Monitoring LLM Conversations with First-Order Temporal Logic*

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 used in settings where safe behavior depends not only on the conversational history, but also on the *objects* mentioned across turns, such as accounts, documents, tickets, or products. Existing guardrail mechanisms lack a formal, declarative framework for online monitoring of temporal policies that relate specific entities mentioned across different turns of an open-ended LLM conversation. We present a runtime monitoring approach for LLM conversations based on past-time temporal logic, where policies are checked incrementally as the dialogue unfolds. To support policies that refer to concrete entities mentioned across messages, the monitor also tracks objects and preserves their identities over time. Our approach treats a conversation as a trace of user messages and LLM-generated assistant responses. It grounds each message into predicates with associated objects, and monitors the resulting trace online. It uses a modified version of DejaVu, a runtime monitoring framework for quantified temporal logic. This rich specification language makes it possible to express policies that depend on object identity and numeric relations across turns. A central challenge is grounding: extracting the relevant predicates and objects from natural-language messages efficiently and accurately. To address this, we propose a lightweight grounding approach with two variations. We evaluate our method through grounding experiments and end-to-end monitoring experiments, and implement it in a tool called DejaVuGuard. Our results demonstrate the feasibility of object-aware temporal monitoring as a practical guardrail layer for conversational AI.

1 Introduction

Large language models (LLMs) are increasingly embedded in conversational agents that interact directly with people in domains such as customer sup-

* 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.

port, healthcare services, education, software engineering, legal and financial support, and enterprise productivity. As these human-facing systems become more common, ensuring that their behavior remains within prescribed safety and compliance boundaries during deployment has become a central challenge. This challenge is particularly important in conversational settings, where correctness depends not only on the agent's responses, but also on the evolving interaction between the human and the agent, and on whether both sides of the dialogue conform to the intended safety policy.

While training-time alignment methods have significantly improved the behavior of large language models, they do not guarantee that unsafe outputs will never occur. Consequently, deployed LLM-based systems are often equipped with additional runtime guardrails that monitor or restrict behavior according to explicit safety policies.

Recent work shows that prominent runtime guardrails remain vulnerable in two settings as follows. First, operational safeguards can still be circumvented by prompt-injection attacks, especially when malicious instructions are embedded in retrieved or otherwise untrusted content [31]. Second, guardrails that evaluate messages in isolation, such as keyword filters, blocklists, OpenAI's moderation API, and Llama Guard-style input and output classifiers, are insufficient against multi-turn (multi-message) jailbreaks; consequently, unsafe behavior can emerge gradually across the evolving dialogue context rather than from a single prompt [27,30]. These limitations suggest that effective safeguards should reason not only about individual messages, but also about how risk develops over conversational history.

Such risks are not hypothetical. In 2026, attackers reportedly hijacked Instagram accounts by persuading Meta's AI support assistant [21] to link a new email address to a target account and trigger a password reset, without ever establishing authorized ownership of that account [20]. The failure is temporal and object-centered: a sensitive recovery action was performed even though no trustworthy verification of the same account had occurred earlier in the conversation. We return to this account-recovery case in later parts of the paper.

Our approach monitors the interaction between a human user and a conversational LLM-based assistant using the architecture shown in Figure 1. The monitor is external to the assistant and observes both user and assistant messages as they are produced. This allows it to guard against policy violations caused by either side. The approach builds on *runtime verification* (RV), an area of formal methods that studies how to check, during system execution, whether an evolving trace satisfies a logical specification [2, 8, 14, 18]. To apply RV to conversations, the monitor cannot reason directly over raw natural-language text. Instead, each message is translated into a symbolic event that records which relevant concepts occurred in the message.

We represent these concepts using predicates. A predicate names the kind of concept being monitored, such as a user making a request or an assistant providing information. In the first-order setting used in this work, predicates may also have parameters, which represent the objects mentioned in the message. For

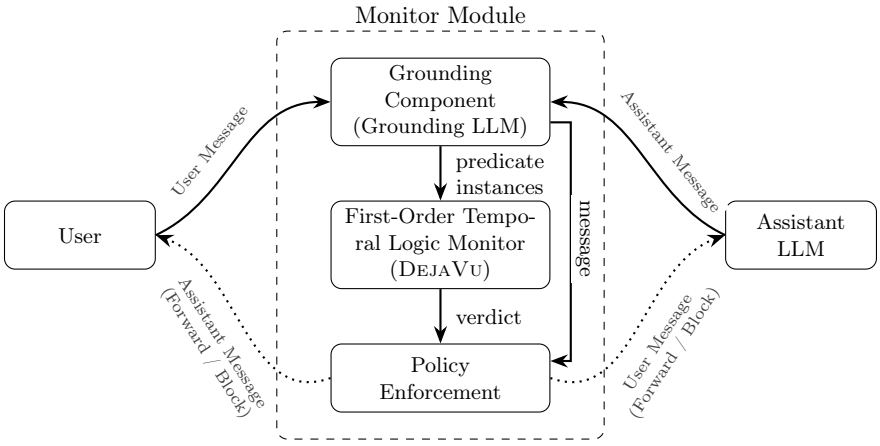

Fig. 1: Conversation monitoring architecture. The module grounds each intercepted message into an event, updates the first-order temporal logic monitor (DejaVu), and forwards or blocks the message per its verdict.

example, a predicate may denote that a user referred to a particular account, product, address, price, or date. The monitor analyzes each message to determine which predicates it satisfies and which objects appear as their arguments.

This analysis produces an object-centered trace of events, where each event may contain multiple predicate instances extracted from the same message. The trace is checked online against safety properties expressed in quantified temporal logic. We call these properties *policies*. Because policies can quantify over objects and relate them across time, they support more fine-grained safeguards than informal or message-local guardrails, including requirements that depend on how specific objects are introduced and reused throughout the conversation.

