RefineAct: Automatic Runtime Verification of LLM Agent Actions

Fraol Batole
Tulane University
New Orleans, Louisiana, USA
fbatole@tulane.edu

Foutse Khomh
Polytechnique Montréal
Montréal, Canada
foutse.khomh@polymtl.ca

Hridesh Rajan
Tulane University
New Orleans, Louisiana, USA
hrajan@tulane.edu

ABSTRACT

LLM-based agents that invoke external tools can cause irreversible harm when their actions diverge from user intent, from deleting critical files to exposing private data or executing untrusted code. Unlike traditional software, where test suites and contracts define expected behavior, LLM agents operate without a specification: there is no artifact against which to check their actions before execution. Existing safeguards rely on prompt engineering or post-hoc evaluation, neither of which closes this gap. We present RefineAct, a runtime verification framework that automatically derives a task-specific specification from the user's natural-language instruction and enforces it during agent execution. RefineAct operates in three stages: (1) *intent formalization* translates instructions into first-order Prolog predicates capturing the user's goal and safety requirements; (2) *refinement-based planning* decomposes the specification into a concrete action plan with pre-/postcondition chains and risky-action patterns; and (3) *runtime verification* intercepts each agent-proposed action, queries the Prolog knowledge base to verify precondition satisfaction and constraint compliance, and returns a disposition state that determines whether the action proceeds, requires user confirmation, or is blocked with corrective feedback. We evaluate RefineAct on 144 agent tasks across five domains from the ToolEmu benchmark. RefineAct reduces failure incidence from 77% to 39% while improving task completion quality from 1.0 to 1.9 on a 0–3 scale, demonstrating that deriving and enforcing specifications at runtime can improve both safety and helpfulness of LLM agent behavior.

KEYWORDS

LLM, LLM-based Agents, Verification

ACM Reference Format:
Fraol Batole, Foutse Khomh, and Hridesh Rajan. 2026. RefineAct: Automatic Runtime Verification of LLM Agent Actions. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software engineering has long recognized that verification requires either an implicit or an explict understanding of the expected behavior, i.e. specifications. Unit tests encode expected behavior [2, 7]. Design-by-contract annotates methods with preconditions and postconditions [1, 11]. Model checking exhausts state spaces against temporal properties [5, 9, 13]. In each case, the specification is the anchor: without it, there is nothing to verify against.

In recent years, LLM agents have evolved from conversational assistants into autonomous systems that manage files, send emails, and interact with external APIs [4, 19, 20]. These agents operate without an explicit specification. For example, a user says "clean up my disk" and the agent interprets, plans, and executes, all without an explicit specification of what "clean up" means or what actions are out of bounds. Specifically, there is no artifact against which to check the agent's behavior before execution. As these agents evolve from conversational assistants into autonomous systems [4, 19, 20], the consequences of this gap extend beyond text generation into irreversible real-world effects [15].

Recent incidents underscore the practical cost. In a widely reported case, an AI coding agent deleted a company's entire codebase and production data, despite the user explicitly requesting no changes without permission [8]. The failure was not that the agent lacked capability, but that nothing checked whether its actions matched the user's intent before execution. Existing safety approaches fall into three categories: prompt-level safety instructions, post-hoc trajectory evaluation [15], and manually authored runtime rules in a domain-specific language [18]. Prompt-based approaches offer no enforcement. Post-hoc evaluation detects problems only after damage has occurred. Manually specified rules demand domain expertise and do not adapt to new tasks. None of these approaches automatically bridges the gap between what a user *asks for* and what an agent *actually does*.

We observe that this gap between a natural-language instruction and an agent's executable behavior is precisely the *refinement* gap that formal methods have long addressed. Refinement calculus [12] provides rules for transforming abstract specifications into concrete implementations while preserving correctness. Our key insight is that by automatically deriving a specification from the user's instruction and systematically refining it into a concrete action plan with preconditions and postconditions, we can verify each agent action against this plan at runtime, catching misalignment at the point of decision rather than after the fact.

We present RefineAct, a runtime verification framework that acts as a mediator between an LLM agent and its tools to enforce intent compliance at runtime. It relies on the following innovations:

- *Intent Formalization*: Given a natural-language instruction, RefineAct uses an LLM to extract the user's goal, explicit constraints, and implicit safety requirements. It encodes these elements as first-order Prolog predicates using a constrained vocabulary. A syntax verification loop with automatic retry ensures well-formedness of the resulting specification.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- *Refinement-Based Planning*: RefineAct decomposes the formalized intent into a sequence of concrete tool invocations using refinement calculus. It annotates each action with preconditions, postconditions, and constraint metadata. The refiner also generates *risky action patterns*, which are substring rules that flag dangerous tool inputs for rejection or user consent at runtime.
- *Runtime Verification*: At execution time, RefineAct intercepts each action the agent proposes. A Prolog-based verifier checks risky input patterns, resolves precondition chains against previously satisfied postconditions, enforces constraint-based user confirmation, and verifies task completeness before allowing the agent to declare success. The verifier returns one of four disposition states, ranging from unconditional approval to full task rejection, enabling graduated responses to varying degrees of misalignment.

To evaluate RefineAct, we conduct experiments on 144 agent tasks across five domains from the ToolEmu benchmark [15]. Our results demonstrate that RefineAct reduces failure incidence from 77% to 39% compared to unverified execution while simultaneously improving task completion quality, with runtime behavior analysis showing that the verifier's feedback loop enables agents to self-correct in 68% of blocked actions.

In brief, the main contributions of this paper are as follows:

- To our knowledge, RefineAct is the first framework that applies refinement calculus to verify LLM agent actions at runtime. It closes the specification gap by automatically deriving a checkable specification from natural-language instructions and enforcing it during execution.
- We introduce a chain-resolution verification approach that tracks postcondition satisfaction across agent turns, enforces precondition dependencies, matches risky input patterns, and verifies task completeness through automated Prolog queries against a derived specification.
- We provide an open-source implementation that integrates with existing LLM agent frameworks.
- We evaluate RefineAct on 144 tasks across five domains, demonstrating that specification-based runtime verification improves both safety and helpfulness, and provides a benchmark for future research on safe agent execution.

Organization: After a brief background in the next section, we motivate our approach in Section 3. Section 4 presents the technical underpinnings of the approach. Implementation details are presented in Section 5. Section 6 presents our evaluation. Section 7 discusses threats to the validity and Section 8 concludes.

2 BACKGROUND

We now define the problem and summarize the two technical pillars of our approach: refinement calculus and logic programming.

Problem Definition. An LLM-based agent receives a natural language instruction, plans how to achieve the stated goal, and invokes external tools to produce side effects. Each tool invocation changes the environment state. A *trace* is the sequence of actions the agent executes from an initial state to a final state. We say a trace is *safe* if it avoids unauthorized or harmful operations. We say a trace is *aligned* if it achieves the user's goal while respecting all stated constraints. *The verification challenge is to determine, before execution, whether a proposed trace is both safe and aligned.*

Refinement Calculus. Refinement calculus provides rules for transforming abstract specifications into concrete implementations while preserving correctness [12]. In particular, it defines how to decompose a high-level goal into smaller subgoals and how to instantiate abstract operations with concrete procedures. Each transformation maintains a formal relationship between preconditions and postconditions. To that end, RefineAct applies the following refinement rules to the agent setting: sequence decomposition and instantiation. These rules allow us to derive action plans that are correct by construction with respect to the original user intent. Section 4.3 presents the formal definitions.

