Temporal Guardrails for LLM Conversations: A Runtime Verification Framework*

Itay Cohen[1], Klaus Havelund[2], Moran Omer[1], and Doron Peled[1]

[1] Department of Computer Science Bar-Ilan University, Ramat Gan, Israel
[2] Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, USA

Abstract. Large Language Models (LLMs) are increasingly integrated into organizational workflows, raising growing concerns about their potential abuse for fraud, security breaches, or intellectual property leakage. While LLMs embed protective mechanisms and organizations develop their own *guardrails*, many practical guardrail approaches remain stateless and lack formal temporal semantics, and existing formal methods are often domain-specific or rely on structured event representations. We propose a runtime verification (RV) framework that treats an LLM conversation as an execution trace that can be formally verified and develop a corresponding tool called TemporalGuard. It observes the stream of user messages and LLM-generated assistant responses, grounds each message into a set of atomic propositions, and thereby constructs a Boolean-labeled trace. Safety policies are specified as formulas in *past-time linear temporal logic* (ptLTL), and the monitor checks the evolving Boolean trace online to decide whether the conversation satisfies the policy. A central challenge is grounding: bridging the gap between the precise Boolean semantics of temporal logic and the ambiguity of natural language utterances. To address this, we developed a semantic grounding layer and experimentally evaluated a range of grounding strategies, including an embedding-based approach, Natural Language Inference (NLI), and LLM-based zero/few-shot classification. We demonstrate the effectiveness of TemporalGuard through grounding and end-to-end monitoring experiments.

1 Introduction

Large Language Models (LLMs) are rapidly being integrated into conversational systems such as customer-support agents, healthcare assistants, educational tutors, coding assistants, and collaborative multi-agent platforms. As these systems

* The research performed by Itay Cohen, Moran Omer and Doron Peled was partially funded by Israeli Science Foundation grant 2454/23: "Validating and controlling software and hardware systems assisted by machine learning". The research performed by Klaus Havelund was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

increasingly interact directly with users, ensuring that their behavior complies with safety policies during deployment has become an important challenge.

Although recent training-time alignment and safety techniques have substantially improved the behavior of large language models, they do not fully eliminate unsafe outputs. As a result, deployed LLM systems often incorporate additional safeguards during operation to monitor or constrain model behavior according to safety policies.

Yet such safeguards remain susceptible to adversarial use. Existing work has shown that LLMs can be manipulated through prompt injection [26], jailbreak attacks [38], and multi-turn conversational strategies that gradually induce unsafe behavior across dialogue context [20]. These limitations suggest that enforcing safety in conversational systems requires mechanisms that can reason not only about individual messages, but also about how risk emerges across dialogue context.

The approach we present in this work is to monitor the user and LLM interaction by treating a conversation as an evolving trace that can be checked against a formal safety specification. In this setting, an external monitor observes the sequence of user and LLM-generated assistant messages; it determines which atomic propositions hold at each step, and thereby constructs a Boolean trace representing the conversation at the semantic level. The resulting trace can then be verified against safety properties expressed in the chosen logical formalism. The monitor sits inline between the user and the LLM: after evaluating each incoming message against the policy, it either forwards the message or blocks it based on the current verdict, following the runtime enforcement perspective [10]. In this work, the safety properties we consider are formalized in past-time linear temporal logic and are referred to as policies. This perspective draws on techniques from *runtime verification* (RV), a branch of formal methods concerned with checking whether execution traces satisfy logical specifications during system operation [2, 19]. Figure 1 illustrates the high-level architecture of our approach. We introduce TemporalGuard, a tool that demonstrates our approach by allowing a supervisor to define temporal policies over LLM conversations and monitor their satisfaction as the dialogue unfolds.

Applying runtime verification to conversational systems is motivated by three observations. First, training-time alignment of the LLM embeds safety constraints implicitly within the language model parameters. These constraints are sometimes hard to formalize, cannot easily be modified after deployment, and may degrade under adversarial prompting. An external monitoring mechanism can provide an additional enforcement layer that operates independently of the model.

Second, most existing guardrail mechanisms operate on individual messages rather than entire conversations. Keyword filters and blocklists can often be bypassed through paraphrasing or indirect phrasing. Classifier-based moderation systems such as Llama Guard [18] typically analyze messages independently, without explicitly modeling conversational context. As a result, these approaches may fail to detect violations that emerge only through multi-turn interactions.

Third, many safety policies are inherently temporal, related to a sequence of messages during a conversation. For example, a policy may specify that once a user has requested help accessing a private bank account without authorization, the assistant must not provide such guidance at any later point in the conversation. Similarly, if a role-play scenario is introduced and the conversation subsequently escalates toward unsafe content, the assistant must refrain from generating harmful responses. Such requirements depend on relationships between events across conversational history and cannot be expressed using stateless filtering mechanisms.

Despite the maturity of runtime verification techniques, applying them to conversational AI introduces a fundamental challenge. In classical runtime verification, atomic propositions are typically evaluated over software objects such as assignments to the program variables. Conversational systems instead operate over natural language. Determining whether a proposition such as "the user has requested help accessing a private bank account" holds requires semantic interpretation of free-form text.

To address this challenge, we introduce a *semantic grounding* layer that connects temporal logic propositions with natural-language utterances. For this layer, we experiment with several complementary approaches for identifying whether a proposition is expressed in a message: (a) semantic similarity over sentence embeddings, (b) an approach that is based on Natural Language Inference (NLI) models [17, 30], (c) a hybrid method that combines the former two approaches and (d) LLM-based judging using both local open-source and commercial models. In our experiments, the LLM-based grounding approach achieved the highest overall accuracy, while the embeddings and NLI-based methods exhibited uneven performance, succeeding on some propositions but failing on others. Based on these findings, we advocate LLM-based grounding as the preferred method for proposition evaluation. However, we present all approaches in detail, as the lightweight methods may be preferable under latency or cost constraints.

Contributions. This paper makes several contributions. First, we propose a runtime verification approach for conversational AI that models LLM interactions as event traces and specifies safety policies using past-time linear temporal logic with atomic propositions. Second, we introduce a semantic grounding architecture with a proposition evaluation layer that maps natural-language messages to Boolean predicates, and we evaluate multiple grounding strategies on a benchmark of representative safety propositions and labeled messages. Finally, we present a tool, TemporalGuard [5], with illustrative case studies showing how our approach can capture and monitor temporal safety policies in multi-turn LLM conversations.