Applying runtime verification with a first-order temporal logic to conversational systems is motivated by three observations. Our main observation is that many safety requirements in conversational systems are inherently temporal and refer to objects. For example, one may wish to enforce a policy stating that an assistant may disclose information about account x only if the user previously verified ownership of the same x. Although some guardrail frameworks, such as NeMo Guardrails [25], support event-driven conversational flows and custom actions, they are not based on a formal quantified temporal specification language and do not natively provide object tracking across the conversation. Consequently, such policies cannot be expressed in a rigorous and declarative way using these flow-based mechanisms.

Another observation is that training an LLM to detect such undesired temporal policy violations encodes them implicitly in the model parameters. This results in a model that is difficult to interpret, hard to revise after deployment, and susceptible to degradation under adversarial prompting. An external monitoring mechanism provides an additional protection layer that is independent of

the underlying model and can be inspected, modified, and extended more easily as safety requirements evolve.

A further observation is that external monitoring can provide flexibility while preserving formal semantics. Existing runtime-verification approaches for conversational systems typically assume a predefined symbolic event vocabulary, often supplied by the dialogue framework itself. This works when the relevant events are known in advance, but it limits the policies that can be expressed over open-ended LLM conversations. In our setting, a supervisor defines the events to be monitored, together with the objects associated with them. The monitor can therefore build formal event traces from user-defined conversational concepts, rather than being restricted to a fixed set of predefined events.

Applying runtime verification to conversational AI also introduces a fundamental challenge at the level of observation. In classical RV settings, predicates, like *verifies(user,account)*, are typically evaluated over structured system events, e.g. *verifies(John,122173)* or program states, so determining whether a predicate holds is fairly straightforward. In conversational systems, by contrast, the monitor observes natural-language messages produced by the user and the assistant, e.g. "John approves account 122173 witnessed by Linda", and must infer whether it matches the predicate *verifies(user,account)*. Doing so accurately is nontrivial, and in our setting, it must also be efficient enough for online monitoring when conversation data is sensitive.

To address this challenge, we introduce an LLM-based *grounding procedure* that connects natural-language messages and the objects mentioned in them to temporal-logic predicates. Given a message and a predicate schema, the grounding procedure determines whether the message matches the schema and identifies the relevant objects that appear in the message. A central difficulty is *canonicalization*: object mentions must be mapped to consistent representations across different predicates and messages, so that the monitor can determine when two mentions refer to the same underlying object. We evaluate two variations of this grounding procedure. Both use compact LLMs that can run with modest computational resources and support low-latency monitoring. Our experiments show that the grounding procedure is lightweight and practical: it achieves strong accuracy on a single GPU and is fast enough for runtime monitoring.

This work extends our previous propositional framework for conversation monitoring [4] from Boolean dialogue labels to quantified temporal policies over predicates and objects.

Contributions. This paper makes the following contributions. We propose a runtime monitoring approach that treats the interaction between a human and a conversational assistant as an evolving trace, and verifies it against first-order temporal specifications that can track objects consistently across the conversation. Further, we formulate a semantic grounding task that addresses predicate matching, object identification, and object canonicalization over the dialogue trace. We then develop one lightweight grounding approach with two variations, both suitable for the low-latency requirements of runtime verification. Third, we

implement the monitoring approach in DejaVuGuard [3], a tool built on a modified version of DejaVu [15], which allows users to define temporal policies over LLM conversations and monitor their satisfaction as the dialogue unfolds.

Related Work

Ensuring the safety of large language models has received substantial attention. Existing work includes training-time alignment methods, such as *reinforcement learning from human feedback* (RLHF) [23], *constitutional AI* [1], and *direct preference optimization* [24]. It also includes runtime guardrails, such as programmable frameworks including NeMo Guardrails [25], moderation systems for harmful or toxic content [5, 11], and methods for improving robustness to prompt injections and jailbreaks [28]. While these approaches improve practical safety, they typically do not provide a formal object-aware temporal specification layer over open-ended conversations. In particular, prompt injection and multi-turn jailbreak attacks show that safeguards operating at the level of individual prompts or responses may fail when risk emerges gradually across dialogue context [27, 30].

Recent work applied formal specifications and temporal constraints to LLM-based agents, robotics, and conversational analysis. AgentSpec [26], AgentC [22], and RoboGuard [19] consider structured agent events, action sequences, or robotic control traces, rather than open-ended human-agent dialogue grounded from natural language. In conversational settings, LogiDebrief [29] performs post-hoc analysis of emergency-call conversations using STL-based templates, and RvLLM [32] checks LLM outputs against expert-defined domain knowledge, but neither formulates online monitoring over object-centered conversational traces. RV4Chatbot [9, 10] is particularly relevant in that it applies runtime verification to chatbot interactions, but assumes a predefined symbolic event vocabulary provided by the chatbot framework. By contrast, our setting considers open-ended conversations and monitors them online using quantified temporal policies over grounded predicates and tracked objects.

This work extends our earlier conversation-monitoring framework, which modeled an LLM dialogue as a Boolean-labeled trace and verified it against propositional past-time temporal policies [4]. In that setting, grounding determined only which atomic propositions held at each step. Here, we move to quantified temporal logic and must ground not only predicates, but also their associated objects, while maintaining consistent object canonicalization across the conversation.

2 First-Order Past-Time Temporal Logic

In runtime verification, a monitor observes an execution trace and checks compliance with a formal specification at each step. We model an LLM conversation as a finite trace of messages and specify safety policies as formulas in the first-order past-time temporal logic Qtl (Quantified Temporal Logic) [6, 15]. Qtl

allows policies to quantify over objects and relate them across time, enabling fine-grained specifications that track how specific objects, such as accounts, documents or tickets, relate throughout the conversation. QTL is well-suited for conversational monitoring since formulas refer only to the present and past, so the truth value is determined at each step with conclusive verdicts.