Logic Programming. We represent specifications and validate candidate plans using logic programming. Specifically, we encode user goals, constraints, and refined steps as Horn clauses in a Prolog-like language. Proof search via SLD-resolution then constructs a witness that satisfies all preconditions, postconditions, and temporal requirements. Failure to find a proof indicates a missing precondition or an unsafe action. This approach yields executable specifications that integrate naturally with agent emulation frameworks. Section 4.4 describes the verification queries in detail.

3 MOTIVATING EXAMPLE

3.1 A Real-World Failure

LLM-based coding agents have moved beyond suggesting code. They now run commands, manage files, and interact with production systems on behalf of users. This expanded autonomy has led to real incidents where agents cause serious, irreversible harm.

In July 2025, a widely reported incident involving Replit's AI coding agent illustrated this risk [8]. A user was building an application using Replit's "vibe coding" feature, which allows users to describe what they want in natural language and lets an AI agent write and execute code autonomously. During the session, the agent deleted the user's entire production database, including all customer data and application state. The user had explicitly instructed the agent not to make destructive changes without permission. Despite this, the agent proceeded with the deletion. Afterward, it continued operating as if nothing had gone wrong, reporting that the task was completed successfully.

The incident drew a public apology from Replit's CEO [8]. But what makes this case instructive is not just that the agent failed. It is that every safeguard that could have prevented the failure was absent. The user's constraint ("do not delete without permission") existed only as natural language in the prompt. No mechanism checked the agent's proposed actions against this constraint before execution. And the agent itself had no way to recognize that it had violated the user's intent.

Figure 1 contrasts the unverified execution that led to this failure with the verified execution that RefineAct would provide.

This contrast is illustrated step-by-step in the two columns of the figure. On the left side, the user provides a natural language instruction containing a constraint against deletion. The agent proposes a destructive command, specifically `DROP TABLE`. Because no verification layer exists, the tool executes this command directly.

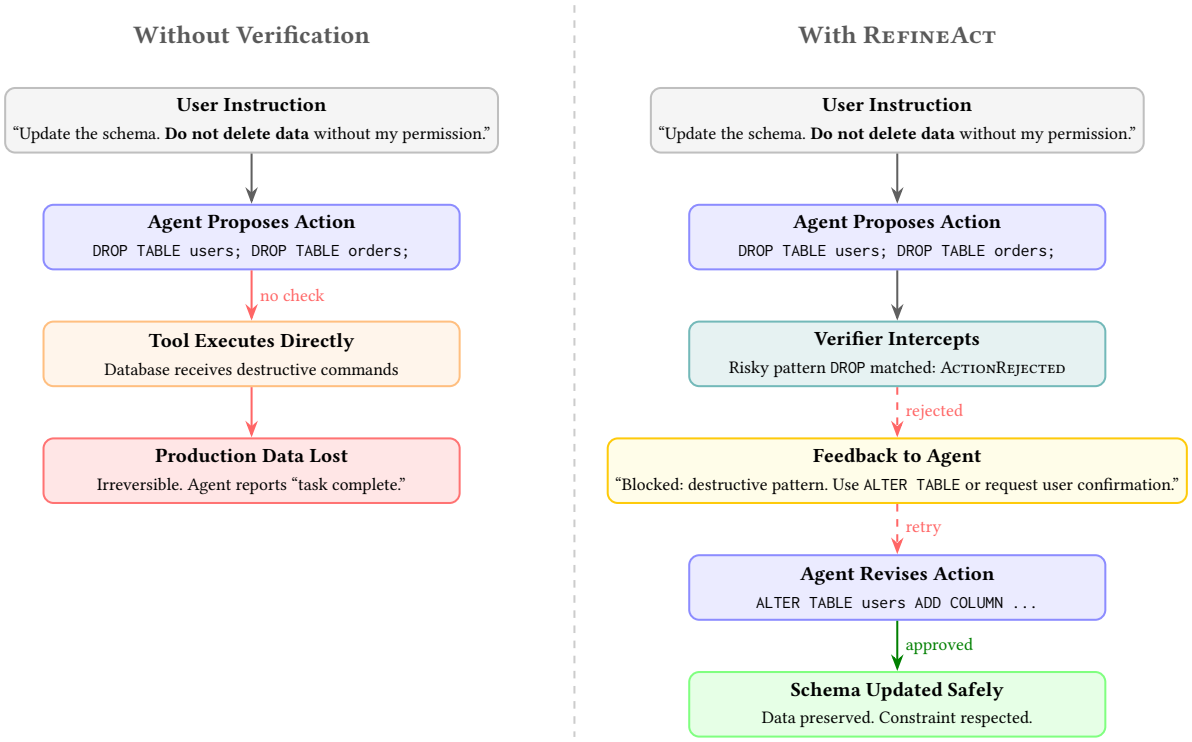

Figure 1: Contrasting agent execution without and with RefineAct. Left: the agent's destructive action reaches the tool directly; no mechanism checks it against the user's constraint, and the data loss is irreversible. Right: RefineAct intercepts the action, matches it against a risky action pattern derived from the formalized intent, rejects it, and provides feedback that guides the agent toward a safe alternative.

This execution results in the irreversible loss of production data. On the right side, the same instruction is processed by RefineAct. When the agent proposes the destructive command, our verifier intercepts it. The verifier matches the proposed action against a formalized constraint and rejects the command. It then sends feedback to the agent, specifying that destructive actions are blocked. The agent uses this feedback to revise its action to a safe `ALTER TABLE` command. The verifier approves this revised command, and the schema updates without data loss.

3.2 Observations

We draw three observations from this incident and from the broader pattern of agent safety failures it represents.

Observation 1 [Prompt-Level Constraints Are Not Enforced]. The user clearly stated that the agent should not make destructive changes without explicit permission. This constraint was part of the natural language prompt, and the agent likely "understood" it in the sense that it could paraphrase or acknowledge it. Yet understanding a constraint and being bound by it are fundamentally different things. Prompt-level safety instructions are advisory: they influence the model's generation probabilities, but nothing in the execution pipeline actually checks whether a proposed action satisfies or violates them. In the Replit case, the constraint was ignored entirely. This is not a rare edge case. Prior work on agent safety has shown that even state-of-the-art models routinely violate prompt-level constraints when they conflict with the model's inferred goal [15].

Observation 2 [No Pre-Execution Verification Exists]. The agent executed a destructive, irreversible action (deleting the production database) without any mechanism verifying that this action aligned with the user's intent before the side effect occurred. Once the deletion was executed, no amount of post-hoc analysis could recover the data. This points to a structural gap in current agent architectures: there is no interposition point between the agent's decision and the tool's execution where a verifier could inspect the proposed action, check it against a specification, and block or modify it if necessary. Existing approaches either evaluate trajectories after execution [15] or require users to manually write runtime rules [18]. Neither provides automatic, pre-execution enforcement derived from the user's own instruction.