Related Work

Guardrails and safety mechanisms for LLMs. Ensuring the safety of large language models has become an active research area. Alignment techniques such

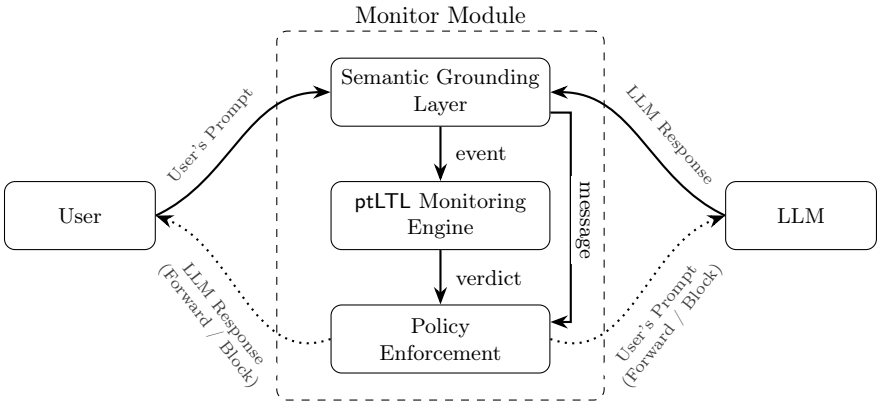

Fig. 1: Conversation monitoring architecture. A standalone module intercepts each user message and assistant response, grounds the message against predefined patterns to produce a Boolean-labeled event, updates the ptLTL monitor, and forwards or blocks the message based on the monitor's verdict.

as Reinforcement Learning from Human Feedback (RLHF) [25], Constitutional AI [1] and Direct Preference Optimization [28] aim to shape model behavior during training.

At inference time, various guardrail mechanisms have been proposed. NVIDIA's NeMo Guardrails [29] provides a domain-specific language for specifying conversational control flows. Other moderation tools focus on detecting toxic or offensive language through supervised learning [8, 13]. In [34], LLMs are trained to prioritize system-level instructions over user inputs, improving robustness against prompt injections and jailbreaks, but remain limited to model-internal enforcement with no temporal awareness of conversational history. While these approaches improve practical safety, they typically lack formal temporal semantics and often evaluate messages independently rather than considering the full conversational history. Indeed, these stateless mechanisms are known to be vulnerable to adversarial prompting strategies, including prompt injection, single-turn jailbreaks, and multi-turn conversational attacks that gradually manipulate model behavior across turns.

Recent work on formal specifications and temporal constraints has focused mainly on LLM agents and robotics. AgentSpec [31] introduces a domain-specific language for runtime constraints over agent events, but assumes structured event streams from agent workflows rather than grounding propositions from open-ended dialogue text. Agent-C [24] enforces temporal constraints over agent action sequences, whereas our monitor reasons over role-tagged natural-language utterances across user and assistant messages. RoboGuard [21] translates natural-language safety rules into formal constraints for LLM-enabled robotic control, but its monitored state is specific to robot actions and environment transitions, not conversational message traces with proposition-level semantic grounding.

In prior work that combines temporal logic with LLMs for conversational assessment, LogiDebrief [36] studies post-hoc quality analysis of 9-1-1 dispatch calls. It represents dispatch requirements as Signal Temporal Logic (STL) formulas over the call timeline, and then uses LLM-based evaluators to determine whether each requirement is satisfied by the conversation. Our approach differs in three ways: (i) we target online runtime safety enforcement rather than retrospective call auditing, (ii) we specify policies over finite conversational traces with past-time operators instead of domain-specific STL templates, and (iii) we support grounding predefined propositions at specification time, while LogiDebrief relies on a predefined, expert-curated set of requirements predicates.

RvLLM [42] addresses a different verification problem from ours. It checks LLM outputs against expert-defined domain knowledge encoded in ESL, and although it can reuse inferred facts to issue follow-up queries and assess consistency across successive responses, it does not formulate verification as temporal monitoring over a dialogue trace. Our framework instead models the conversation itself as a trace of role-tagged user and assistant messages, grounds each message against predefined natural-language patterns, and verifies ptLTL policies over the resulting trace.

A particularly relevant prior work is RV4Chatbot [11, 12], which targets intent-based chatbots such as Rasa and Dialogflow, where the conversation is first mapped into a predefined set of symbolic events supplied by the chatbot designer, including intent labels, extracted entities, and bot actions. Runtime verification is then performed over this fixed event vocabulary to check whether the interaction follows the expected protocol. Our setting is different: we monitor open-ended LLM conversations directly from raw role-tagged messages, without assuming any predefined intent inventory or framework-specific event schema. Instead, a policy in our framework may contain arbitrary textual descriptions defined by the supervisor at specification time, and the key challenge is to ground each free-form conversation message against those predefined descriptions before applying temporal monitoring.

Semantic grounding and related NLP tasks. In our framework, the grounding task is to determine, for each conversation message, whether it matches a predefined description. The input is therefore a text message together with a description written in natural language, and the output is a Boolean decision indicating whether the message matches that description. Unlike standard supervised classification, the target descriptions are not fixed in advance, but are supplied dynamically as part of the monitored policy. From a natural language processing (NLP) perspective, this setting is closest to *zero-shot text classification*. In zero-shot classification, the goal is to assign a text to one or more labels without task-specific training examples for those labels [41]. This view is closely related to our setting, except that our "labels" are predefined descriptions rather than elements of a fixed, predefined taxonomy.

One common way to perform zero-shot classification is through *embedding-based semantic matching*. In this approach, the input text and the candidate label descriptions are mapped into a shared vector space, and classification is

performed using similarity scores such as cosine similarity. Prior work has explored joint input label embedding methods for text classification [35], as well as more recent sentence-embedding-based approaches for zero-shot and unsupervised classification [32].

Another closely related approach relies on *natural language inference* (NLI). NLI is typically formulated as predicting whether a premise entails, contradicts, or is neutral with respect to a hypothesis, as in benchmark datasets such as SNLI and MultiNLI [3,39]. This formulation naturally supports zero-shot classification by treating the input text as the premise and each candidate label as a verbalized hypothesis. Under this view, classification reduces to deciding whether the text entails the label description [41].

More recently, zero-shot classification has increasingly been performed with *large language models*. The variant most relevant to our setting is *label-wise judging*, in which the model is prompted with the input text and a natural-language description of a candidate label, and is asked to decide whether that label applies. This formulation is especially suitable for our grounding problem, since the descriptions are supervisor-defined and can be supplied directly in natural language at inference time [27, 37].

Motivated by these connections, our grounding layer experiments with methods drawn from all three paradigms discussed above: embedding-based semantic matching, NLI-based zero-shot classification, and LLM-based label-wise judging. We evaluate these alternatives in the setting of predefined propositions over conversational utterances.

2 Past-Time Linear Temporal Logic

We model an LLM conversation as a finite trace of messages specifying safety policies as formulas in past-time linear temporal logic (ptLTL) [22]. In runtime verification (RV) [19], a monitor observes an execution trace and checks compliance with such a formal specification at each step. ptLTL is particularly well-suited to conversational monitoring for two reasons: (1) formulas refer only to the present and past, so the truth value is *determined* at each step with no inconclusive verdicts, meaning that a single violation irrevocably constitutes a policy breach; and (2) monitors run in time and space complexity that is linear in the size of the specification φ per event [16]. This makes them lightweight enough to operate inline between the user and the LLM without introducing significant latency.