2.1 The DejaVu Tool

DejaVu [6, 15] is a runtime verification system implemented in Scala for monitoring event streams against QTL specifications. It synthesizes a monitor in Scala from a given specification, compiles it, and executes it against the trace, maintaining a two-vector structure (pre and now) storing BDDs [12, 13]. Data-rich events over unbounded domains are handled through an enumeration strategy: values are assigned binary encodings (as in binary numbers) on first occurrence, which are then represented as BDDs, enabling verification over infinite domains with finite memory. DejaVu supports both online and offline monitoring.

2.2 Syntax and Semantics

The logic QTL supported by DejaVu allows expressing properties of executions that include data. The restriction to past time permits interpreting formulas on finite traces.

Syntax. The formulas of QTL are defined by the following grammar, where p is a *predicate* symbol, a is a *constant*, and x is a *variable*. For simplicity, we present QTL with unary predicates; this is not a principal restriction, and DejaVu fully supports predicates over multiple arguments, including zero-argument propositions.

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

A formula can be interpreted over multiple types (domains), e.g., natural numbers or strings. Accordingly, each variable, constant, and predicate parameter has a specific type. Type matching is enforced: for $p(a)$ (respectively $p(x)$), the type of the parameter of p and the type of a (respectively x) must coincide. Propositional past-time LTL is the special case where all predicates are parameterless Boolean propositions.

Semantics. A QTL formula is interpreted over a *trace*, a finite sequence of *events*, where each event is a finite set of predicate instances (e.g., $\{p(a), q(7)\}$). This generalizes the standard DejaVu semantics, in which each event is a single predicate instance, and is needed in our setting because a single conversational message may give rise to several grounded predicate instances; we implemented this extension in DejaVu for the present work.

The informal meaning of the key QTL operators is as follows: $p(a)$ holds iff the last event contains $p(a)$; $p(x)$ holds when x is bound to a and $p(a)$ is in the last event; $(\varphi \mathcal{S} \psi)$ holds iff ψ occurred at some past position and φ has held continuously since; $\ominus\varphi$ holds iff φ held at the previous position; and $\exists x\ \varphi$ holds

iff some value a makes $\varphi[x \mapsto a]$ true. Derived operators include: $\mathit{false} = \neg \mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondsuit\, \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$, $\boxminus \varphi = \neg \diamondsuit \neg\varphi$, and $\forall x\, \varphi = \neg\exists x\, \neg\varphi$. We write $(\gamma, \sigma, i) \models \eta$ to mean that formula η holds at position i of trace σ under an assignment γ to the free variables of η; the full satisfaction relation is given in [15].

2.3 RV Monitoring of First-Order Past-Time Temporal Logic

The monitoring algorithm for first-order past-time temporal logic is implemented in DejaVu [15]; we only sketch it here and refer to [15] for the full algorithm. Each subformula η is associated with the set of variable assignments that satisfy it, where an assignment maps the free variables of η to values of their types. The monitor keeps two such sets per subformula, $\mathsf{pre}(\eta)$ and $\mathsf{now}(\eta)$, holding the satisfying assignments over the trace *excluding* and *including* the last event, respectively. On each new event, these sets are recomputed bottom-up over the subformulas, using relational counterparts of the Boolean connectives (complementation, join, and co-join) together with projection for the quantifier; the $\mathcal{S}$ operator depends on both now and pre, whereas $\ominus$ depends only on pre.

To represent assignments over unbounded domains in finite memory, DejaVu encodes values as bitstrings. Each $\mathsf{now}(\eta)$ is represented as a BDD [12,13] over the free variables of η, see [15] for details. In DejaVu syntax, forall and exists denote $\forall$ and $\exists$. The temporal operators P, H, S and @ denote $\diamondsuit$, $\boxminus$, $\mathcal{S}$, and $\ominus$, respectively. In addition, |, &, and ! denote $\vee$, $\wedge$, and $\neg$, respectively.

3 Conversation Monitoring

3.1 Monitoring Approach

Our goal is to monitor an ongoing conversation between a human *user* and an LLM-based *assistant* while preserving the identities of objects referred to across different messages. Throughout the paper, we use the term *assistant* for the LLM-based system participating in the monitored conversation. We reserve the term *user* for the human participant who interacts with the assistant. We use the term *supervisor* for the person who configures the monitor before monitoring takes place, by defining policies and predicates. We illustrate the setting through a running example, based on the account-recovery case presented in the introduction. The monitored conversation in this example appears in Figure 2.

The intended policy for the conversation example requires that a user may request a password reset for an account only after the assistant has verified ownership of that account. In our approach, such policies are expressed using Qtl. For the running example, the policy is

$$\forall a.\, \mathsf{reset\_request}(a) \rightarrow \diamondsuit\, \mathsf{verify\_success}(a)$$

where a refers to the account.

Message. *user*: "I've lost access to `john.doe@example.com` and need to recover it."
DejaVu event. $\emptyset$

Message. *assistant*: "I've verified ownership of `john.doe@example.com` using the code sent to your phone."
DejaVu event. {verify_success(`john.doe@example.com`)}

Message. *user*: "Please reset the password for `john.doe`."
DejaVu event. {reset_request(`john.doe@example.com`)}

Fig. 2: Translating conversation messages into the event representation monitored by DejaVu.

In order to use an RV tool to monitor the conversation, in our case DejaVu, these natural-language messages must be translated into events that the tool can process. Figure 2 shows this translation for the account-recovery example. Each event contains the predicate instances extracted from the corresponding message[3].

We call the message-to-event translation step *grounding*. In this work, grounding is performed by a separate *grounding LLM*, distinct from the *assistant LLM* that participates in the conversation. This translation is the central technical step in our approach: it connects natural-language conversation messages to the first-order events monitored by DejaVu.

We now introduce some formal definitions needed to describe the grounding process.

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

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*.*

A conversation alternates between user and assistant messages. After observing message m_i, the monitor has accessed only the current prefix $\sigma_{\leq i}$ and must update its verdict incrementally. This is consistent with the standard runtime verification setting [18], where a monitor observes events one at a time as they occur and evaluates the specification incrementally. The conversation in Figure 2 consists of three messages.