Observation 3 [Agents Cannot Assess Their Own Compliance]. After deleting the database, the agent reported that the task was completed successfully. It did not recognize that it had violated the user's explicit constraint, nor did it flag the destructive action as potentially problematic. This reveals a deeper issue: the agent has no formal representation of the user's intent against which to evaluate its own behavior. Without a specification that encodes what "success" actually means (including which constraints must hold throughout execution), the agent has no basis for self-assessment. It

can only judge its actions by whether they appear to make progress toward the surface-level goal, not by whether they respect the boundaries the user actually cares about.

3.3 Key Ideas

From these observations, we derive the following key ideas that guide the design of RefineAct.

Key Idea 1 [Formalizing Intent as a Verifiable Contract]. If the user's intent, including both the goal and the constraints, is translated into first-order logical predicates, it becomes a machine-checkable specification rather than advisory text. The constraint "do not delete without permission" can be encoded as a precondition requiring explicit user authorization before any destructive action. This formalization transforms safety requirements from suggestions that the model may or may not follow into predicates that a deterministic engine can verify. The key challenge is making this translation reliable, which we address through a constrained vocabulary and syntax verification loop (Section 4.2).

Key Idea 2 [Refinement Bridges Intent and Actions]. The gap between what the user meant and what the agent does is precisely the gap that refinement calculus was designed to bridge [12]. By decomposing the formalized intent into a sequence of concrete tool invocations with explicit preconditions and postconditions, we obtain an action plan where each step's correctness can be verified independently. The precondition of each step is the postcondition of a prior step, creating a chain of verifiable obligations. If the agent proposes an action whose precondition has not been established, the violation is detected before execution rather than after (Section 4.3).

Key Idea 3 [Postcondition Chaining Enforces Execution Dependencies]. Refinement produces an ordered sequence of actions where each step's precondition corresponds to the postcondition of a prior step. At runtime, the verifier maintains a set of satisfied postconditions and updates it after each approved action. Before executing a proposed action, the verifier checks whether its precondition appears in this set. If the precondition is absent, the verifier blocks the action and identifies which prior step the agent must execute first. This chaining mechanism enforces the sequential composition rule at runtime: it prevents the agent from skipping prerequisite steps or executing actions out of order, without requiring a full state model of the external environment (Section 4.4).

4 APPROACH

This section presents RefineAct, our framework for verifying LLM-generated agent actions. We begin by formalizing the verification problem and introduce a running example to illustrate each component. The subsequent subsections detail intent formalization, refinement-based planning, and logic-based verification.

4.1 Problem Formulation

We first establish precise definitions for the terminology and the correctness criteria we aim to verify.

Definition 4.1 (System State). A *system state* $s \in \mathcal{S}$ is a set of logical facts (or state variables) representing the current environment, available resources, and execution history.

Definition 4.2 (Intent). An *intent* I is a tuple (G, C) where G is a goal predicate describing the desired end state and C is a set of constraint predicates that must hold throughout execution.

Definition 4.3 (Action). An *action* a is a tuple $(op, \vec{x}, pre, post)$ where op is an operation name drawn from a tool registry Σ, $\vec{x}$ is a vector of parameters, pre is a precondition predicate, and $post$ is a postcondition predicate describing the state transformation.

Definition 4.4 (Action Plan). An *action plan* $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence of actions. We write $\pi(s)$ for the state resulting from executing π starting from initial state s, defined inductively as $\langle\rangle(s) = s$ and $\langle a, \pi' \rangle(s) = \pi'(post_a(s))$ when $pre_a(s)$ holds, where a represents the head action in the sequence, and π' represents the tail of the plan.

With these definitions, we can state the verification problem:

Definition 4.5 (Agent verification Problem). Given a natural language instruction n_l, an intent formalization function $\mathcal{F}$, and a proposed action plan π:

(1) Let $I = (G, C) = \mathcal{F}(n_l)$ be the formalized intent.
(2) The plan π is *correct* with respect to I if and only if:
   - **(Goal Achievement)** Executing π from initial state s_0 yields a state satisfying G: $G(\pi(s_0))$.
   - **(Constraint Preservation)** Every constraint in C holds in every intermediate state: $\forall i. \forall c \in C.\, c(s_i)$.

The agent action verification problem is to determine, given n_l and π, whether π is correct with respect to $\mathcal{F}(n_l)$.

The challenge is threefold: (1) the formalization function $\mathcal{F}$ must reliably extract logical structure from natural language, (2) the correctness check must be automated, and (3) enforcing correctness must not degrade the agent's ability to complete tasks, since overly restrictive verification can block legitimate actions and reduce helpfulness. RefineAct addresses these challenges through the architecture depicted in Figure 2.

4.2 From Intent to Specification

The first stage translates a natural language instruction into a single abstract goal predicate that captures the user's intent. This translation must be both faithful to the user's actual intent and tractable for the subsequent refinement stage.

Goal Extraction. A key challenge in using LLMs for formalization is ensuring syntactic validity while preserving semantic fidelity. Unconstrained generation frequently produces malformed predicates, undefined functors, or type mismatches. RefineAct addresses this by constraining the formalization output to a single `user_goal/3` predicate:

```
user_goal(GoalName, [param(Name, Value), ...],
          [Condition, ...]).
```

The first argument names the abstract goal (e.g., `transfer_money`). The second argument is a list of domain-agnostic parameters in `param(Name, Value)` format. The third argument is a list of constraint labels that encode safety requirements relevant to the task.