Syntax. Let AP be a finite set of atomic propositions. We view each event as a set $e \subseteq AP$ indicating which propositions hold at that step. The formulas of ptLTL are built from the following grammar, where $p \in AP$:

$$\varphi \ ::= \ \mathit{true} \ \mid \ p \ \mid \ \neg\varphi \ \mid \ (\varphi \wedge \varphi) \ \mid \ \ominus\varphi \ \mid \ (\varphi \, \mathcal{S} \, \varphi)$$

We also define the following derived operators: $\mathit{false} = \neg\mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondsuit\varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ ("once", or "previously"), and $\boxminus\varphi = \neg\diamondsuit\neg\varphi$ ("historically", or "always in the past").

The operators have the following informal meaning: $\ominus\varphi$ (*yesterday* φ) asserts that φ "was true in the previous step". $(\varphi\,\mathcal{S}\,\psi)$ (φ *since* ψ) holds if both ψ "held in the past", and φ "has been holding ever since". $\diamondsuit\,\varphi$ (*once* φ) holds if φ "held sometime in the past", and $\boxminus\,\varphi$ (*historically* φ) holds if φ "has always held in the past". The following semantics definition formalizes how the different operators are interpreted over an observed trace.

Semantics. Let $\sigma = e_0\, e_1 \cdots e_n$ be a finite trace of events. We write $(\sigma, i) \models \varphi$ to mean that φ holds at position i of σ, defined inductively as follows [9, 16]:

$$
\begin{array}{ll}
(\sigma,i) \models p & \text{iff } p \in e_i \\
(\sigma,i) \models \neg\varphi & \text{iff } (\sigma,i) \not\models \varphi \\
(\sigma,i) \models (\varphi_1 \wedge \varphi_2) & \text{iff } (\sigma,i) \models \varphi_1 \ \wedge\ (\sigma,i) \models \varphi_2 \\
(\sigma,i) \models \ominus\varphi & \text{iff } i > 0 \text{ and } (\sigma, i{-}1) \models \varphi \\
(\sigma,i) \models (\varphi_1\,\mathcal{S}\,\varphi_2) & \text{iff } \exists\, j \le i.\ (\sigma,j) \models \varphi_2 \ \textit{ and } \ \forall k.\ j < k \le i \,(\sigma,k) \models \varphi_1
\end{array}
$$

In runtime verification, the monitor evaluates after each new event in the observed prefix the verdict, which is the truth value of φ. This verdict may vary across steps as the trace grows, so φ can alternate between *True* and *False* over time.

Monitoring algorithm. The classical algorithm for monitoring propositional past-time LTL [15, 16] maintains two Boolean vectors over the subformulas of φ. The vector pre stores the truth value of each subformula based on the trace observed so far *except* the last event. The vector now stores the truth value of each subformula based on the trace *including* the last event. When a new event e occurs, now is copied to pre, and a new now is computed from pre and e as follows:

- $\mathsf{now}(\mathit{true}) = \mathit{True}$
- $\mathsf{now}(p) = (p \in e)$
- $\mathsf{now}(\neg\varphi) = \neg\,\mathsf{now}(\varphi)$
- $\mathsf{now}(\varphi \wedge \psi) = (\mathsf{now}(\varphi) \wedge \mathsf{now}(\psi))$
- $\mathsf{now}(\ominus\varphi) = \mathsf{pre}(\varphi)$
- $\mathsf{now}(\varphi_1\,\mathcal{S}\,\varphi_2) = \mathsf{now}(\varphi_2) \ \vee\ (\mathsf{now}(\varphi_1) \wedge \mathsf{pre}(\varphi_1\,\mathcal{S}\,\varphi_2))$

Both vectors are initialized to *False* before the first event, reflecting the absence of any past. This scheme requires two Boolean variables per subformula, yielding $O(|\varphi|)$ memory and time per event. The operators $\diamondsuit$ and $\boxminus$ are treated as derived forms and are evaluated using their standard translations to the $\mathcal{S}$ operator.

3 Conversation Monitoring Approach

In this section, we present our monitoring approach and introduce the formal definitions needed to connect our conversational setting to the runtime verification framework. Our goal is to monitor conversations between a human user and a large language model through an approach grounded in runtime verification. Throughout the paper, we reserve the term *user* for the human participant who

interacts with the assistant LLM during the monitored conversation. We use the term *supervisor* for the person, such as a system administrator or policy author, who configures the monitor by defining policies.

Below are the formal definitions of a message and a conversation trace in our setting:

Definition 1 (Message). *A* message *is a tuple* $m = (role, text)$ *where* $role \in \mathcal{R} = \{\mathsf{user}, \mathsf{assistant}\}$ *is the message sender identity and text is the natural-language content of the message.*

The set of all messages is denoted by $\mathcal{M}$. We refer to messages produced by the human participant as *user messages*, and to messages produced by the language model as *assistant messages*.

Definition 2 (Conversation Trace). *A* conversation trace *is a finite sequence* $\sigma = m_0\, m_1 \cdots m_n$ *of messages. We write* $|\sigma| = n + 1$ *for its length and* $\sigma_{\leq i}$ *for the prefix up to position* i *for some* $i \geq 0$.

After step i, the monitor has access only to the current prefix of the conversation trace $\sigma_{\leq i}$ and must emit a verdict based solely on that prefix. This is consistent with the standard RV setting [19], where a monitor observes events one at a time as they occur and evaluates the specification incrementally.

Since we monitor conversations consisting of text messages, a key part of the monitoring process is to determine whether each message matches a supervisor-defined textual description. Concretely, the supervisor specifies textual criteria that describe what should be detected in a message. We refer to such criteria as *patterns*.

Definition 3 (Pattern). *A pattern is a triple* $p = (id_p, \delta_p, role_p)$ *where* id_p *is a Boolean proposition,* δ_p *is a text string that represents the description to be detected, and* $role_p \in \mathcal{R}$ *is the role, identifying the sender. The truth value of the proposition* id_p *is evaluated with respect to a specific message from* $\mathcal{M}$*: it is True for a message if the message matches that pattern, and False otherwise.*

We refer to the process of determining whether a message matches a pattern as *grounding*. Formally, we define a grounding function that evaluates a single message against a single pattern:

$$\mathsf{ground} : \mathcal{M} \times \mathcal{P} \to \{\mathit{True}, \mathit{False}\},$$

where $\mathcal{P}$ is the set of patterns. For a message $m = (role, text)$ and a pattern $p = (id_p, \delta_p, role_p)$, $\mathsf{ground}(m, p) = \mathit{True}$ iff m matches the description δ_p and $role = role_p$. In practice, ground is realized by a semantic grounding layer, to be described in Section 4. We also define a labeling function λ that maps each message to the set of Boolean pattern propositions that evaluate to *True* for it:

$$\lambda(m) \;=\; \{\, id_p \mid p \in \mathcal{P} \;\wedge\; \mathsf{ground}(m, p) = \mathit{True} \,\}.$$

Hence, $\lambda(m)$ provides a Boolean event abstraction of the message. In preliminary experiments with our grounding methods, we found that conditioning this decision on the sender role improves accuracy when determining whether a message matches a pattern. This motivates including the explicit role constraint $role_p$ in each pattern. In the next section, we present three alternative methods for implementing the semantic grounding layer.