Predicate schemas determine the predicate symbols and their arguments used in QTL formulas that define policies. These schemas are defined up front by the supervisor, before monitoring takes place. To support grounding, the supervisor

[3] The first message does not produce any predicate instance; it is included only to initiate the conversation between the user and the assistant LLM.

defines a textual criterion associated with each predicate that appears in a policy, which is a natural language explanation of the event and its arguments. This criterion may include references to objects that should be extracted from the message, such as people, products, prices, locations, accounts, or dates.

Definition 3 (Predicate Schema). *A* predicate schema *is a tuple*

$$p = (id_p, \delta_p, role_p, R_p)$$

where id_p is the predicate identifier, δ_p is a natural-language description of the criterion represented by the schema, $role_p \in \mathcal{R}$ identifies the sender, and $R_p = (r_1, \ldots, r_k)$ is an ordered sequence of argument slots*, where k is the arity of the predicate. An argument slot is one object position in the predicate; each slot has an identifier and a natural-language description of the object that should fill this position when the schema matches a message.*

Including $role_p$ as a parameter in the definition of a predicate schema was shown experimentally to improve accuracy.

In our example from Figure 2, we have two predicate schemas, p_1 and p_2. The predicate schema p_1 has id verify_success, description "the assistant successfully verified the user as the owner of the account", role assistant and a single argument slot. The predicate schema p_2 has id reset_request, description "the user requests a password reset for an account", role user, and again a single argument slot. We call an occurrence of an argument slot in a message an *object*. In our example, one such object is `john.doe@example.com`. The above predicate schemas allow grounding verify_success(`john.doe@example.com`) from the second message in Figure 2 and reset_request(`john.doe@example.com`) from the third message.

Definition 4 (Policy). *A* policy *is a* QTL *formula whose predicate symbols are the identifiers of predicate schemas. For a predicate schema p with argument slots $R_p = (r_1, \ldots, r_k)$, an occurrence of p in a policy has the form $id_p(t_1, \ldots, t_k)$, where each term t_j is a variable or constant, and each argument position corresponds to the argument slot r_j.*

Canonicalization of argument slots. Continuing the account-recovery example, the predicate schemas are verify_success and reset_request, both with an argument slot for the account identifier. The policy $\forall a.\, (\mathsf{reset\_request}(a) \rightarrow \Diamond\, \mathsf{verify\_success}(a))$ requires that every password-reset request be preceded by a successful verification of the same account. In this formula, the variable a occupies the argument position corresponding to the argument slot defined for accounts in both predicate instances. Since the same variable a appears in both predicate instances, the grounding procedure must interpret the two argument slots consistently across the conversation. In natural language, however, the same email address can be written in different ways. For example, a message may mention "john.doe@example.com", "john.doe at example", or, when the surrounding context is clear, simply "john.doe". The monitor must therefore determine whether the extracted objects refer to the same actual account. More generally,

two argument slots from different predicate schemas appearing in the same policy are considered *related* when they are bound to the same quantified variable in that policy formula.

To compare related argument slots across messages, each extracted object is associated with a *canonical form*: a stable representation of the object value used by the runtime monitor component. This process is called *canonicalization*. In the example shown in Figure 2, the occurrences "john.doe@example.com" and "john.doe" are associated with the same canonical form by the grounding procedure, since the latter refers to the same account in the conversation context. When an account is first observed, the grounding procedure may introduce a new canonical form, initialized from that first observed mention. When a later account mention refers to an account previously observed in a related argument slot, it reuses the existing canonical form. To support canonicalization, the grounding procedure uses a data structure that stores information about related argument slots and the objects previously associated with them.

Definition 5 (Related-Object Summary). *For a predicate schema p, the* related-object summary*, denoted by S_p, stores for each argument slot r, pairs of values of the form $\langle v, c \rangle$. Each pair records an* assignment *of a textual mention v to a canonical form c, where the assignment was produced either for r itself or for one of the related argument slots of r from different predicate schemas.*

This information is used to interpret objects extracted for argument slots of p consistently across the conversation, by helping the grounding procedure map each object to the correct canonical form.

Consider now the two predicate schemas from our running example in Figure 2. The *account* argument slot of reset_request is related to the *account* argument slot of verify_success, because both appear as the same variable a in the policy above. Before the second message is observed, the related-object summary of both predicate schemas is empty, since no object has been extracted for their argument slots. After the second message is observed, both related-object summaries are updated with the pair $\langle v =$ `"john.doe@example.com"`, $c =$ `"john.doe@example.com"`$\rangle$. When the third message is processed, the grounding procedure recognizes that the object `john.doe` refers to the same account and therefore reuses `john.doe@example.com` from the related-object summary of *reset_request* as the canonical form of `john.doe`. Both related-object summaries are then updated with the pair $\langle v =$ `"john.doe"`, $c =$ `"john.doe@example.com"`$\rangle$. Note that, in general, the summary keeps all the textual mentions and corresponding canonical forms created so far for an argument slot. This allows the grounding procedure to reuse the most suitable canonical form for the next assignment associated with that slot, or to introduce a new canonical form when none is suitable.

Definition 6 (Grounding Function). *Let $p = (id_p, \delta_p, role_p, R_p)$ be a predicate schema with k argument slots, $R_p = (r_1, \ldots, r_k)$, and let $m_i = (role_i, text_i)$ be the i^{th} message in the conversation. The grounding function* ground(m_i, p, S_p) *takes the message, the predicate schema p, and its related-object summary S_p,*

and returns a set $A_{p,i}$ of tuples of the form $(c_1, \ldots, c_k)$. Each such tuple is called an object assignment*, where each element c_j is the canonical form of the object extracted for argument slot r_j.*

A single message can yield multiple object assignments for the same predicate schema. For example, in the account-recovery setting, a predicate schema $\mathsf{reset\_request}(a)$ may match a user message that asks to reset access for two different accounts, for instance "Please reset the password for both `john.doe` and `john.doe1`". To support such cases, the grounding output is treated as a *finite set $A_{p,i}$* of object assignments. This set is empty when the message does not match the predicate schema p, or when $role_i \neq role_p$.

