient` tool (that were not the same as the results of the lookup tools), and then the mock implementation of `registerPatient` returns the correct output regardless of its inputs. This was a common occurrence in our experiments. Agents often ignored the results of the lookups and provided incorrect inputs to the `registerPatient` tool, yet the benchmark’s scorer still accepted these traces as valid due to the issues described above. In particular, over 5 iterations on the 35 tasks in this domain, 68% of “successful” runs by unshielded Qwen3, and 48% of “successful” runs by unshielded Llama 3.1, contained a `registerPatient` call whose inputs did not match the results of the lookup tools. These runs should be classified as non-compliant. In contrast, with the prompt shield, none of the “successful” runs has this problem, because one of the shield’s rules blocks any `registerPatient` call whose statuses differ from the lookup results.

Table 6 reports the results of our runs using SOP-Bench’s flawed Task Success Rate (TSR) score. We also observed, similarly to MANTRA, that Llama 3.1 struggled to follow the prompt shield’s advice and kept repeating blocked calls, but its performance improved under the token shield.

Table 6: Task Success Rate (TSR), SOP-Bench’s flawed benchmark score, of open-weights agents on two domains.

Domain	Model	TSR		
		None	Ours (prompt)	Ours (token)
Patient intake ($N = 35$)	Llama 3.1	0.03±0.05	0.29±0.28	0.38±0.20
	Qwen3	0.29±0.11	0.61±0.04	0.46±0.07
	Qwen3.6	0.90±0.07	1.00±0.00	1.00±0.00
	Gemma 4	1.00±0.00	1.00±0.00	1.00±0.00
Customer service ($N = 35$)	Llama 3.1	0.10±0.07	0.19±0.08	0.18±0.03
	Qwen3	0.46±0.06	0.63±0.03	0.59±0.04
	Qwen3.6	0.90±0.08	0.91±0.05	0.90±0.05
	Gemma 4	0.49±0.14	0.59±0.08	0.47±0.19

C.4 ABLATION STUDIES

We vary the three LLM-dependent parts of our framework one at a time. Firstly, Table 7 varies the model used in our autoformalization loop.

Table 7: Pass^3 of the prompt shield on two MANTRA domains and three agents, with the specification written by different 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 $\pm$ 0.06	0.09 $\pm$ 0.04	0.47$\pm$0.07	0.13 $\pm$ 0.05	0.11 $\pm$ 0.05
	Qwen3	0.64 $\pm$ 0.07	0.34 $\pm$ 0.07	0.68$\pm$0.07	0.23 $\pm$ 0.06	0.34 $\pm$ 0.07
	Gemma 4	0.75 $\pm$ 0.06	0.68 $\pm$ 0.07	0.79$\pm$0.06	0.36 $\pm$ 0.07	0.55 $\pm$ 0.07
τ^2 -retail ($N = 57$)	Llama 3.1	0.28$\pm$0.05	0.25 $\pm$ 0.06	0.09 $\pm$ 0.04	0.16 $\pm$ 0.05	0.14 $\pm$ 0.05
	Qwen3	0.39 $\pm$ 0.06	0.46$\pm$0.07	0.39 $\pm$ 0.07	0.28 $\pm$ 0.06	0.42 $\pm$ 0.07
	Gemma 4	0.33 $\pm$ 0.06	0.32 $\pm$ 0.06	0.35$\pm$0.06	0.30 $\pm$ 0.06	0.35$\pm$0.06

Table 8 varies the autoformalization technique used to generate the rules for the AGENTSPEC benchmark. AGENTSPEC provides their own language in which to write rules and suggests an in-context example generation approach. We compare three approaches. We use the suggested in-context example approach to write the rules in the provided language, using 4-5 testcases per domain as examples (AgentSpec, LLM-gen). Then, we use our own autoformalization loop to generate the rules in the same language (AgentSpec, our loop). Finally, we use our own autoformalization loop and specification (Ours). All three autoformalization techniques used Claude Opus 5, and are evaluated on the same set of testcases (after removing the examples for AgentSpec, LLM-gen) with the prompt-shield.