Monitoring is performed with respect to a supervisor-defined policy expressed as a ptLTL formula. A policy is constructed from Boolean connectives, the temporal operators of ptLTL, and atomic propositions. The translation of natural-language requirements into formal specifications is a nontrivial problem and is outside the scope of this paper; we have studied LLM-based approaches to this translation problem in prior work [6]. In our setting, an atomic proposition is either:

- A pattern proposition id_p for some $p \in \mathcal{P}$.
- The built-in proposition is_user, which evaluates to *True* for message m iff $role(m) = \mathsf{user}$, and to *False* otherwise.

Our approach is implemented as a standalone module that can be inserted into any conversational system in which a user and an assistant (LLM) exchange messages. Every message passed between the user and the assistant is routed through this module, which monitors the interaction against the policy regardless of the specific LLM used by the conversational system.

The monitoring flow proceeds as follows. When the user sends a message m_i, the module applies the semantic grounding layer to compute $\lambda(m_i)$. The resulting set of propositions $\lambda(m_i)$ is treated as an event translated from m_i. This event is then passed to the ptLTL monitoring engine, which updates a per-formula *summary* as it observes the conversation trace. Concretely, the engine applies the classical incremental monitoring algorithm of Section 2, updating the `pre` and `now` Boolean vectors for each policy formula across the evolving trace. If, after the update, there exists a policy formula whose current verdict is *False* (which constitutes a policy violation), the module blocks the current and subsequent messages, and returns a violation alert to the user. Otherwise, the message is forwarded to the assistant (the LLM). The workflow is then applied again when the assistant responds.

While the presentation above focuses on two roles, the approach extends directly to additional participants by expanding the role set and associating each pattern with the appropriate role constraint.

Motivating Specification Examples. We present three specification examples that motivate our approach. In each example, patterns are referred to by their corresponding pattern identifiers. Each example illustrates a policy that is expressed using a temporal specification about the conversational history, where its interpretation is based on the mapping of each message (user prompt, assistant response) to a Boolean assignment using grounding.

Example 1: Medical assistant safeguard. Consider an LLM deployed as a medical assistant. The policy requires that once the user has requested a diagnosis at any earlier step, the assistant must not provide a definitive diagnosis without recommending consultation with a medical professional. To map the conversation into Boolean assignments, the supervisor defines two patterns (see Definition 3), with identifiers p_{sym} and p_{diag}: (1) p_{sym} describes "the user describes symptoms and requests a diagnosis" and has role user; and (2) p_{diag} describes "the assistant presents a definitive diagnosis without recommending consultation with a medical professional" and has role $\mathsf{assistant}$. The temporal property is

$$\diamondminus\, p_{sym} \rightarrow \neg p_{diag}.$$

Enforcing this policy naturally requires tracking the conversational history.

Example 2: Escalation under repeated unsafe requests. Consider an LLM deployed as a general-purpose assistant. The policy requires that an assistant message immediately following a hacking-related user message must include a refusal; if the user repeats such a request after it has occurred previously in the conversation, the subsequent assistant message must both refuse and include an explicit escalation. To map the conversation into Boolean assignments, the supervisor defines three patterns: (1) p_{hack} with the description "the user asks about hacking or bypass techniques" and role user; (2) p_{refuse} with the description "the assistant refuses to cooperate" and role $\mathsf{assistant}$; and (3) p_{escalate} with the description "the assistant explicitly asks the user not to raise this topic again" and role $\mathsf{assistant}$. The temporal property is

$$\left(\ominus\, p_{\mathsf{hack}} \rightarrow p_{\mathsf{refuse}}\right) \;\wedge\; \left(\ominus\left(p_{\mathsf{hack}} \wedge \ominus(\diamondminus\, p_{\mathsf{hack}})\right) \rightarrow p_{\mathsf{escalate}}\right).$$

Intuitively, the first conjunct enforces an immediate refusal whenever a hacking request occurs. The second conjunct detects repetition by checking $\ominus(\diamondminus\, p_{\mathsf{hack}})$: it holds exactly when, in the previous position, it was already the case that a hacking request had occurred at some earlier point in the trace. Thus, after a repeated hacking request, the policy requires the assistant response to satisfy both the refusal condition p_{refuse} and the stronger escalation condition p_{escalate}.

Example 3: Customer-support compliance protocol. Consider an LLM deployed as a customer support agent that must follow a consistent compliance protocol and, once the assistant has responded at least once, restrict subsequent user messages to a small set of supported issue types. The policy enforces two constraints: (i) whenever the previous message was sent by the user, the next assistant message must include a compliance footer, i.e., a short privacy/security note such as "do not share full credit card numbers"; and (ii) once the assistant has responded at least once, every subsequent user message must be either a billing/refund message or a shipping/delivery message. To map the conversation into Boolean assignments, the supervisor defines three patterns: (1) p_{foot} with the description "the assistant includes a short security note (e.g., *do not share full credit card*

numbers)" and role assistant; (2) p_{bill} with the description "the user message concerns billing/refund" and role user; and (3) p_{ship} with the description "the user message concerns shipping/delivery" and role user. In addition, the policy uses the built-in proposition is_user that evaluates to *True* exactly when the current message is a user message. The temporal property is

$$\big(\ominus\, \mathsf{is\_user} \rightarrow p_{\mathsf{foot}}\big) \ \wedge\ \big((\mathsf{is\_user} \wedge \Diamond\!\!\!-\, p_{\mathsf{foot}}) \rightarrow (p_{\mathsf{bill}} \vee p_{\mathsf{ship}})\big)\,.$$

These three examples motivate the temporal logic framework and semantic grounding layer developed in the following section.

4 Semantic Grounding

In this section, we present the semantic grounding layer. At each step, this component determines which pattern propositions evaluate to *True* for the current message by deciding, for each pattern, whether the message matches it.

The grounding layer implements the grounding function defined in Section 3. For each message $m_i = (role_i, text_i)$ and each pattern $p = (id_p, \delta_p, role_p)$, the layer decides whether $\mathsf{ground}(m_i, p) = \mathit{True}$ in two stages:

1. **Role filter.** If $role_i \neq role_p$, the pattern does not apply to this message and *False* is returned.
2. **Semantic scoring.** A scorer $s_p : \mathcal{T} \rightarrow [0, 1]$ measures how closely $text_i$ matches the pattern described by δ_p. The grounding decision is positive when the score meets a configurable threshold:

$$\mathsf{ground}(m_i, p) = \mathit{True} \iff role_i = role_p \ \wedge\ s_p(text_i) \ \geq\ \theta_{\mathrm{match}} \qquad (1)$$

Each pattern has its own scorer function, allowing patterns to be configured or replaced independently.

We developed and evaluated three alternative scoring methods: embedding-based grounding (Section 4.1), Natural Language Inference based grounding (Section 4.2), and LLM-based grounding (Section 4.3).

4.1 Embedding-Based Grounding

The first grounding method relies on *sentence embeddings*: numerical vectors that encode the meaning of a text in a high-dimensional space, such that semantically similar sentences are mapped to similar vectors. Similarity between vectors is measured using *cosine similarity* [23, 33]: cosine similarity compares the angle between two vectors, where a higher cosine value indicates that the two texts are more semantically aligned.