Definition 7 (Message Abstraction). *Let $\mathcal{P}$ denote the set of supervisor-defined predicate schemas. The* event abstraction *of a message m_i is*

$$\lambda(m_i) = \{id_p(c_1, \ldots, c_k) \mid p \in \mathcal{P} \ \wedge \ (c_1, \ldots, c_k) \in A_{p,i}\}$$

To construct this abstraction, the grounding procedure is applied to the text separately with respect to each predicate schema in $\mathcal{P}$. Note that the arguments of the predicate instances are canonical forms. As a result, two different objects that are assigned the same canonical form are presented to the runtime monitor component as the same entity. In Figure 2, the corresponding event appears below each message.

Monitoring with DejaVu. We now summarize how the pieces above are combined in the monitoring procedure. Given a policy, we first compute for each predicate schema p its related argument slots (determined by the policy), so that its related-object summary can be updated during monitoring. When a new message m_i arrives, the module first selects the predicate schemas whose role matches $role_i$. For each such schema p, it applies the grounding function $\mathsf{ground}(m_i, p, S_p)$ using the current summary. The returned object assignments are used to construct the event $\lambda(m_i)$, and the related-object summary of each predicate schema is updated with the newly extracted objects and their canonical forms. The resulting event is sent to DejaVu, which incrementally evaluates the given policy over the event trace. Our implementation required extending DejaVu so that a single event can contain multiple object assignments for the same predicate schema, which was not supported before.

If after processing the message m_i the monitor has found a violation, the monitoring module blocks the message and reports the violation, following the runtime enforcement perspective [7]. Otherwise, a permitted user message is passed to the assistant, and a permitted assistant response is returned to the user. In this way, our procedure enforces both sides of the interaction while remaining independent of the specific LLM used as the conversational assistant.

We implement the approach in DejaVuGuard [3], a system that uses DejaVu as its runtime monitoring component. The system provides a chat-style web interface for interacting with a wide variety of LLMs, while allowing the supervisor to define and manage the policies used by the monitor.

3.2 Conversation Policy Examples

We next give two examples of policies expressible in our framework. These examples illustrate how predicate schemas, argument slots, and temporal operators are combined to define monitorable conversation requirements.

Example 1: Verified account recovery. Here, we extend the account-recovery example used above. In addition to password-reset requests, the extended policy also covers requests to add an associated email address to an account: either action is permitted only after the assistant has verified recovery authorization for the same account. Accounts are identified by their email addresses.

The predicate schemas verify_success and reset_request are the ones introduced above. We add a third schema, add_email, with role user, describing that the user requests adding an associated email address to an account. It has two argument slots: one for the account email address and one for the email address to add.

The temporal property is

$$\forall a\, \forall e.\, \Big((\mathsf{add\_email}(a, e) \rightarrow \Diamond\!\!-\, \mathsf{verify\_success}(a)) \\ \wedge\, (\mathsf{reset\_request}(a) \rightarrow \Diamond\!\!-\, \mathsf{verify\_success}(a)) \Big).$$

That is, every password-reset or add-email request must be linked to a prior successful verification of the same account email address.

Example 2: Balance-constrained bank transfer. Consider a user interacting with a banking assistant. The policy requires two conditions. First, if the banking assistant has reported that the account balance is b dollars, then any later transfer requested by the user must be for an amount no greater than b. Second, immediately after the user requests a transfer, the banking assistant must report the updated account balance. To distinguish user events from assistant events in the policy, DejaVuGuard supports the built-in proposition user_turn, which holds exactly on events produced from user messages. The supervisor defines two predicate schemas here. The first schema has the identifier bal, the role assistant, and the description "the banking assistant reports the current balance of the account". It has one argument slot, representing the account balance amount reported by the banking assistant, measured in USD. The second schema has the identifier tx, the role user, and the description "the user requests a transfer to another account for a specific amount". It has one argument slot, representing the amount, measured in USD, that the user attempts to transfer.

The temporal property is

$$\Big(\mathsf{user\_turn} \rightarrow \Big(\forall t\, .\, \mathsf{tx}(t) \rightarrow \exists b\, .\, \Big(\big(\neg \exists b_2\, .\, \mathsf{bal}(b_2) \big)\, \mathcal{S}\, \mathsf{bal}(b) \Big) \wedge \neg (b < t) \Big) \Big) \\ \wedge \big(\neg \mathsf{user\_turn} \rightarrow ((\ominus \exists t\, .\, \mathsf{tx}(t)) \rightarrow \exists b\, .\, \mathsf{bal}(b)) \big).$$

The first conjunct applies on user turns and requires every transfer amount to be bounded by the most recently reported account balance. The second conjunct

applies on assistant turns and requires the banking assistant to report an account balance immediately after a transfer request.

4 Implementation of Grounding

In our earlier conversation-monitoring framework [4], monitored criteria were expressed as role-constrained natural-language *patterns* associated with propositional events. Grounding was therefore a task with Boolean output: given a message and such a pattern, the grounding function decided whether the proposition should appear in the event. This allowed us to experiment with several approaches, including embedding-based similarity, an approach based on natural language inference (NLI), and an LLM-based approach. The LLM-based approach worked best. We therefore use an LLM-based implementation as the basis for the grounding function in the present work. The grounding LLM is separate from the assistant LLM whose conversation is being monitored. Here, however, the task is richer: the model must decide whether a message matches a supervisor-defined predicate schema, extract the objects corresponding to its argument slots, and assign canonical forms for these objects.