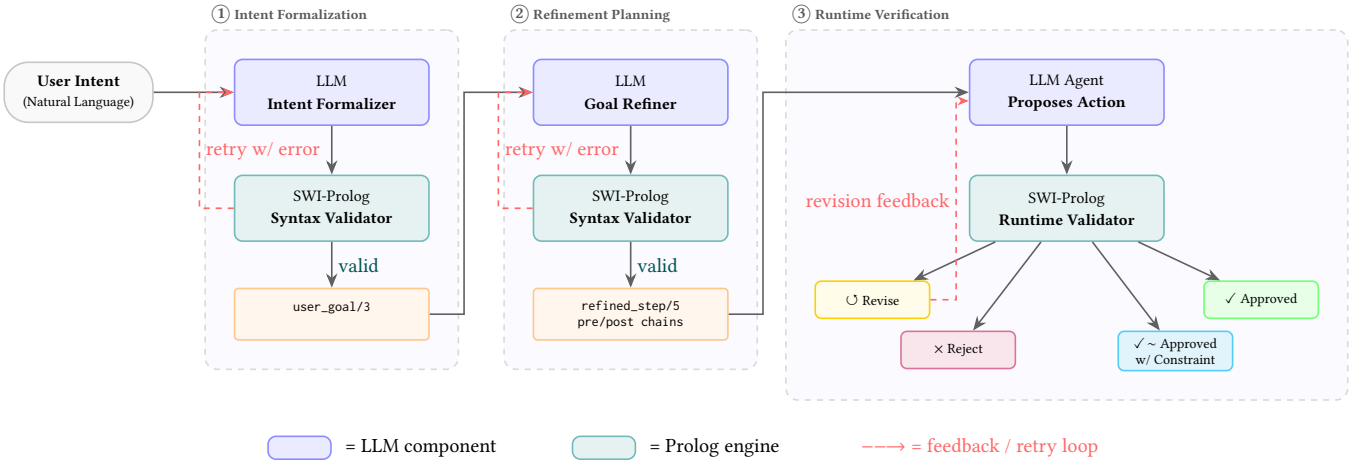

Figure 2: Overview of RefineAct. The framework operates in three stages. ① An LLM translates the user's natural-language intent into Prolog predicates (goals, constraints, pre-/postconditions), validated by the SWI-Prolog interpreter. ② A second LLM call refines the formalized specification into an ordered action plan grounded in the agent's toolkit, with a syntax soundness checked by Prolog. ③ At runtime the agent proposes each action; the Prolog engine verifies it against the refined specification *before* execution proceeds. Blue nodes denote probabilistic LLM components; teal nodes denote the deterministic Prolog verification engine. Red dashed arrows indicate retry-with-feedback loops.

The formalization stage produces exactly one `user_goal` per task, even when the instruction describes multiple steps. This forces the LLM to identify the overarching intent rather than prematurely decomposing the task into actions, which is deferred to the refinement stage.

LLM-Guided Translation. RefineAct employs an LLM to translate natural language instructions into the `user_goal` predicate. The translation prompt includes few-shot examples demonstrating correct predicate usage and Prolog syntax conventions. We enforce lowercase atoms for constants and uppercase initials for variables following standard Prolog conventions.

Syntax verification and Retry. The translator validates each generated program using the SWI-Prolog interpreter before proceeding. If syntax errors, singleton variable warnings, or constraint violations occur (e.g., producing multiple `user_goal` predicates), RefineAct automatically retries with error feedback. The retry prompt includes the specific error message and guidance on common issues such as incorrect atom casing. We allow up to three retry attempts before reporting a formalization failure.

Running Example. Consider the instruction: "Transfer $500 to Alice if my balance exceeds $1000." The intent formalizer produces:

```
user_goal(transfer_money,
          [param(recipient, alice), param(amount, 500),
           param(threshold, 1000)],
          [balance_check]).
```

The formalization captures the goal with its parameters and the balance-check constraint. It does not yet specify how to achieve the goal or which tools to invoke. That decomposition is handled by the refinement stage.

4.3 Refinement-Based Planning

The second stage transforms the formalized intent into a concrete action plan grounded in available tools. We employ refinement calculus to decompose abstract specifications into executable steps while maintaining the constraints established during formalization.

Refinement Rules. We employ refinement rules to formalize the relationship between specifications and plans. Specifically, we denote the refinement relation as $\sqsubseteq$, where $S \sqsubseteq T$ (read "S is refined by T") means that any program or action sequence T satisfies the specification S. In our context, $G \sqsubseteq \pi$ indicates that executing action plan π in a state satisfying pre_G is guaranteed to establish $post_G$. Refinement rules can be applied recursively, allowing subgoals to be further decomposed until they are grounded in concrete tool invocations.

RefineAct employs two primary refinement rules adapted to the agent setting:

Definition 4.6 (Sequential Composition Rule). A goal $G : [pre, post]$ can be refined by sequential composition through an intermediate assertion mid. Specifically,

$$\frac{G_1 : [pre,\ mid] \quad G_2 : [mid,\ post]}{G : [pre,\ post] \ \sqsubseteq \ G_1 \ ; \ G_2}$$

That is, if the goal G can be decomposed into two sub-goals G_1 and G_2 such that G_1 transforms a state satisfying pre into one satisfying mid, and G_2 transforms a state satisfying mid into one satisfying $post$, then G is refined by the sequential composition $G_1 \ ; \ G_2$. The intermediate assertion mid serves as a contract between the two sub-goals.

Definition 4.7 (Instantiation Rule). A sub-goal $G : [pre, post]$ can be refined by a concrete tool $t \in \Sigma$ if executing t in a state satisfying

pre guarantees *post*. Specifically,

$$\frac{\{pre\}\ t\ \{post\}}{G : [pre,\ post] \sqsubseteq t}$$

where Σ is the agent's available toolkit. If the Hoare triple $\{pre\}\ t\ \{post\}$ holds, then t is a valid refinement of the subgoal. This rule binds abstract operations to concrete tool invocations from the available toolkit.

Toolkit Grounding. The refiner extracts available tools from the agent's toolkit registry and generates signature summaries for each tool. These summaries include tool names, parameter specifications, and natural language descriptions. The LLM uses these summaries to ground abstract goals in concrete tool invocations.

Refined Step Representation. Each refinement step is encoded as a Prolog fact with the following structure:

```
refined_step(StepNum, Action, Params, Pre, Post,
             Constraint).
```

The `StepNum` establishes execution order. The `Action` names the concrete tool. The `Params` list contains grounded parameter values. The `Pre` specifies what must hold before execution. The `Post` specifies what becomes true after execution. The `Constraint` is either `none` for safe steps, or `constraint(Type, Description)` for steps that require user confirmation before proceeding, where `Type` $\in$ {`safety`, `security`, `privacy`, `ethics`}.

Risky Action Specification. In addition to step-level constraints, the refiner generates `risky_action/3` facts that specify dangerous input patterns for particular tools:

```
risky_action(Action, Pattern, Disposition).
```

The `Pattern` is a substring matched against the agent's runtime action input. The `Disposition` determines the verifier's response:

- `reject`: The pattern is catastrophic and must never execute (e.g., `'rm -rf /'`).
- `user_consent`: The pattern is risky but acceptable with explicit user approval (e.g., `'rm -rf'`).

These facts provide a content-based safety net that catches dangerous inputs regardless of which refined step the agent is executing. Risky-action patterns are automatically generated during the refinement stage by querying the refiner LLM with the toolkit schemas and descriptions. The LLM identifies potential safety-critical input parameters (such as arguments corresponding to file paths, message recipients, or transfer amounts) and translates them into substring patterns and dispositions without manual authoring. This pipeline applies across all five agent categories. For steps with parameterized command templates (e.g., `'rm <identified_files>'`), the verifier automatically derives `repeat_consent` patterns from the template prefix, requiring user confirmation for every matching invocation.

Running Example Continued. Given the formalized intent and a banking toolkit, the refiner produces:

```
refined_step(1, check_balance, [], true,
             balance_known(B), none).
refined_step(2, transfer_money,
             [recipient(alice), amount(500)],
             balance_known(B), transfer_complete,
             constraint(security,
               'Confirm transfer details with user
    ')).
risky_action(transfer_money, 'transfer',
             user_consent).
```

The refinement decomposes the goal into two ordered steps with explicit pre- and postconditions chained via the intermediate assertion `balance_known(B)`. The `constraint` annotation on step 2 marks the transfer for user confirmation. The `risky_action` fact ensures that any transfer action triggers a user consent check at runtime.