We represent each pattern description by a set of example embeddings that approximate its relevant semantic region, and score an incoming message based on its proximity to these examples. The method consists of an offline preparation phase and an online scoring phase that runs during the conversation as each message arrives.

Offline: anchor generation. A *sentence encoder* $\mathcal{E}$ (e.g., all-mpnet-base-v2 [30]) maps text strings to dense vectors in $\mathbb{R}^d$, where cosine similarity between vectors represents semantic relatedness. For each pattern p, we precompute three sets of representative texts, called *anchor sets*, and embed them (using $\mathcal{E}$) into $\mathbb{R}^d$: **positive anchors** A_p^+ (diverse examples that genuinely match p), **negative anchors** A_p^- (near-miss examples that share vocabulary with p but clearly do not match the pattern), and **neutral anchors** A_p^0 (off-topic examples that do not match the pattern). The three sets jointly partition the embedding neighborhood of p into regions of positive, near-miss, and off-topic semantics. Example anchor sets generated for our patterns are included in the project repository [5].

Anchors are generated through an LLM-driven procedure designed to promote semantic diversity in embedding space. Positive and negative anchors are produced in multiple rounds. First, an initial step uses an LLM to generate a core set of anchor texts across configurable categories (e.g., information-seeking requests, creative generation requests, procedural requests), which are then embedded using the sentence encoder $\mathcal{E}$. Next, we run k-means clustering over the anchor embeddings (with the number of clusters set to twice the number of anchor categories, which worked well in our preliminary experiments) to partition the space into semantic regions and identify clusters that currently contain relatively few anchors. We then repeatedly prompt the LLM to propose additional candidate anchors, embed each candidate, and assign it to its nearest k-means cluster. A candidate is added to the anchor sets only if its embedding falls into one of the sparsely populated clusters. This *generate–embed-add* loop is repeated iteratively until the desired coverage is reached. Finally, from the resulting pool, a selection step [4] chooses a subset that balances high relevance to the pattern with broad coverage of the embedding space.

Online: K Nearest Neighbors (KNN) voting. At evaluation time, the scorer embeds the message $text_i$. Then we retrieve the K anchors in $A_p^+ \cup A_p^- \cup A_p^0$ with the highest cosine similarity to $text_i$, denoted by top-$K(text_i)$, and define the score as the proportion of positive anchors among these K neighbors [7]:

$$s_p^{\cos}(text_i) \;=\; \frac{|\{a \in \text{top-}K(text_i) \mid a \in A_p^+\}|}{K} \tag{2}$$

If the proportion of positive anchors among the K nearest neighbors exceeds the match threshold θ_{match}, we treat the message as matching the pattern; otherwise, the message is considered a non-match.

Since the anchor sets are generated and embedded offline, online evaluation is computationally lightweight. At runtime, the method only embeds the message and computes cosine similarities to the precomputed anchor embeddings, followed by a simple KNN vote.

For long messages that may contain pattern-relevant content mixed with otherwise benign text, we apply a multi-view step that splits the message into sentences, forms overlapping windows of adjacent sentences, scores each window independently, and takes the maximum score to capture a pattern match regardless of where it appears.

4.2 NLI-Based Grounding

Cosine similarity in embedding space captures topical relatedness but can fail to distinguish semantic roles: the sentences "how to prevent credit card fraud" and "how to commit credit card fraud" may receive similar embeddings despite opposite intents. To address this, we introduce a second grounding method based on Natural Language Inference (NLI). NLI is the task of determining whether a natural-language hypothesis is entailed by, contradicted by, or neutral with respect to a given premise [3,39]. We use an NLI model (e.g., DeBERTa-v3 [17]) that takes the message and an anchor as a joint input and produces three probabilities: P_E (entailment), P_N (neutral), and P_C (contradiction).

Offline: anchor reuse. This method reuses the same anchor sets generated for the embedding-based approach (Section 4.1). No additional offline preparation is required.

Online: cosine prefilter with NLI re-ranking. Running the NLI model over all the anchors for every message would be expensive. We therefore use a two-stage procedure: first, cosine similarity (as in the embedding-based method) filters the anchor set to the top R candidates most similar to the message. Then, the NLI model is applied to each (message, anchor) pair and outputs probabilities P_E, P_N, and P_C for entailment, neutral, and contradiction. In this re-ranking stage, we treat all anchors uniformly and do not use their positive/negative/neutral labels; instead, we rely on the NLI outputs to assess alignment between the message and each anchor. From these outputs, we compute a score $P_E - P_C$, i.e., the difference between the entailment and contradiction probabilities. A positive score indicates that the message expresses the same intent as the anchor, while a negative score may indicate that the message does not match the anchor. The top K anchors by this score are selected, and the final score is computed by KNN voting exactly as in Equation 2.

Hybrid scoring. We also provide a hybrid mode that combines both methods. The embedding-based method selects its top $K/2$ anchors and the NLI based approach selects its own top $K/2$ anchors, forming a merged voter pool of size K. The final score is computed by KNN voting over this merged pool, giving higher weight to anchors that both methods agree on.

4.3 LLM-based Grounding

The third grounding approach uses an LLM to decide whether a message matches a pattern. This grounding LLM is invoked separately from the assistant LLM being monitored, with its own classifier prompt and context. This LLM may use the same underlying model family, but it is a separate session used only for proposition evaluation. This task can be viewed as a binary classification via prompting. LLMs tend to perform well in this setting, and in our experience the task remains effective even with relatively small LLMs. Supporting small, locally runnable LLMs is important for our setting, since it enables fully on-premises deployment of the grounding layer without relying on remote providers, while still achieving high grounding accuracy. This is beneficial both for cost and for

security and privacy, as organizations can evaluate sensitive user and assistant messages locally without sending conversation content to external services. We therefore also evaluate this grounding approach with smaller LLMs that can be run locally (Section 6).

Few-shot grounding. Few-shot grounding incorporates pattern-specific labeled demonstrations to guide the matching decision.

Offline: demonstration generation. When a pattern is first defined by the supervisor, we use the supervisor-provided description to generate a small set of labeled message demonstrations by querying an LLM. Positive examples are messages that clearly match the pattern. Negative examples are *challenging near-misses*: messages that use similar terminology and context to the pattern and may appear relevant, but do not actually match it. Since this generation is performed only once per pattern in an offline phase, it can leverage stronger (and potentially slower) LLMs to produce diverse and high-quality examples.

Online: prompted classification during the conversation. At runtime, when a new message m_i arrives, the grounding layer prompts an LLM to evaluate $\mathsf{ground}(m_i, p)$ for each relevant pattern. Each query uses a fixed *system prompt*: an instruction prefix that defines the classification task and emphasizes distinctions important for monitoring (e.g., harmful requests versus benign educational mentions).