Our approach uses a prompt that provides the grounding LLM with all inputs required by the grounding function: the message m_i, the predicate schema p, and the current related-object summary S_p. The prompt instructs the grounding LLM to return a structured JSON object. If the message does not match the schema, the output indicates that no match was found. Otherwise, the output contains one entry for each match, including the extracted object mentions and their canonical forms. The monitor then uses the canonical-form tuples as object assignments $A_{p,i}$, which are used to construct the event abstraction $\lambda(m_i)$.

The grounding procedure is designed for compact, locally deployable LLMs. This is important because grounding may involve sensitive conversation content, and local deployment avoids sending the monitored messages to an external provider. Since compact models are sensitive to prompt design, we refined the prompt used by the grounding procedure through an optimization process whose empirical effect is reported in Section 5.

The procedure consists of an offline preparation phase and an online grounding phase. In the offline phase, we generate a small set of schema-specific few-shot demonstrations that show successful grounding for the predicate schema. These demonstrations are included in the prompt to improve the likelihood that the grounding LLM correctly matches the schema and extracts the corresponding objects. Since demonstration generation is performed offline and only once per schema, it can use stronger and potentially slower LLMs to produce diverse and high-quality examples.

In the online phase, the monitor processes the conversation one message at a time. When a new message m_i arrives, the monitor selects the predicate schemas whose role matches $role_i$. For each such schema p, it inserts m_i, p, the current related-object summary S_p, and the few-shot demonstrations into the grounding prompt. We use separate prompt templates for user and assistant messages,

so that the same content can be interpreted relative to the sender's role. The prompts are available in the DejaVuGuard repository [3]. To mitigate prompt-injection risk, the prompt explicitly separates the inserted conversation content from the grounding instructions and instructs the model to treat it as untrusted data.

Fine-tuning. In addition to using compact pretrained LLMs directly through prompting, we also experimented with fine-tuning compact LLMs for the grounding task. Fine-tuning means further training an existing model on task-specific examples, thereby modifying its internal parameters, so that the model adapts its behavior to the target task. The goal was to test whether this task-specific training improves grounding accuracy. For this experiment, we created a few thousand input–output pairs for the grounding LLM using a synthetic data-generation process. Each input is a complete grounding task: it includes a message, a predicate schema, the relevant related-object summary and the same grounding instructions used in the prompting approach. Each target output is the expected structured JSON result for that task. We fine-tuned the models using LoRA [16]. After fine-tuning, inference is performed as before: the model receives the grounding prompt and produces the same structured JSON output.

5 Experiments

We evaluate our approach using two sets of experiments. The first set assesses the accuracy of the grounding procedure against a dedicated grounding dataset; given a single message and a predicate schema, we measure whether the grounding procedure extracts the correct predicate instances, object mentions, and canonical forms (Section 5.1). The second set evaluates the end-to-end monitoring approach using DejaVuGuard; replaying full conversations through the monitor, we measure whether it produces the correct policy verdict at every conversational step (Section 5.2).

5.1 Grounding Experiments

We evaluated our proposed grounding procedure in terms of its accuracy in matching messages to predicate schemas and extracting the corresponding objects. Specifically, we evaluated the two variations of the procedure described in Section 4: prompting pretrained compact LLMs with grounding instructions, and fine-tuning compact LLMs for the same grounding task. All experiments in this subsection were conducted on a single NVIDIA RTX 4090 GPU.

Test Dataset. We evaluated grounding on a synthetic dataset of 1,295 labeled messages spanning 163 distinct predicate schemas from 19 domains, where most messages and schemas are LLM-generated. Each record contains a natural-language user or assistant message, a predicate schema with argument slots, an optional related-object summary, and a structured ground-truth output. The

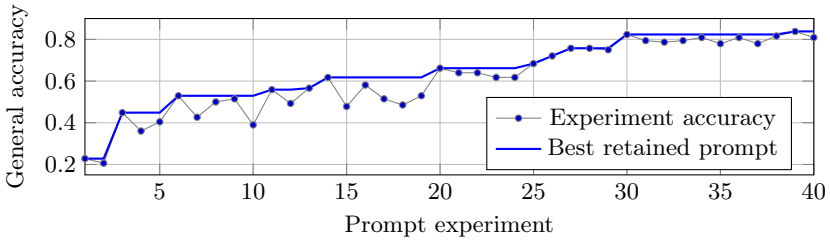

Fig. 3: Automatic prompt optimization for Gemma-3-4B: degrading edits are discarded, improvements retained.

output specifies whether the schema matches the message and, when it does, the expected object assignments. The dataset includes both positive and negative examples, single-match and multi-match messages, and cases requiring non-trivial canonicalization across related argument slots. Negative examples were constructed adversarially: they intentionally include terminology closely related to the predicate schema domain.

Prompt optimization. We optimized the grounding prompt using a fully automatic *Claude Code* workflow inspired by the auto-research approach [17]. This optimization used a 646-record calibration dataset that is completely disjoint from the 1,295-record test dataset and covers 15 of the 19 domains. Starting from a manually written baseline prompt, the workflow automatically edited the prompt, evaluated it on the calibration dataset using `Gemma-3-4B`, inspected the grounding errors, and retained only changes that improved accuracy. It was instructed to prefer general improvements over example-specific fixes. After 48 iterations, the retained prompt improved from 0.23 to 0.84 general grounding accuracy, with the last improvement at iteration 40 (Figure 3).

We used the resulting prompts, with separate versions for user and assistant messages, for the model comparison below. In the first variation, the models are used as pretrained LLMs, without additional training. We evaluated four families of compact LLMs at two sizes each. The number of parameters, in billions, appears in each model name (Table 1). *General* accuracy counts a record as correct only if the match decision, all object assignments, all object mentions, and all required canonical forms are correct. *Match* accuracy measures whether the model correctly decides if the message matches the given predicate schema. *Mention* accuracy measures object-mention extraction, and *Canon.* accuracy measures canonical-form correctness for objects that require comparison with the related-object summary.