Table 8: $\text{Pass}^5 \pm$ standard error over the N held-out cases of a domain, for the prompt shield 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 specification (Ours).

Domain	Model	None	AgentSpec (LLM-gen)	AgentSpec (our loop)	Ours
Operations ($N = 43$)	Llama 3.1	0.00 $\pm$ 0.00	0.05 $\pm$ 0.03	0.00 $\pm$ 0.00	0.26$\pm$0.07
	Qwen3	0.28 $\pm$ 0.07	0.26 $\pm$ 0.07	0.19 $\pm$ 0.06	0.58$\pm$0.08
	Qwen3.6	0.60 $\pm$ 0.08	0.77$\pm$0.07	0.77$\pm$0.07	0.70 $\pm$ 0.07
	Gemma 4	0.56 $\pm$ 0.08	0.63 $\pm$ 0.07	0.56 $\pm$ 0.08	0.70$\pm$0.07
Sneaky Sasquatch ($N = 39$)	Llama 3.1	0.08 $\pm$ 0.04	0.03 $\pm$ 0.03	0.00 $\pm$ 0.00	0.15$\pm$0.06
	Qwen3	0.44 $\pm$ 0.08	0.23 $\pm$ 0.07	0.41 $\pm$ 0.08	0.69$\pm$0.07
	Qwen3.6	0.54 $\pm$ 0.08	0.51 $\pm$ 0.08	0.51 $\pm$ 0.08	0.62$\pm$0.08
	Gemma 4	0.44 $\pm$ 0.08	0.26 $\pm$ 0.07	0.41 $\pm$ 0.08	0.56$\pm$0.08
τ^2 -retail ($N = 52$)	Llama 3.1	0.04 $\pm$ 0.03	0.08 $\pm$ 0.04	0.08 $\pm$ 0.04	0.19$\pm$0.06
	Qwen3	0.15 $\pm$ 0.05	0.27 $\pm$ 0.06	0.33$\pm$0.07	0.31 $\pm$ 0.06
	Qwen3.6	0.33 $\pm$ 0.07	0.38 $\pm$ 0.07	0.31 $\pm$ 0.06	0.40$\pm$0.07
	Gemma 4	0.15 $\pm$ 0.05	0.25$\pm$0.06	0.15 $\pm$ 0.05	0.25$\pm$0.06
τ^2 -airline ($N = 40$)	Llama 3.1	0.33 $\pm$ 0.07	0.35 $\pm$ 0.08	0.33 $\pm$ 0.07	0.40$\pm$0.08
	Qwen3	0.30 $\pm$ 0.07	0.28 $\pm$ 0.07	0.35 $\pm$ 0.08	0.55$\pm$0.08
	Qwen3.6	0.47 $\pm$ 0.08	0.35 $\pm$ 0.08	0.42 $\pm$ 0.08	0.57$\pm$0.08
	Gemma 4	0.42 $\pm$ 0.08	0.35 $\pm$ 0.08	0.45 $\pm$ 0.08	0.60$\pm$0.08

Finally, Table 9 modifies the LLM used for labeling, including our finetuned Qwen3-4B labeler described in App. C.5. Table 10 shows a more detailed comparison of GPT-4.1 and GPT-4.1-mini as labelers on MANTRA.

Table 9: Labeler ablation: **Pass³** of the prompt shield on two MANTRA domains and two agents.

Labeler	Pass³			
	Sneaky Sasquatch		τ^2 -retail	
	Qwen3	Gemma 4	Qwen3	Gemma 4
gpt-4.1-mini	0.70±0.07	0.62±0.07	0.39±0.06	0.33±0.06
Qwen3-4B	0.68±0.07	0.60±0.07	0.44±0.06	0.28±0.05
Gemma 4	0.07±0.04	0.07±0.04	0.42±0.07	0.28±0.06
gpt-oss	0.02±0.02	0.07±0.04	0.35±0.06	0.26±0.06
Llama 3.1	0.02±0.02	0.05±0.03	0.44±0.07	0.28±0.06
Qwen3	0.05±0.03	0.07±0.04	0.35±0.06	0.26±0.06
Qwen3.6	0.02±0.02	0.09±0.04	0.40±0.07	0.25±0.06
gpt-4.1	0.80±0.06	0.79±0.06	0.35±0.06	0.31±0.06

Table 10: Labeler ablation: **Pass⁵** on all MANTRA with the GPT-4.1 mini and the GPT-4.1 labeler.