The message-specific information is then provided in a *user prompt* that includes (i) pattern description, (ii) the current message text, and (iii) the labeled few-shot demonstrations generated offline. To mitigate prompt-injection risk, the prompt explicitly separates this inserted content from the classifier instructions and instructs the model to treat it as untrusted data. In addition, the prompt includes concise classification instructions that are role-aware: the instructions differ depending on whether the current message text originates from a *user message* or an *assistant message*, since the same topic can carry different policy implications across roles. The LLM returns a structured output (e.g., a JSON object) containing a Boolean match decision and a brief explanation. The match decision is considered as the value of $\mathsf{ground}(m_i, p)$. The concrete prompt templates are included in the accompanying repository [5].

Zero-shot grounding. Zero-shot grounding uses the same overall prompting structure but omits the pattern-specific demonstrations. The user prompt contains the pattern description, the current message text, and the same role-aware classification instructions. This mode has minimal setup cost and is easy to apply to new patterns. However, because the LLM receives no pattern-specific examples, it is typically more sensitive to borderline cases where intent must be distinguished from topical overlap.

5 The System TemporalGuard

To demonstrate the feasibility of the conversation monitoring approach described in Sections 2–4, we implemented TemporalGuard, a web-based proof-of-concept system [5]. It provides a chat-style interface for interacting with a wide variety of LLMs as an assistant, while allowing a supervisor to define monitoring policies and enforce them in real time during the conversation.

Architecturally, TemporalGuard realizes the standalone monitoring module described in Section 3 by placing it inline between the user and the assistant (LLM) model. Every user message and assistant response is routed through this module before being forwarded, so monitoring is independent of the particular LLM used for the chat. The module consists of three components. The first is a semantic grounding component that implements the function $\mathsf{ground}(m, p)$ described in Section 3. Given a message and a pattern, it returns a Boolean verdict indicating whether the message matches that pattern. Based on the empirical results of our grounding evaluation (to be described in Section 6.1), which indicate that few-shot LLM-based grounding yields the highest accuracy, TemporalGuard adopts this approach as its default grounding method. The LLM used for the grounding task can be deployed either locally (e.g., via a local server such as Ollama) to satisfy privacy constraints, or remotely via a cloud provider when local resources are limited. In our cloud-based configuration, evaluating a grounding prompt for a single pattern took at most four seconds, which is acceptable for interactive real-time message processing.

The second component is the runtime verification engine, which implements the incremental ptLTL monitoring algorithm of Section 2. For each policy formula, the engine maintains the pre and now Boolean vectors over subformulas and updates them on each Boolean-labeled event induced by grounding.

The third component is a policy enforcement layer that acts on the verdict of the RV engine. For each new event induced by grounding, it checks whether any policy formula evaluates to *False* and, if so, blocks the corresponding message and returns a violation alert. Enforcement applies to both user and assistant messages.

TemporalGuard provides a web interface for defining and managing policies, shown in Figure 2. The supervisor first defines patterns by specifying proposition names and a description and selecting a role constraint (Definition 3). Policies are then defined as ptLTL formulas composed from pattern propositions and temporal operators such as $\boxminus$ (UI syntax `H`), $\diamondminus$ (UI syntax `P`), $\ominus$ (UI syntax `Y`), and $\mathcal{S}$ (UI syntax `S`).

6 Experiments

We evaluate our approach using two complementary experiments. The first set of experiments (Section 6.1) assesses the accuracy of the semantic grounding layer against a dedicated dataset. The second set of experiments (Section 6.2) evaluates the end-to-end monitoring approach using TemporalGuard, with LLM-based few-shot grounding.

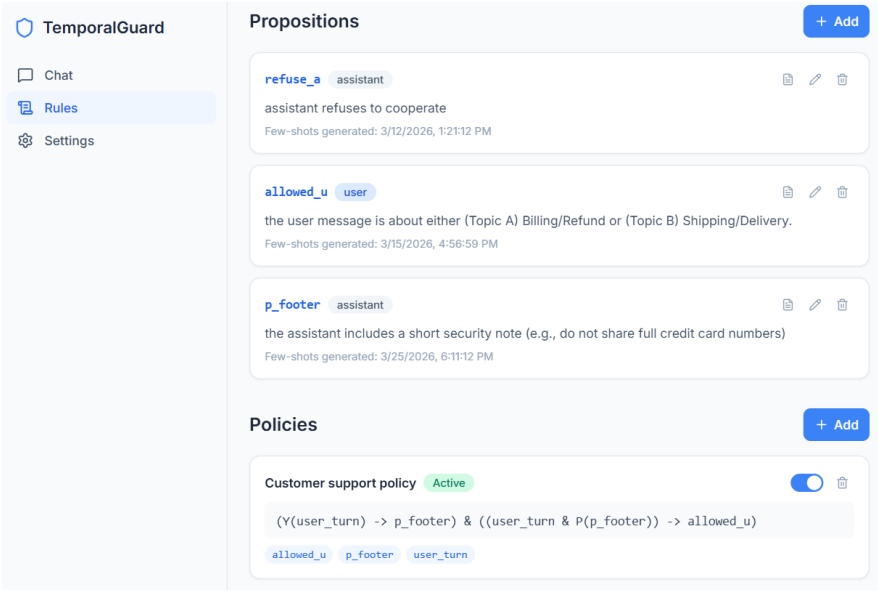

Fig. 2: TemporalGuard web application rules screenshot

In the grounding experiment, we compare five grounding approaches: embedding-based scoring, NLI-based scoring, a hybrid of embedding and NLI, LLM-based zero-shot classification and LLM-based few-shot classification. For the end-to-end experiment, we use our TemporalGuard tool (Section 5), which integrates the semantic grounding layer with ptLTL monitoring over conversations. The tool and all grounding implementations are publicly available in our repository [5]. The non-LLM experiments were conducted on an Apple M1 machine with 16 GB of RAM. LLMs were accessed via API.

6.1 Grounding Experiments

The grounding experiment compares semantic grounding approaches in terms of their accuracy in matching messages to patterns. We evaluate all methods on the labeled dataset described next.

Message Dataset. We constructed a dataset of 100 labeled messages across 10 patterns covering safety-relevant behaviors from both the user and assistant perspectives, such as hacking, financial fraud, privacy leakage, unsafe medical or legal advice, and sensitive personal data. For each pattern, we created five positive examples (messages that match the pattern) and five negative examples, yielding 50 positive and 50 negative messages in total. The messages were a mixture of manually authored and LLM-generated examples. Negative examples were constructed to be adversarial: they intentionally include terminology closely related to the pattern's domain (e.g., "What exactly is a chargeback?" for a pattern about financial fraud) while expressing clearly benign intent, testing

Table 1: Aggregate grounding accuracy across 10 patterns (100 messages), where TP = true positive, FP = false positive, TN = true negative, FN = false negative.

Mode	TP	FP	TN	FN	Prec.	Recall	F1	Acc.
Embedding-based	49	14	36	1	77.8%	98.0%	86.7%	85.0%
NLI-based	45	7	43	5	86.5%	90.0%	88.2%	88.0%
Hybrid	49	8	42	1	86.0%	98.0%	91.6%	91.0%
LLM (zero shot)	45	0	50	5	100.0%	90.0%	94.7%	95.0%
LLM (few shot)	50	0	50	0	100.0%	100.0%	100.0%	100.0%

whether the grounding method can separate intent from superficial vocabulary overlap.