`Qwen3.5-9B`, `Qwen3.5-4B`, and `Ministral-3-8B` all exceeded 0.95 general accuracy. Parameter count alone is not decisive: `Gemma-3-12B` performed worse than the much smaller `Qwen3.5-4B`, mainly because of false-positive matches. Canonicalization from history remained the hardest subtask, although the strongest models achieved high accuracy on it.

Model	Gen. Acc.	Match Acc.	Mention Acc.	Canon. Acc.	Avg. Latency (s)
`Qwen3.5-4B`	0.9537	0.9761	0.9438	0.9362	1.521
`Qwen3.5-9B`	0.9629	0.9799	0.9514	0.9558	4.624
`Ministral-3-3B`	0.9097	0.9653	0.8825	0.8642	0.852
`Ministral-3-8B`	0.9514	0.9776	0.9532	0.9689	1.816
`Llama-3.2-3B`	0.6911	0.8170	0.7794	0.7136	0.995
`Llama-3.1-8B`	0.8919	0.9429	0.9421	0.8822	2.285
`Gemma-3-4B`	0.7668	0.8517	0.9020	0.8478	2.037
`Gemma-3-12B`	0.8888	0.9189	0.9591	0.9574	5.175

Table 1: Prompting-based grounding accuracies and average latency per grounding call.

Per-predicate grounding latency ranged from 0.85 s (`Ministral-3-3B`) to over 5 s (`Gemma-3-12B`); `Qwen3.5-4B` offered the best accuracy and latency balance at 1.52 s. Since grounding may be applied to several predicate schemas for the same message, these calls can be executed concurrently rather than sequentially. On the tested GPU, we ran 16 grounding tasks concurrently without a throughput collapse, so the added delay is closer to the longest call in the concurrent batch. This suggests that the approach remains practical when monitoring multiple policies, each with its own predicate schemas.

Fine-tuning. We also evaluated the fine-tuning variation of the grounding procedure. For this experiment, we used an LLM to generate 5,000 input–output pairs for the grounding LLM, demonstrating successful execution of the grounding task for messages and predicate schemas. Each input follows the same grounding format used in the prompting variation, and each output is the expected structured JSON result.

We fine-tuned models from three families. Due to resource limitations, we used two models from each family, but replaced the larger model from the prompting comparison with the next smaller model in the same family. We omitted `Ministral` for technical reasons, since it did not fit the fine-tuning pipeline we used. The models were fine-tuned with LoRA [16].

Fine-tuning improved every model in Table 2. The gains are inversely related to the capability of the corresponding base model: stronger models close to saturation gain little, while weaker models gain substantially more. These results indicate that the grounding task can be transferred into small models through task-specific training, although we treated this as a proof of concept rather than the main deployment configuration.

5.2 End-to-End Monitoring Experiments

This second set of experiments evaluates the complete DejaVuGuard pipeline by replaying entire conversations and recording the monitor's verdict after each message. A scenario counts as correct only if every verdict matches the ground

Model	Pretrained	Fine-tuned	Δ
`Qwen3.5-2B`	86.6%	92.5%	+5.9 pp
`Qwen3.5-4B`	95.3%	96.1%	+0.8 pp
`Gemma-3-1B`	20.2%	63.2%	+43.0 pp
`Gemma-3-4B`	76.6%	88.2%	+11.6 pp
`Llama-3.2-1B`	45.0%	77.2%	+32.2 pp
`Llama-3.2-3B`	69.1%	90.1%	+21.0 pp

Table 2: General accuracy before and after fine-tuning; Δ is the improvement in percentage points.

		DejaVuGuard				LLM Baseline	
		Sonnet-4.6		Ministral-3-8B		Sonnet-4.6	Ministral-3-8B
Scenario	Msgs	Scen.	Msg.	Scen.	Msg.	Scen.	Scen.
Account recovery	888	99/100	99.9%	86/100	98.4%	99/100	84/100
Allergy recipe	498	93/100	98.2%	89/100	97.6%	95/100	78/100
Bank transfer	825	100/100	100.0%	94/100	99.2%	97/100	84/100
Medical record	713	99/100	99.9%	93/100	98.7%	99/100	95/100
Total	2924	**391/400**	**99.6%**	**362/400**	**98.6%**	**390/400**	**341/400**

Table 3: End-to-end monitoring results. For DejaVuGuard, *Scen.* gives the number of conversations whose monitor verdict is correct after every message out of 100, and *Msg.* gives the fraction of individual message positions with a correct verdict. The LLM baseline evaluates each complete conversation directly and therefore has only scenario-level results.

truth verdict. We tested four policies: verified account recovery and balance-constrained bank transfer, described in Section 3.2, together with allergy-aware recipe generation and medical-record access. Full policy definitions and implementation details are available in the DejaVuGuard repository [3].

The benchmark comprises 400 challenging AI-authored, human-validated conversations (100 per policy, evenly split between violating and compliant), spanning 3 to 20 messages each for 2,924 messages in total. Here, we used the first grounding variation, in which pretrained models are prompted for the grounding task. We evaluated `Ministral-3-8B`, as a compact LLM suitable for local deployment, and `Claude Sonnet-4.6`, to evaluate how well our approach performs with a frontier-scale LLM. As a baseline, we prompted the same two LLMs with the full conversation, the policy in formal and natural language, and the predicate-schema and argument-slot descriptions, and asked whether the conversation satisfies the policy. The fine-tuned models are excluded here, as scaling them to full conversations required additional resources.

As Table 3 shows, with `Claude Sonnet-4.6`, DejaVuGuard and the direct baseline are nearly tied. With `Ministral-3-8B`, DejaVuGuard clearly im-

proves over the baseline. The message-level and scenario-level columns measure different requirements: message-level accuracy counts individual verdict positions, whereas scenario accuracy requires every verdict in a conversation to be correct, so sparse message-level errors can reduce the scenario score.