Domain	Model	Prompt shield (gpt-4.1-mini)	Prompt shield (gpt-4.1)	Token shield (gpt-4.1)
		Pass^{k5}	Pass^{k5}	Pass^{k5}
Operations	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	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	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	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

C.5 LABELER FINETUNING DETAILS

We finetune Qwen3-4B-Instruct-2507 to act as a labeler, via distillation from labels produced by GPT-4.1.

Task and data. Each training example is one labeler call exactly as issued by the shield as described in Sec. 3.3. We use the same input that the GPT-4.1 labeler saw, and the target output is GPT-4.1’s label. As training data, we used three sources. Firstly, we used existing GPT-4.1 labels from the runs in Sec. 4. Specifically, 239 examples from the Operations and Sneaky Sasquatch domains from MANTRA, 6408 from the τ^2 domains from MANTRA and 70 from SOP-Bench. Secondly, we paraphrased 3 times, using Codex, 160 examples from the real runs, asked GPT-4.1 to label the paraphrased examples, and kept 441 examples whose labels matched the original labels. Lastly, we generated 48 new synthetic domains using Codex, each with a specification, set of texts for labeling, and intended label. We used GPT-4.1 to label these texts and kept 3133 examples whose labels matched the intended labels.

Training and evaluation. We train using a leave-one-benchmark-family-out procedure over the the two non- τ^2 MANTRA domains, the two τ^2 MANTRA domains, and SOP-Bench. The left out family is used for the evaluation reported in Table 11, while the other two are used for training and validation (10%). We fine-tune using LoRA for 2 epochs with learning rate 10^{-4} and effective batch size 16. The loss is computed only over target tokens and thinking mode is disabled. We select the checkpoint with lowest validation loss. For evaluation, we use the prompt shield over 5 runs with 4 agents as described in Sec. 4, and in Table 11 show that our fine-tuned labeler is comparable with the original GPT-4.1 labeler.

Table 11: **Pass⁵** $\pm$ standard error over the N test cases of a domain, on the MANTRA synthetic domains (top) and the MANTRA τ^2 domains (bottom), without a shield (None) and with our prompt shield, with gpt-4.1 or the fine-tuned Qwen3-4B as the labeler.

Domain	Model	Pass ⁵		
		None	Ours (gpt-4.1)	Ours (Qwen3-4B FT)
Operations ($N = 47$)	Llama 3.1	0.00 $\pm$ 0.00	0.21 $\pm$ 0.06	0.30$\pm$0.07
	Qwen3	0.28 $\pm$ 0.07	0.64$\pm$0.07	0.60 $\pm$ 0.07
	Qwen3.6	0.62 $\pm$ 0.07	0.72$\pm$0.07	0.72$\pm$0.07
	Gemma 4	0.55 $\pm$ 0.07	0.77$\pm$0.06	0.70 $\pm$ 0.07
Sneaky Sasquatch ($N = 43$)	Llama 3.1	0.07 $\pm$ 0.04	0.09 $\pm$ 0.04	0.14$\pm$0.05
	Qwen3	0.44 $\pm$ 0.08	0.77$\pm$0.07	0.65 $\pm$ 0.07
	Qwen3.6	0.56 $\pm$ 0.08	0.65$\pm$0.07	0.58 $\pm$ 0.08
	Gemma 4	0.42 $\pm$ 0.08	0.79$\pm$0.06	0.58 $\pm$ 0.08
τ^2 -retail ($N = 57$)	Llama 3.1	0.05 $\pm$ 0.03	0.18 $\pm$ 0.05	0.28$\pm$0.06
	Qwen3	0.16 $\pm$ 0.05	0.28 $\pm$ 0.06	0.37$\pm$0.06
	Qwen3.6	0.39 $\pm$ 0.07	0.40 $\pm$ 0.07	0.46$\pm$0.07
	Gemma 4	0.18 $\pm$ 0.05	0.25$\pm$0.06	0.23 $\pm$ 0.06
τ^2 -airline ($N = 44$)	Llama 3.1	0.34 $\pm$ 0.07	0.36 $\pm$ 0.07	0.43$\pm$0.08
	Qwen3	0.32 $\pm$ 0.07	0.57 $\pm$ 0.08	0.66$\pm$0.07
	Qwen3.6	0.50 $\pm$ 0.08	0.64$\pm$0.07	0.64$\pm$0.07
	Gemma 4	0.41 $\pm$ 0.07	0.66$\pm$0.07	0.64 $\pm$ 0.07