Grounding methods. We evaluated five grounding approaches, all described in Section 4, organized into two categories.

Anchor-based methods. This category includes three approaches: embedding-based scoring (Section 4.1), NLI-based scoring, and hybrid scoring (Section 4.2). All three rely on precomputed sets of positive and negative anchor texts generated offline for each pattern. Anchor texts were generated using `Claude Haiku` 4.5, producing approximately 100 positive anchors per pattern, with the exact number varying depending on the semantic diversity of each pattern's domain. The sentence encoder used was `BAAI/bge-large-en-v1.5` [40], and the NLI model used was `cross-encoder/nli-deberta-v3-large` [17]. All three methods were configured with a KNN neighborhood size of $K = 20$ (Equation 2) and a match threshold of $\theta_{\text{match}} = 0.7$ (Equation 1).

LLM-based methods. This category includes two approaches: zero-shot and few-shot classification (Section 4.3). Both use `Claude Sonnet` 4.5 as the LLM.

Results. Table 1 presents the overall performance on the message dataset. The LLM-based few-shot approach achieved the highest accuracy over all the messages (100%), followed by the zero-shot LLM (95%). Among the non-LLM methods, the hybrid approach performed best (91% accuracy), improving over both the NLI-based (88%) and embedding-based (85%) variants.

The error profiles differ substantially across methods. The zero-shot LLM attained perfect precision (no false positives) but had a recall of 90%. In contrast, the embedding-based and hybrid approaches achieved very high recall (98%) but suffered from lower precision due to false positives.

Overall, the three anchor-based methods (embedding-based, NLI-based, and hybrid) exhibit lower precision than the LLM-based methods. A plausible reason is that anchor-based matching is more likely to confuse messages that truly match a pattern with non-matching messages that share overlapping vocabulary with the pattern description. We observed this failure mode most clearly for patterns p_5, p_9, and p_{10}, where distinguishing intent from vocabulary overlap is particularly challenging.

Small local LLMs: zero-shot vs. few-shot grounding. In the experiments above, the LLM-based approach uses an LLM that is not intended for local execution. We therefore additionally evaluated three lightweight models: `Ministral` 3B, `Gemma` 4B, and `Qwen` 3.5 4B, using both the zero-shot and few-shot prompting variants from Section 4.3, on the same 100-message dataset.

Table 2: Grounding results for small LLMs (100 messages).

Mode	TP	FP	TN	FN	Prec.	Recall	F1	Acc.
`Gemma` 4B (zero shot)	45	8	42	5	84.9%	90.0%	87.4%	87.0%
`Gemma` 4B (few shot)	48	0	50	2	100.0%	96.0%	98.0%	98.0%
`Ministral` 3B (zero shot)	33	0	50	17	100.0%	66.0%	79.5%	83.0%
`Ministral` 3B (few shot)	43	0	50	7	100.0%	86.0%	92.5%	93.0%
`Qwen` 3.5 4B (zero shot)	45	0	50	5	100.0%	90.0%	94.7%	95.0%
`Qwen` 3.5 4B (few shot)	47	0	50	3	100.0%	94.0%	96.9%	97.0%

Overall, few-shot prompting improves the performance of all small models, primarily by increasing recall while preserving high precision. `Gemma` 4B benefits most dramatically: few-shot eliminates false positives (FP drops from 8 to 0) and reaches 98% accuracy. `Ministral` 3B is highly conservative in the zero-shot setting (66% recall), and few-shot substantially mitigates this effect (86% recall, 93% accuracy).

`Qwen` 3.5 4B exhibits an especially strong zero-shot performance: its aggregate results (95% accuracy) match the zero-shot performance of the LLM reported earlier in Table 1. Few-shot still provides a consistent gain for `Qwen` 3.5 4B, improving recall to 94% and accuracy to 97%.

6.2 TemporalGuard Experiments

We additionally evaluated TemporalGuard end-to-end using the two non-trivial `ptLTL` specifications introduced in Section 3: the escalation policy (Example 2) and the customer-support compliance protocol (Example 3). Each specification combines multiple patterns and past-time operators to capture multi-turn dependencies.

For each specification, we constructed 20 short conversations (4 to 6 messages) mixing manually authored and LLM-assisted traces: 10 satisfying the specification and 10 containing a violation. We then ran TemporalGuard on each trace and checked whether it reported a violation at any point, using Claude Haiku 4.5 as the grounding model for pattern evaluation. For the escalation specification, TemporalGuard agreed with the expected outcome on 19 out of 20 conversations (95% accuracy). For the customer-support specification, TemporalGuard agreed with the expected outcome on 20 out of 20 conversations (100% accuracy), indicating reliable end-to-end behavior on both temporal patterns.

7 Conclusion

We introduced a runtime verification approach that treats an LLM conversation as a finite trace of messages that can be checked online against safety policies in ptLTL. By grounding supervisor-defined patterns into Boolean propositions, the monitor can enforce temporal requirements over the full conversation and apply them to both user and assistant messages. We illustrated the approach with policy examples and implemented it in the TemporalGuard tool.

A main contribution of our setting is that it supports custom patterns defined by the supervisor at specification time, rather than relying on a predefined inventory of events as in prior work. This flexibility makes semantic grounding a central challenge: it must be accurate yet lightweight, both to keep inline monitoring efficient and to allow local deployment for privacy and security. Our grounding strategies are designed to be computationally light, and the LLM-based approach remains effective even with small models that can run on-premises, enabling local evaluation without sending conversation content to external services. While the paper focuses on two roles, the framework may generalize to additional participants by extending the role set and role constraints in patterns.

An important limitation is that patterns are evaluated per message; supporting multi-message patterns would better capture intent expressed gradually across turns. Another natural extension is to enrich the policy language with first-order features over extracted entities [14].

References

1. Bai, Y., et al.: Constitutional ai: Harmlessness from ai feedback. arXiv preprint arXiv:2212.08073 (2022). https://doi.org/10.48550/arXiv.2212.08073
2. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification – Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5
3. Bowman, S.R., Angeli, G., Potts, C., Manning, C.D.: A large annotated corpus for learning natural language inference. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 632–642 (2015). https://doi.org/10.18653/v1/D15-1075
4. Carbonell, J., Goldstein, J.: The use of MMR, diversity-based reranking for reordering documents and producing summaries. In: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 335–336 (1998). https://doi.org/10.1145/290941.291025
5. Cohen, I., Omer, M., Peled, D.: LLMrv: Runtime verification framework for LLM conversations. https://github.com/moraneus/LLMrv (2026), source code, datasets, and experiment logs
6. Cohen, I., Peled, D.: End-to-end ai generated runtime verification from natural language specification. In: Bridging the Gap Between AI and Reality (AISoLA 2023 Selected Papers). Lecture Notes in Computer Science, vol. 14129, pp. 362–384. Springer (2025). https://doi.org/10.1007/978-3-031-73741-1_23
7. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Transactions on Information Theory **13**(1), 21–27 (1967). https://doi.org/10.1109/TIT.1967.1053964