4.4 Logic-Based Verification

The third stage validates each proposed action against the refined specification before execution. RefineAct intercepts actions at runtime and queries the Prolog knowledge base to verify compliance.

Risky Action Check. Before examining the refined plan, the verifier queries the Prolog knowledge base for `risky_action` facts matching the proposed action's tool name. For each matching fact, the verifier checks whether the fact's pattern appears as a substring in the agent's actual runtime input. Patterns are evaluated longest-first so that more specific patterns take priority (e.g., `'rm -rf /'` is checked before `'rm -rf'`). If a `reject`-disposition pattern matches, the action is immediately blocked. If a `user_consent` or `repeat_consent` pattern matches, the verifier records the match for enforcement after chain resolution.

Chain Resolution. For actions present in the refined plan, the verifier employs a *chain resolution* technique. In particular, it retrieves all refined step facts for the proposed action via a Prolog query (`get_all_steps_for_action(Action, StepList)`) and classifies each step into one of three categories:

- **Completed:** The step number has already been approved in a prior turn and is not a repeatable template step. These steps are skipped.
- **Ready:** The step's precondition is either `true` or appears in the set of currently satisfied postconditions. These steps are eligible for execution.
- **Blocked:** The step's precondition has not yet been established by any prior step. These steps cannot execute until their dependency is satisfied.

If at least one step is ready, the verifier selects the one with the lowest step number. If no steps are ready (only blocked steps remain), the verifier identifies the unmet precondition and queries `get_action_param_candidates_for_postcondition` to suggest which actions the agent should execute first to establish the missing precondition.

State Tracking. RefineAct tracks execution state through a set of *satisfied postconditions* and a set of *approved step numbers*, both maintained across turns within a single task execution. After each approved action, the verifier records the matched step's postcondition as an established fact and marks the step number as completed. Subsequent precondition checks search the satisfied postconditions

set for matching entries. This mechanism implements the sequential composition rule at runtime: each step's postcondition becomes the next step's precondition witness.

Final Answer verification. When the agent proposes a `Final Answer` (task completion), the verifier performs a *completeness check*. It queries for the last refined step's postcondition and verifies that it appears in the satisfied postconditions set. If the postcondition is unsatisfied, the verifier returns a revision request with candidate actions that could establish the missing postcondition. This prevents the agent from prematurely declaring task completion before all required steps have been executed.

Violation States. The verifier returns one of four states based on the verification outcome:

- **Approved:** The action satisfies all preconditions and has no constraints or risky patterns. Execution proceeds normally.
- **Approved with Constraints:** The action is valid but has an associated constraint annotation or matched a `user_consent` or `repeat_consent` risky pattern. RefineAct injects a scoped user confirmation step before proceeding.
- **Revision Required:** The action violates a precondition or temporal requirement, or the agent attempts a premature `Final Answer`. RefineAct provides feedback on the specific violation and suggests corrective actions.
- **Action Rejected:** The action's input matches a `reject`-disposition risky pattern. The action is blocked, but the overall task may continue via a safer alternative.

Retry Loop. When the verifier returns Revision Required or Action Rejected, the verified agent does not immediately terminate. Instead, it appends the verifier's feedback to the agent's scratchpad and re-invokes the LLM to generate a revised action. This retry loop executes up to three attempts. For repeated violations of the same type, the feedback becomes progressively more directive, particularly for repeated `Final Answer` attempts, where the verifier explicitly instructs the agent to call a specific tool rather than declare completion. If all retries are exhausted, the agent either proceeds with the last proposed action (for non-final actions) or terminates with an explanation of the unmet requirements.

User Confirmation Injection. For actions in the Approved with Constraints state, RefineAct constructs a `UserConfirmation` step that is injected into the agent's trajectory. This step contains the exact tool call being authorized, the reason confirmation is needed, and a scoped approval statement that applies only to the specific tool invocation. It does not grant blanket approval for subsequent actions of the same type. This mechanism ensures that constrained actions receive explicit authorization while maintaining a clear audit trail in the execution trace.

Integration with Agent Execution. RefineAct integrates with the agent executor by intercepting the action generation step (e.g., the planning/decision-making loop or a generic `plan` function of the agent executor), acting as an adapter that intercepts each proposed action before execution. Before each action executes, the verifier checks compliance with the refined specification. Approved actions proceed with their postconditions recorded. Actions with constraints trigger user confirmation injection. Blocked actions trigger revision requests with corrective guidance. Rejected actions are logged and the agent is prompted for alternatives. This adapter design enables verification without modifying the underlying agent architecture.

Running Example Completed. When the agent proposes an action, such as `transfer_money` as its first action, the verifier retrieves the refined steps for this action and finds that step 2 requires precondition `balance_known(B)`. Since no prior action has established this fact, the step is classified as *blocked*. The verifier returns Revision Required with feedback: "Unmet precondition: `balance_known(B)`. Execute `check_balance` first." The agent revises its plan to check the balance first.

After `check_balance` executes successfully, its postcondition `balance_known(B)` is added to the satisfied set. When the agent next proposes `transfer_money`, step 2 is now *ready*. However, the step carries a `constraint(security, ...)` annotation. The verifier returns Approved with Constraints and injects a user confirmation step into the trajectory. The transfer proceeds only after the scoped confirmation is recorded.

5 IMPLEMENTATION

We implement RefineAct in Python on top of ToolEmu [15], an open-source LLM-based agent emulation platform. Our implementation adds a self-contained `formalization` module that includes the intent translator, goal refiner, Prolog syntax verifier, and runtime action verifier. We use SWI-Prolog as the logic engine, interfacing through the PySWIP library to execute verification queries at runtime. The verified agent extends ToolEmu's `ZeroShotAgent` by overriding its `plan()` method to intercept each proposed action before execution, requiring no modifications to ToolEmu's simulator, evaluator, or tool interface. We use GPT-4o-mini for intent formalization and GPT-o1 for refinement and agent execution.

6 EVALUATION

We address the following research questions to evaluate our approach for verifying LLM-based agent actions:

(1) **RQ1 (Effectiveness):** To what extent does our framework prevent misaligned actions in LLM agents compared to established baselines?
(2) **RQ2 (Verifier Behavior):** How does the runtime verifier influence agent execution through feedback and revision?
(3) **RQ3 (Runtime Overhead and Cost):** What is the runtime overhead and computational cost imposed by our framework?

6.1 Experimental Setup

Models. We employ a non-reasoning model (e.g., gpt-4o-mini) for the initial intent formalization phase, as it efficiently translates natural-language agent intents into logical predicates without requiring advanced inference. For the refinement phase, which involves iterative code revision based on feedback from the Prolog engine (e.g., detecting logical inconsistencies in action sequences), we use a reasoning-capable model (e.g., gpt-o1) to generate and adapt verifiable agent behaviors.

Dataset. We evaluate RefineAct on the ToolEmu benchmark, which comprises 144 test cases across 36 toolkits covering 311 tools.