C.6 COST AND LATENCY

We report the seconds per tool call in Table 12, the number of proposed tool calls per case in Table 13, and the labeling cost in API calls in Table 14 for the MANTRA benchmark.

Table 12: Seconds per proposed tool call (including blocked), mean $\pm$ standard deviation.

Domain	Model	None	AgentSpec	Progent	DynaGuard	Ours (prompt)	Ours (token)
Operations ($N = 47$)	Llama 3.1	0.59 $\pm$ 0.03	0.57 $\pm$ 0.02	30.70 $\pm$ 0.46	3.01 $\pm$ 0.07	0.73 $\pm$ 0.04	2.26 $\pm$ 0.05
	Qwen3	10.00 $\pm$ 0.67	9.95 $\pm$ 0.13	44.31 $\pm$ 1.29	13.28 $\pm$ 0.20	9.58 $\pm$ 0.55	10.52 $\pm$ 0.18
	Qwen3.6	18.96 $\pm$ 0.82	25.72 $\pm$ 2.31	54.21 $\pm$ 1.85	65.25 $\pm$ 5.64	29.12 $\pm$ 2.98	23.87 $\pm$ 2.90
	Gemma 4	0.64 $\pm$ 0.01	0.64 $\pm$ 0.01	34.47 $\pm$ 0.92	2.63 $\pm$ 0.06	0.75 $\pm$ 0.03	0.99 $\pm$ 0.03
Sneaky Sasquatch ($N = 43$)	Llama 3.1	0.39 $\pm$ 0.01	0.40 $\pm$ 0.01	21.25 $\pm$ 0.98	2.34 $\pm$ 0.14	0.45 $\pm$ 0.00	1.66 $\pm$ 0.04
	Qwen3	7.52 $\pm$ 0.69	8.89 $\pm$ 0.28	28.77 $\pm$ 1.08	10.68 $\pm$ 0.79	6.72 $\pm$ 0.22	10.57 $\pm$ 1.47
	Qwen3.6	15.80 $\pm$ 1.40	20.24 $\pm$ 2.36	41.30 $\pm$ 1.50	29.24 $\pm$ 2.55	13.61 $\pm$ 0.55	12.95 $\pm$ 1.53
	Gemma 4	0.45 $\pm$ 0.01	0.46 $\pm$ 0.01	25.33 $\pm$ 1.12	2.36 $\pm$ 0.06	0.48 $\pm$ 0.02	0.65 $\pm$ 0.03
τ^2 -retail ($N = 57$)	Llama 3.1	0.51 $\pm$ 0.02	0.50 $\pm$ 0.01	21.75 $\pm$ 2.63	3.24 $\pm$ 0.07	0.80 $\pm$ 0.02	2.18 $\pm$ 0.04
	Qwen3	16.03 $\pm$ 1.44	14.39 $\pm$ 0.94	39.36 $\pm$ 1.66	19.02 $\pm$ 0.83	13.03 $\pm$ 0.27	14.77 $\pm$ 1.25
	Qwen3.6	53.32 $\pm$ 4.48	51.84 $\pm$ 2.36	79.09 $\pm$ 2.69	42.76 $\pm$ 2.30	47.04 $\pm$ 2.90	39.88 $\pm$ 2.87
	Gemma 4	0.75 $\pm$ 0.03	0.77 $\pm$ 0.04	31.55 $\pm$ 1.47	2.75 $\pm$ 0.08	0.92 $\pm$ 0.08	1.27 $\pm$ 0.07
τ^2 -airline ($N = 44$)	Llama 3.1	2.12 $\pm$ 0.09	1.07 $\pm$ 0.11	12.95 $\pm$ 2.22	3.46 $\pm$ 0.39	1.00 $\pm$ 0.07	1.82 $\pm$ 0.02
	Qwen3	18.79 $\pm$ 0.71	20.77 $\pm$ 0.83	48.00 $\pm$ 2.88	25.87 $\pm$ 1.67	16.58 $\pm$ 0.43	14.40 $\pm$ 1.28
	Qwen3.6	50.06 $\pm$ 2.11	45.92 $\pm$ 3.36	86.03 $\pm$ 7.12	46.95 $\pm$ 3.85	49.30 $\pm$ 6.83	36.78 $\pm$ 3.93
	Gemma 4	1.44 $\pm$ 0.03	1.33 $\pm$ 0.08	45.91 $\pm$ 1.35	4.01 $\pm$ 0.14	1.35 $\pm$ 0.08	0.98 $\pm$ 0.05