8. Davidson, T., Warmsley, D., Macy, M., Weber, I.: Automated hate speech detection and the problem of offensive language. In: ICWSM (2017)
9. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Proceedings of the 15th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2725, pp. 27–39. Springer (2003)
10. Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. pp. 89–105. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
11. Gatti, A., Ferrando, A., Mascardi, V.: RV4Rasa: A formalism-agnostic runtime verification framework for verifying ChatBots in Rasa. In: Proceedings of VORTEX (2023). https://doi.org/10.1145/3605159.3605855
12. Gatti, A., Mascardi, V., Ferrando, A.: RV4Chatbot: Are chatbots allowed to dream of electric sheep? arXiv preprint arXiv:2411.14368 (2024)
13. Google Jigsaw: Perspective api. https://perspectiveapi.com (2017)
14. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with bdds. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 116–123. IEEE (2017)
15. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 2280, pp. 342–356. Springer (2002)
16. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. International Journal on Software Tools for Technology Transfer **6**(2), 158–173 (2004). https://doi.org/10.1007/s10009-004-0143-7
17. He, P., Gao, J., Chen, W.: Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing. arXiv preprint arXiv:2111.09543 (2021). https://doi.org/10.48550/arXiv.2111.09543
18. Inan, H., Upasani, K., Chi, J., et al.: Llama guard: Llm-based input-output safeguard for human-ai conversations. arXiv preprint arXiv:2312.06674 (2023)
19. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
20. Li, H., et al.: Multi-turn jailbreak attacks on large language models. arXiv preprint arXiv:2310.01810 (2023)
21. Li, Z., Tang, H., Zhang, Y., Chen, Y., Wang, N., Xie, L., Wang, J.: Roboguard: Safety guardrails for llm-enabled robots. IEEE Robotics and Automation Letters (2026). https://doi.org/10.1109/LRA.2025.3577444
22. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs, Conference. Lecture Notes in Computer Science, vol. 193, pp. 196–218. Springer (1985)
23. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press (2008). https://doi.org/10.1017/CBO9780511809071
24. Osei, E., Lu, M., Bian, Y., Liu, C., Shen, X., Sun, Y., Lin, C., Yuan, Y.: Agent-c: Enforcing temporal constraints for llm agents. arXiv preprint arXiv:2503.04562 (2025). https://doi.org/10.48550/arXiv.2503.04562
25. Ouyang, L., et al.: Training language models to follow instructions with human feedback. In: NeurIPS (2022). https://doi.org/10.48550/arXiv.2203.02155

26. Perez, F., Ribeiro, I.: Ignore previous prompt: Attack techniques for language models. arXiv preprint arXiv:2211.09527 (2022). https://doi.org/10.48550/arXiv.2211.09527
27. Peskine, Y., Korenčić, D., Grubisic, I., Papotti, P., Troncy, R., Rosso, P.: Definitions matter: Guiding GPT for multi-label classification. In: Findings of the Association for Computational Linguistics: EMNLP 2023. pp. 4054–4063. Association for Computational Linguistics, Singapore (2023). https://doi.org/10.18653/v1/2023.findings-emnlp.267, https://aclanthology.org/2023.findings-emnlp.267/
28. Rafailov, R., Sharma, A., Mitchell, E., Manning, C.D., Ermon, S., Finn, C.: Direct preference optimization: Your language model is secretly a reward model. In: Advances in Neural Information Processing Systems (NeurIPS) (2023)
29. Rebedea, T., Dinu, R., Sreedhar, M., Parisien, C., Cohen, J.: Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails. arXiv preprint arXiv:2310.10501 (2023)
30. Reimers, N., Gurevych, I.: Sentence-bert: Sentence embeddings using siamese bert-networks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 3980–3990 (2019). https://doi.org/10.18653/v1/D19-1410
31. Ren, T., Sun, Y., Lu, M., Bian, Y., Lin, C., Yuan, Y.: Agentspec: Formal specification and verification framework for large language model based agent. arXiv preprint arXiv:2503.18666 (2025). https://doi.org/10.48550/arXiv.2503.18666
32. Schopf, T., Braun, D., Matthes, F.: Evaluating unsupervised text classification: Zero-shot and similarity-based approaches. In: Proceedings of the 2022 6th International Conference on Natural Language Processing and Information Retrieval (NLPIR). pp. 6–15. Association for Computing Machinery, Bangkok, Thailand (2023). https://doi.org/10.1145/3582768.3582795, https://doi.org/10.1145/3582768.3582795
33. Singhal, A.: Modern information retrieval: A brief overview. IEEE Data Engineering Bulletin **24**(4), 35–43 (2001)
34. Wallace, E., Xiao, K., Leike, R., Weng, L., Heidecke, J., Beutel, A.: The instruction hierarchy: Training LLMs to prioritize privileged instructions. arXiv preprint arXiv:2404.13208 (2024)
35. Wang, G., Li, C., Wang, W., Zhang, Y., Shen, D., Zhang, X., Henao, R., Carin, L.: Joint embedding of words and labels for text classification. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2321–2331. Association for Computational Linguistics, Melbourne, Australia (2018). https://doi.org/10.18653/v1/P18-1216, https://aclanthology.org/P18-1216/
36. Wang, S., Zhang, L., Zhao, Y., Zhang, C., Chai, H., McAuley, J., Zhu, X., Ji, H.: Logidebrief: Integrating signal temporal logic and llms for enhanced 9-1-1 dispatch debriefing. In: Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI). pp. 9578–9586 (2025). https://doi.org/10.24963/ijcai.2025/1065
37. Wang, Z., Pang, Y., Lin, Y.: Large language models are zero-shot text classifiers. arXiv preprint arXiv:2312.01044 (2023). https://doi.org/10.48550/arXiv.2312.01044, https://arxiv.org/abs/2312.01044
38. Wei, A., Haghtalab, N., Steinhardt, J.: Jailbroken: How does llm safety training fail? arXiv preprint arXiv:2307.02483 (2023). https://doi.org/10.48550/arXiv.2307.02483

39. Williams, A., Nangia, N., Bowman, S.R.: A broad-coverage challenge corpus for sentence understanding through inference. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT). pp. 1112–1122 (2018). https://doi.org/10.18653/v1/N18-1101
40. Xiao, S., Liu, Z., Zhang, P., Muennighoff, N., Liang, D., Dou, Z.: C-Pack: Packaged resources to advance general Chinese embedding. arXiv preprint arXiv:2309.07597 (2024). https://doi.org/10.48550/arXiv.2309.07597
41. Yin, W., Hay, J., Roth, D.: Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 3914–3923. Association for Computational Linguistics (2019). https://doi.org/10.18653/v1/D19-1404, https://aclanthology.org/D19-1404/
42. Zhang, Y., Emma, S.Y., En, A.L.J., Dong, J.S.: Rvllm: LLM runtime verification with domain knowledge. CoRR **abs/2505.18585** (2025)