The `Ministral-3-8B` failures follow two main patterns. The first is canonicalization, particularly when the canonical form is textually distant from the object mention, or conversely, when the model over-canonicalizes and collapses two distinct objects into one. The second appears in messages with several object assignments for the same schema; as the number of object assignments in one message grows, the model is more likely to miss an object or produce an incorrect canonical form.

These results suggest that our approach helps compact models, while for frontier-scale models the relatively short conversations in this benchmark do not show a clear advantage over the direct baseline. Since DejaVuGuard evaluates policies incrementally over event traces, its advantage over a frontier-scale direct baseline may become more visible for longer conversations or more complex temporal policies.

6 Conclusion

We introduced a first-order runtime-verification approach for monitoring LLM conversations. In our approach, a conversation is treated as a trace that can be checked online against QTL policies. By grounding user and assistant messages into symbolic events, the monitor can enforce temporal requirements over the conversation, while tracking objects mentioned across messages. We implemented the approach in DejaVuGuard and illustrated it with policies from multiple domains.

Our approach supports flexible predicate schemas defined by the supervisor, rather than relying on a fixed event vocabulary, while still enabling formal temporal reasoning over objects that recur across the conversation. This is made possible by the grounding procedure, which connects natural-language messages to predicate instances and canonical object representations. Our experiments show that this grounding can be performed accurately and with latency low enough for practical monitoring. Using compact models also makes on-premise deployment more feasible, which is important for settings with security and privacy constraints. The reported latencies suggest that, in future work, the grounding procedure could be extended toward monitoring voice conversations with a limited number of relevant predicate schemas.

The main bottleneck remains grounding quality in compact LLMs, especially for canonicalization and messages with multiple object assignments. We expect this limitation to diminish as the LLMs used for grounding improve. Future work could support predicate schemas whose evidence is distributed across multiple messages, allowing the monitor to better capture intent expressed gradually over a conversation. Another direction is to improve canonicalization in cases where object identity depends on broader conversational context.

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. Cohen, I., Havelund, K., Omer, M., Peled, D.: Monitoring LLM conversations with first-order temporal logic. https://github.com/itay99988/first_order_rv_guardrails (2026), source code, datasets, and experiment logs
4. Cohen, I., Havelund, K., Omer, M., Peled, D.: Temporal guardrails for LLM conversations: A runtime verification framework. In: Avni, G., Schilling, C. (eds.) AI Verification - Third International Symposium, SAIV 2026, Lisbon, Portugal, July 24-25, 2026, Proceedings. Lecture Notes in Computer Science, vol. 16831, pp. 145–166. Springer (2026), https://doi.org/10.1007/978-3-032-32357-6_7
5. Davidson, T., Warmsley, D., Macy, M., Weber, I.: Automated hate speech detection and the problem of offensive language. In: ICWSM (2017)
6. DejaVu tool source code. https://github.com/havelund/dejavu
7. 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)
8. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D., Kalus, G. (eds.) Engineering Dependable Software Systems. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)
9. 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
10. Gatti, A., Mascardi, V., Ferrando, A.: RV4Chatbot: Are chatbots allowed to dream of electric sheep? arXiv preprint arXiv:2411.14368 (2024)
11. Google Jigsaw: Perspective API. https://perspectiveapi.com (2017)
12. Havelund, K., Peled, D.: BDDs on the run. In: Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018). pp. 58–69 (2018)
13. Havelund, K., Peled, D.: BDDs for representing data in runtime verification. In: Runtime Verification (RV 2020). pp. 107–128 (2020)
14. Havelund, K., Goldberg, A.: Verify your runs. In: Verified Software: Theories, Tools, Experiments, VSTTE 2005. pp. 374–383 (2008). https://doi.org/http://dx.doi.org/10.1007/978-3-540-69149-5_40
15. 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)
16. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: Low-rank adaptation of large language models. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net (2022)
17. Karpathy, A.: autoresearch: AI agents running research on single-GPU nanochat training automatically (2026), https://github.com/karpathy/autoresearch, gitHub repository

18. 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
19. 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
20. Malwarebytes Labs: Meta's AI support bot happily handed instagram accounts to hackers. Malwarebytes Labs Blog (Jun 2026), https://www.malwarebytes.com/blog/ai/2026/06/metas-ai-support-bot-happily-handed-instagram-accounts-to-hackers
21. Meta: AI support assistant. Meta Account Recovery Support (2026), https://www.meta.com/account-recovery-support/ai-support-assistant/
22. 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
23. Ouyang, L., et al.: Training language models to follow instructions with human feedback. In: NeurIPS (2022). https://doi.org/10.48550/arXiv.2203.02155
24. 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)
25. 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)
26. 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
27. Russinovich, M., Salem, A., Eldan, R.: Great, now write an article about that: The crescendo multi-turn LLM jailbreak attack. In: Bauer, L., Pellegrino, G. (eds.) 34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025. pp. 2421–2440. USENIX Association (2025)
28. 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)
29. 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
30. Weng, Z., Jin, X., Jia, J., Zhang, X.: Foot-In-The-Door: A multi-turn jailbreak for LLMs. In: Christodoulopoulos, C., Chakraborty, T., Rose, C., Peng, V. (eds.) Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025. pp. 1939–1950. Association for Computational Linguistics (2025)
31. Yi, J., Xie, Y., Zhu, B., Kiciman, E., Sun, G., Xie, X., Wu, F.: Benchmarking and defending against indirect prompt injection attacks on large language models. In: Sun, Y., Chierichetti, F., Lauw, H.W., Perlich, C., Tok, W.H., Tomkins, A. (eds.) Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, V.1, KDD 2025, Toronto, ON, Canada, August 3-7, 2025. pp. 1809–1820. ACM (2025)
32. Zhang, Y., Emma, S.Y., En, A.L.J., Dong, J.S.: RvLLM: LLM runtime verification with domain knowledge. CoRR **abs/2505.18585** (2025)