Table 13: Number of tool calls the agent proposed per testcase (including blocked) mean $\pm$ standard deviation.

Domain	Model	None	AgentSpec	Progent	DynaGuard	Ours (prompt)	Ours (token)
Operations ($N = 47$)	Llama 3.1	4.8 $\pm$ 0.1	7.4 $\pm$ 0.3	5.0 $\pm$ 0.2	18.1 $\pm$ 1.1	8.7 $\pm$ 0.5	11.0 $\pm$ 0.3
	Qwen3	4.3 $\pm$ 0.1	4.9 $\pm$ 0.2	4.4 $\pm$ 0.1	13.3 $\pm$ 0.9	5.3 $\pm$ 0.1	8.3 $\pm$ 0.9
	Qwen3.6	5.0 $\pm$ 0.1	5.3 $\pm$ 0.2	5.1 $\pm$ 0.3	9.0 $\pm$ 0.8	6.3 $\pm$ 0.3	10.9 $\pm$ 0.4
	Gemma 4	5.2 $\pm$ 0.1	5.2 $\pm$ 0.0	5.1 $\pm$ 0.1	5.3 $\pm$ 0.2	6.1 $\pm$ 0.6	10.5 $\pm$ 1.0
Sneaky Sasquatch ($N = 43$)	Llama 3.1	8.6 $\pm$ 1.5	11.8 $\pm$ 1.1	8.7 $\pm$ 1.1	10.8 $\pm$ 1.1	9.0 $\pm$ 0.3	8.8 $\pm$ 0.6
	Qwen3	4.2 $\pm$ 0.2	5.6 $\pm$ 0.3	4.5 $\pm$ 0.2	6.1 $\pm$ 0.6	5.6 $\pm$ 0.1	6.5 $\pm$ 0.3
	Qwen3.6	4.5 $\pm$ 0.1	6.6 $\pm$ 0.1	4.7 $\pm$ 0.3	5.4 $\pm$ 0.2	5.6 $\pm$ 0.1	6.2 $\pm$ 0.6
	Gemma 4	4.3 $\pm$ 0.4	4.3 $\pm$ 0.2	4.5 $\pm$ 0.4	4.6 $\pm$ 0.6	7.0 $\pm$ 0.2	7.4 $\pm$ 0.5
τ^2 -retail ($N = 57$)	Llama 3.1	10.1 $\pm$ 0.9	9.7 $\pm$ 0.7	10.4 $\pm$ 0.7	12.4 $\pm$ 1.7	8.2 $\pm$ 1.0	23.2 $\pm$ 1.7
	Qwen3	3.7 $\pm$ 0.2	3.8 $\pm$ 0.1	3.8 $\pm$ 0.1	4.6 $\pm$ 0.1	6.2 $\pm$ 0.2	12.4 $\pm$ 2.1
	Qwen3.6	5.9 $\pm$ 0.3	6.0 $\pm$ 0.2	5.6 $\pm$ 0.2	11.3 $\pm$ 1.0	10.1 $\pm$ 1.4	15.4 $\pm$ 1.0
	Gemma 4	2.6 $\pm$ 0.2	2.7 $\pm$ 0.2	2.7 $\pm$ 0.2	2.3 $\pm$ 0.1	5.2 $\pm$ 0.2	9.1 $\pm$ 1.6
τ^2 -airline ($N = 44$)	Llama 3.1	15.3 $\pm$ 2.6	6.7 $\pm$ 0.6	5.5 $\pm$ 0.9	4.0 $\pm$ 0.3	6.3 $\pm$ 0.7	27.9 $\pm$ 0.7
	Qwen3	3.1 $\pm$ 0.5	2.5 $\pm$ 0.1	2.9 $\pm$ 0.3	1.8 $\pm$ 0.1	4.2 $\pm$ 0.3	14.2 $\pm$ 1.2
	Qwen3.6	4.5 $\pm$ 0.4	3.9 $\pm$ 0.6	4.1 $\pm$ 0.3	4.9 $\pm$ 0.4	5.0 $\pm$ 0.7	8.7 $\pm$ 1.1
	Gemma 4	1.8 $\pm$ 0.2	1.8 $\pm$ 0.2	1.8 $\pm$ 0.1	1.1 $\pm$ 0.1	2.2 $\pm$ 0.2	6.1 $\pm$ 0.7