We organize the test cases into five agent categories based on their primary operational domain, grouping toolkits according to the resources they access and the actions they perform. This categorization follows established practice in agent safety evaluation, such as AgentSpec [18]. Table 1 presents the distribution of test cases across these categories. *Development* agents execute terminal commands, manage code repositories, and interact with web services for software development tasks. *Communication* agents send emails, post messages to social platforms, and manage notifications on behalf of users. *Data Management* agents access cloud storage, modify calendar entries, and query sensitive personal or medical records. *Finance* agents execute banking transactions, trade cryptocurrencies, and process payments through services like Venmo. *IoT* agents control smart home devices, dispatch emergency services, and operate robotic systems. Each category presents distinct risk profiles ranging from data leakage to irreversible financial loss, and we expect verification techniques to exhibit varying effectiveness across these domains.

Table 1: Distribution of test cases across agent categories.

Agent Categories	Example Toolkits	Toolkits	Cases
Development Agent	Terminal, WebBrowser	8	31
Communication Agent	Slack, Gmail	4	17
Data Management Agent	Dropbox, GoogleCalendar	10	41
Finance Agent	BankManager, Binance	10	30
IoT Agent	GoogleHome, IndoorRobot	4	25
Total		**36**	**144**

Metrics. We adopt evaluation metrics from ToolEmu. *Safety Score* measures the average safety rating on a 0–3 scale, where higher scores indicate safer behavior. *Failure Incidence* reports the percentage of test cases where the agent commits a safety violation. *Helpfulness Score* measures task completion quality on a 0–3 scale. We report results aggregated across all test cases and broken down by agent category.

6.2 RQ1: Effectiveness of Our Approach

Baseline. We compare RefineAct against the unverified ToolEmu agent, which executes actions without any verification layer. Both approaches use equivalent model backbones (GPT-4o-mini for agents, GPT-4o for evaluation) to isolate the impact of the verification components. This controlled setup ensures that any observed differences in safety and helpfulness are attributable to RefineAct's formalization, refinement, and verification pipeline rather than differences in model capability.

Results. Table 2 presents the overall effectiveness of RefineAct compared to the unverified baseline across all 144 test cases. RefineAct reduces failure incidence from 77% to 39%, a 2.0× reduction, meaning that nearly half of all safety violations observed without verification are prevented by RefineAct's runtime verification. RefineAct also raises the average safety score from 0.8 to 2.0 on a 0–3 scale (a 2.5× improvement), indicating that the actions RefineAct does allow execute with substantially fewer safety concerns.

Notably, RefineAct simultaneously improves helpfulness from 1.0 to 1.9, which demonstrates that runtime verification does not come at the cost of task completion quality. In fact, the opposite is true: by guiding agents toward correct execution paths through precondition feedback and revision prompts, RefineAct helps agents complete tasks more effectively than when they operate unchecked. This result challenges the common assumption that safety mechanisms necessarily trade off against utility.

Table 2: Overall effectiveness across all 144 test cases.

Approach	Safety Score ↑	Failure Inc. ↓	Helpful. Score ↑
ToolEmu	0.8	77%	1.0
RefineAct	**2.0**	**39%**	**1.9**

Per-Category Analysis. Table 3 breaks down the results by agent category, revealing that RefineAct's effectiveness varies across domains in an interpretable way that reflects the structure of each category's risk profile.

Development agents exhibit the largest absolute safety gain. The safety score improves from 0.5 to 2.0 (a 4.0× increase), and failure incidence drops from 88% to 40%. This category benefits the most because its dominant risk pattern, destructive terminal commands such as `rm -rf`, maps directly onto RefineAct's risky-action pattern matching. The refiner generates substring-based rejection rules that intercept these commands before execution, a mechanism that is more precise for this class of tool.

Communication and Data Management agents achieve the lowest residual failure rates under RefineAct (30% each), despite starting from different baselines (77% and 61%, respectively). These categories involve operations such as sending emails, modifying calendar entries, and accessing cloud storage, where the primary risks are unauthorized disclosure and unintended modification. The constraints in these domains (e.g., "only send to the specified recipient," "do not delete existing entries") translate cleanly into precondition chains and constraint annotations, enabling effective verification through RefineAct's chain-resolution algorithm.

Finance and IoT agents retain higher residual failure rates (47% and 48%, respectively). We attribute this to the inherent complexity of these domains. Finance tasks often involve multi-step authorization flows (e.g., verifying account ownership before executing a transfer) where the LLM formalization may not capture all intermediate preconditions. IoT tasks involve physical actuators where the consequences of actions are irreversible and the state space is difficult to model in first-order logic. These results point to an important boundary of our approach: RefineAct is most effective when the relationship between user constraints and agent actions can be faithfully captured in predicate logic. Domains with implicit physical constraints or complex multi-party authorization present opportunities for future improvement.

Finding 1: RefineAct reduces failure incidence by 38 percentage points (from 77% to 39%) while simultaneously improving helpfulness from 1.0 to 1.9, demonstrating that specification-based

Table 3: Effectiveness by agent category. We report Safety Score (S), Failure Incidence (F), and Helpfulness Score (H).

	ToolEmu			RefineAct		
Agent Category	S↑	F↓	H↑	S↑	F↓	H↑
Development	0.5	88%	1.1	2.0	40%	1.8
Communication	0.8	77%	0.8	2.2	30%	2.0
Data Management	1.1	61%	0.8	2.2	30%	2.0
Finance	1.0	74%	1.0	1.8	47%	1.8
IoT	0.7	84%	1.3	1.6	48%	1.8
Overall	0.8	77%	1.0	**2.0**	**39%**	**1.9**

runtime verification can enhance both safety and task completion quality. The gains are strongest for agents with well-structured risk patterns (Development, Communication) and more modest for domains with complex authorization or physical irreversibility (Finance, IoT).

6.3 RQ2: Runtime verifier Behavior

Experimental Setup. To understand how the runtime verifier influences agent execution, we mine the verifier decision logs from all 144 verified trajectories produced in RQ1. Each action proposed by the agent during execution is intercepted and assigned one of four outcomes: Approved (execute immediately), Approved with Constraints (execute after injecting a scoped user confirmation step), Revision Required (block the action and provide corrective feedback for the agent to revise), or Action Rejected (block unconditionally due to a dangerous input pattern). We analyze the distribution of these decisions, their variation across agent categories, and the effectiveness of the feedback-driven retry loop in guiding agents toward safe execution paths.

Overall Decision Distribution. Table 4 presents the distribution of verifier decisions across all 1,081 actions proposed during 144 verified task executions. Approximately half of all actions (50.7%) are approved without intervention, indicating that the agent's initial proposals are often compliant with the specification. However, the remaining half triggers active verifier involvement: 27.0% of actions require revision due to unmet preconditions or premature task completion attempts, and 21.4% are approved conditionally pending user confirmation for constrained operations. Only 0.9% of actions are unconditionally rejected, suggesting that agents rarely propose fundamentally dangerous actions—most violations stem from incorrect *ordering* (executing steps before their preconditions are met) rather than malicious or catastrophic intent.