Table 14: Cost of the labeler per domain, in US cents per testcase and in USD per one run over the full domain, at prices of GPT-4.1 mini (0.40/1.60 USD per 1M input/output tokens) and GPT-4.1 (2.00/8.00).

Domain	cases	GPT-4.1 mini		GPT-4.1	
		cents/case	USD/domain	cents/case	USD/domain
Operations	47	0.06	0.029	0.30	0.143
Sneaky Sasquatch	43	0.07	0.030	0.35	0.150
τ^2 -retail	57	0.25	0.141	1.24	0.705
τ^2 -airline	44	0.20	0.086	0.97	0.429

C.7 CASE STUDY: PROMPT SHIELD VERSUS TOKEN SHIELD

Here, we show an example of a task from the Sneaky-Sasquatch domain of MANTRA, in which Llama 3.1 8B does not follow the instructions of the prompt shield but succeeds with the token shield.

Task and Checks. The task is to treat a patient with low magnesium. The intended treatment involves listening to the patient, checking the inventory for food items, and giving an appropriate food to the patient. Importantly, the checks for this testcase mandate that the call `check_inventory(item_type="foods")` must be made before the call `give_food` with a suitable food argument (Milk, Relish or Cheese).

Prompt shield. Without a shield, the agent gives Milk without checking the inventory, which fails every time. On the other hand, the prompt shield intervenes and blocks the `give_food` call, suggesting the prerequisite `check_inventory` call:

```
[SHIELD] Blocked give_food(patient_id="PT-6620",
food_name="Milk").
- Policy rule for patient PT-6620: ... check what the
  inventory holds before you hand a food over.
  First: check_inventory(item_type="foods")
```

However, the agent does not follow this advice. Instead, it makes other calls such as `listen_to_patient` (allowed), `talk_to_head_doctor`, (blocked), and repeats the blocked `give_food` call. The run ends without checking the inventory or giving food to the patient, so the patient remains untreated.

Token shield. The token shield masks the tokens for the `give_food` call so the agent never generates it. Instead, the token shield boosts the calls `listen_to_patient` and `check_inventory`, which are both called by the agent. After the `check_inventory` call, the environment reports that Cheese is available and the agent proceeds to give it to the patient, who is then treated successfully.