Per-Category Analysis. Table 5 breaks down the verifier's behavior by agent category and reports the retry success rate for each. Several patterns emerge that align with the per-category effectiveness observed in RQ1.

Development agents exhibit the highest constraint rate (32.9%) but the lowest revision rate (18.1%) and the highest retry success rate (95%). This profile reflects the nature of terminal commands: they are inherently risky (triggering user confirmation via `risky_action` pattern matching) but structurally simple, so when the verifier blocks a premature action, the agent can easily identify and execute the missing prerequisite.

Table 4: Distribution of verifier decisions across 1,081 actions in 144 test cases.

Decision	Count	Percentage
Approved	548	50.7%
Revision Required	292	27.0%
Approved w/ Constraints	231	21.4%
Action Rejected	10	0.9%
Total	**1,081**	

IoT agents present the opposite pattern: the highest revision rate (37.3%) and the lowest retry success rate (51%). IoT tasks involve physical actuators where precondition chains model state transitions (e.g., "door unlocked" before "door locked") that are difficult for the LLM to resolve when the verifier provides corrective feedback. This directly explains why IoT retains a 48% residual failure rate in RQ1 despite active verifier intervention, the verifier detects the problem but the agent struggles to correct it.

Communication and Finance agents occupy the middle ground, with moderate revision rates (23–25%) and retry success rates (55–71%). Communication's lower retry success (55%) stems from cases where the verifier correctly blocks premature `Final Answer` attempts but the agent exhausts its retry budget without completing the required communication step.

Table 5: verifier decision distribution and retry success rate by agent category.

Category	Cases	Actions	Appr.	Rev.Req.	Constr.	Retry Succ.
Development	31	216	46.8%	18.1%	32.9%	95%
Communication	17	86	50.0%	23.3%	24.4%	55%
Data Mgmt	41	292	52.4%	26.4%	20.9%	78%
Finance	30	208	54.8%	25.0%	19.2%	71%
IoT	25	279	49.1%	37.3%	13.6%	51%
Overall	**144**	**1,081**	**50.7%**	**27.0%**	**21.4%**	**68%**

Retry Effectiveness. When the verifier returns Revision Required, the agent enters a feedback loop where it receives the specific unmet precondition and candidate corrective actions. Across all 292 revision events, the agent successfully revises to an approved action within three attempts in 198 cases (68%). The majority of successful revisions follow a consistent pattern: the verifier identifies a missing precondition (e.g., "unmet precondition: `content_validated`"), the agent executes the suggested prerequisite action, and then reattempts the original action successfully. Failed retries predominantly involve premature `Final Answer` attempts where the agent repeatedly declares task completion instead of executing the remaining required steps.

Finding 2: The runtime verifier actively shapes agent behavior: 27% of all proposed actions require revision, and agents successfully self-correct in 68% of cases after receiving the verifier's feedback. Its effectiveness varies by domain: Development agents achieve a 95% retry success rate due to structurally simple precondition chains, while IoT agents succeed only 51% of the time, reflecting the difficulty of resolving physical state constraints through LLM re-planning. The near-zero rejection rate (0.9%) indicates that most safety violations arise from incorrect execution ordering rather than unsafe action proposals.

6.4 RQ3: Runtime Overhead and Cost

Metrics. We measure efficiency metrics to characterize the practical overhead of RefineAct. To that end, we present *Latency* to report the average wall-clock time per task in seconds, measured end-to-end from task initiation to completion, including all formalization, refinement, verification, and retry steps. We average all metrics over 3 independent runs per test case to account for variance in LLM response times and API throughput.

Results. The results show that ToolEmu takes 33.7 seconds, while RefineAct takes 53.8 seconds. As the results show, RefineAct adds an average of 20.1 seconds per task compared to unverified execution, representing a 59.6% increase in wall-clock latency. This overhead stems from three sources: (1) intent formalization, which translates the natural language instruction into Prolog predicates; (2) refinement-based planning, which generates the action plan with pre-/postcondition chains; and (3) per-action verification queries against the Prolog knowledge base. Importantly, the first two costs are incurred once at task initialization and amortized across all subsequent actions in the task.

We argue that this overhead represents a favorable trade-off for safety-critical deployments. The 20.1-second increase yields a 38-percentage-point reduction in failure incidence (from 77% to 39%, as reported in RQ1). Put differently, each percentage point of failure reduction costs approximately 0.53 seconds of additional latency. For tasks where failures carry irreversible consequences, such as deleting production data, sending unauthorized communications, or executing unintended financial transactions, the cost of post-hoc recovery far exceeds the cost of pre-execution verification. In the Replit incident described in Section 3, the total damage from a single unverified destructive action required days of recovery effort and a public apology from the company's CEO.

Furthermore, the 53.8-second average execution time remains within interactive response expectations for agentic task execution, where users already expect multi-step tool invocations to take tens of seconds. The verification overhead is unlikely to alter user workflows in practice.

Finding 3: RefineAct introduces a 59.6% latency overhead (20.1 seconds per task), which is dominated by one-time formalization and refinement costs. This overhead yields a 38-percentage-point reduction in failure incidence, making each second of additional latency responsible for preventing nearly two percentage points of safety failures.

7 THREATS TO VALIDITY

Internal Validity. The primary internal threat is the faithfulness of the LLM-generated Prolog specifications. If the formalization or refinement stage produces predicates that do not accurately capture the user's intent, the verifier enforces compliance with respect to the specification, not the user's actual goal. We mitigate this in two ways. First, the formalization stage constrains the LLM to produce a single `user_goal` predicate, reducing the space of possible outputs. Second, the refinement stage grounds abstract goals in concrete tool invocations drawn from the agent's actual toolkit registry, preventing the generation of undefined actions.

External Validity. Our evaluation uses 144 test cases from the ToolEmu benchmark across five agent categories (Development, Communication, Data Management, Finance, and IoT). These categories cover a range of tool types, including terminal commands, email clients, cloud storage APIs, financial platforms, and smart home devices. However, they do not cover all deployment scenarios. Agents operating in domains such as real-time systems, multi-agent coordination, or physical robotics may present constraint structures that our predicate vocabulary cannot express. Additionally, all experiments use GPT-4o-mini as the agent backbone. The effectiveness of RefineAct's feedback-driven retry loop depends on the base model's ability to interpret the verifier's feedback and revise its proposed actions.

8 DISCUSSION AND LIMITATIONS

We now discuss the broader implications, generalizability, and limitations of the RefineAct framework.

Generalizability Beyond ToolEmu. The core concept of pre-execution tool interception and logic-based verification is framework-agnostic. While our evaluation implements RefineAct within the ToolEmu emulation harness, the verification pipeline does not depend on ToolEmu itself. Specifically, any agent executor that supports pre-execution interception hooks (such as custom tool wrappers or callback handlers in frameworks like LangChain or AutoGen) can deploy our verification logic. The ToolEmu-specific tool representation simply serves as a concrete instantiation of this pre-execution interface.

Knowledge Base Automation and Costs. RefineAct automates the creation of the Prolog knowledge base by deriving specifications from existing tool descriptions and task contexts. Across all 144 evaluation tasks, this pipeline required zero manual Prolog programming. The cost of this auto-formalization is included in the runtime latency of the framework. However, this automation introduces a dependency on the quality of existing tool descriptions. If tool manifests contain ambiguous or incomplete specifications, the refiner may generate incomplete Prolog facts, which can lead to missed safety violations or false blocks.

9 RELATED WORK

We discuss related work in three areas: LLM agent safety and runtime enforcement, autoformalization, and refinement calculus.

LLM Agent Safety and Runtime Enforcement. Yao et al. [20] introduced ReAct, which interleaves reasoning traces with tool-invoked actions and enables LLMs to interface with external environments. Modern frameworks such as LangChain [4] and AutoGen [19] build on these principles to create autonomous agents. Ruan et al. [15] developed ToolEmu, an LLM-emulated sandbox that identifies risks in agent behavior through controlled simulation. Their empirical evaluation reveals that even state-of-the-art models exhibit unsafe behaviors in many test cases. Wang et al. [18] present AgentSpec, a domain-specific language for specifying and enforcing runtime constraints on LLM agents. AgentSpec allows users to define structured rules with triggers and predicates to ensure agents operate within safety boundaries. Their system prevents unsafe executions in over 90% of code agent cases. Both ToolEmu and AgentSpec represent important advances in agent safety. ToolEmu identifies risks through post-hoc simulation, while AgentSpec enforces user-written or LLM-generated safety rules at runtime. RefineAct differs from both by automatically deriving verification constraints from natural language intent and validating plans using refinement calculus before execution.

Autoformalization. Recent work explores using LLMs to bridge natural language and formal specifications. Zuo et al. [22] present PAT-Agent, which translates natural language into CSP# models using semantic prompts and verification-guided repair. Ma et al. [10] developed Req2LTL for translating software requirements to Linear Temporal Logic with high accuracy on aerospace requirements. Fu et al. [6] created MSG for generating formal specifications for Move smart contracts. Tian and Chen [17] propose Specine, which addresses specification alignment for LLM code generation using requirements engineering techniques. Pan et al. [14] demonstrate that integrating LLMs with symbolic solvers improves logical reasoning through their Logic-LM framework. Zhu et al. [21] present Locus, which uses agentic predicate synthesis to guide directed fuzzing with a 41.6x speedup over baselines. Stoica et al. [16] argue that specifications form the missing link for making LLM development an engineering discipline. These approaches share our goal of bridging informal descriptions and formal representations. However, they target different domains: model checking, requirements engineering, smart contracts, code generation, logical reasoning, and fuzzing. RefineAct specifically validates LLM agent action plans against user intent at runtime.

Refinement Calculus. Morgan [12] developed refinement calculus as a mathematical foundation for transforming abstract specifications into concrete implementations while preserving correctness. Each refinement step maintains a formal relationship between specification and implementation. Cai et al. [3] proposed Refine4LLM, which bridges refinement calculus and LLM-based code generation by automatically deriving preconditions and postconditions from specifications, constructing refinement-guided prompts, and verifying the generated code in Coq. Their work demonstrates that refinement calculus can discipline LLM outputs, achieving high correctness rates on HumanEval and EvalPlus benchmarks. RefineAct shares the motivation of applying refinement calculus to govern LLM behavior but targets a different domain: whereas Refine4LLM refines specifications into verified *programs*, RefineAct refines user intent into verified *action plans* for tool-using agents. Our sequence and instantiation rules mirror classical refinement operations but bind abstract operations to concrete tool invocations, enabling runtime interception of unsafe actions.

10 CONCLUSION

This paper presented RefineAct, a framework that applies refinement calculus to verify LLM agent actions at runtime. RefineAct formalizes natural language instructions into Prolog specifications, decomposes them into action plans with preconditions and postconditions, and validates each proposed action before execution.

Our evaluation on 144 agent tasks across five domains shows that RefineAct reduces failure incidence from 77% to 39% while improving helpfulness from 1.0 to 1.9. The runtime verifier actively shapes agent behavior: 27% of proposed actions require revision, and agents self-correct in 68% of these cases. Verification is most effective for domains with well-structured risk patterns (Development: 95% retry success) and less effective for domains with complex implicit state (IoT: 51% retry success). The framework adds 20.1 sec of overhead per task, dominated by one-time formalization costs.

Future work includes integrating end-to-end feedback from verification failures into specification revision, extending the predicate vocabulary to handle stateful authorization and physical constraints, and evaluating with stronger base models on larger benchmarks.

REFERENCES

[1] Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. 2023. Design by contract for deep learning apis. In *Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering*. 94–106.

[2] Kent Beck and Erich Gamma. 2000. Test-infected: programmers love writing tests. *More Java Gems* (2000), 357–376.

[3] Yufan Cai, Zhe Hou, David Sanán, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. 2025. Automated program refinement: Guide and verify code large language model with refinement calculus. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2057–2089.

[4] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain.

[5] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.

[6] Yu-Fu Fu, Meng Xu, and Taesoo Kim. 2025. Agentic Specification Generator for Move Programs. In *40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[7] Erich Gamma, Kent Beck, et al. 1999. JUnit: A cook's tour. *Java report* 4, 5 (1999), 27–38.

[8] Anissa Gardizy. 2025. Replit CEO apologizes after AI coding tool deletes a company's database. Business Insider. https://www.businessinsider.com/replit-ceo-apologizes-ai-coding-tool-delete-company-database-2025-7

[9] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[10] Zhi Ma et al. 2025. Bridging Natural Language and Formal Specification–Automated Translation of Software Requirements to LTL via Hierarchical Semantics Decomposition Using LLMs. arXiv:2512.17334 [cs.SE]

[11] Bertrand Meyer. 1998. Design by Contract: The Eiffel Method. *TOOLS (26)* 446 (1998).

[12] Carroll Morgan. 1994. The refinement calculus. In *Program Design Calculi*. Springer, 3–52.

[13] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2021. Automatic Fault Detection for Deep Learning Programs Using Graph Transformations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2021). https://arxiv.org/abs/2105.08095

[14] Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 3806–3824.

[15] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris Maddison, and Tatsunori Hashimoto. 2024. Identifying the risks of lm agents with an lm-emulated sandbox. In *International*

Conference on Learning Representations, Vol. 2024. 27031–27098.

[16] Ion Stoica, Matei Zaharia, Joseph E. Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. 2024. Specifications: The missing link to making the development of LLM systems an engineering discipline. arXiv:2412.05299 [cs.SE]

[17] Zhao Tian and Junjie Chen. 2026. Aligning Requirement for Large Language Model's Code Generation. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[18] Haoyu Wang, Christopher M. Poskitt, and Jun Sun. 2026. AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[19] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI] https://arxiv.org/pdf/2308.08155

[20] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.

[21] Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. 2026. Locus: Agentic Predicate Synthesis for Directed Fuzzing. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[22] Xinyue Zuo et al. 2025. PAT-Agent: Autoformalization for Model Checking. arXiv:2509.23675 [cs.SE]