PALADIN: Self-Correcting Language Model Agents to Cure Tool-Failure Cases

Sri Vatsa Vuddanti, Aarav Shah, Satwik Kumar Chittiprolu, Tony Song, Sunishchal Dev, Kevin Zhu, Maheep Chaudhary

Algoverse AI Research
srivatsa644@gmail.com

Abstract

Tool-augmented language agents routinely fail in deployment due to execution-time tool errors such as timeouts, malformed outputs, or silent API failures. In agentic and multi-agent systems, these failures are especially damaging: a single unhandled error can cascade across reasoning steps or agents, leading to deadlock or hallucinated success. Despite this, most training pipelines optimize only for clean, successful trajectories and leave execution-level recovery largely unmodeled. We propose *PALADIN*, a framework for teaching language agents explicit, generalizable recovery behavior under tool failures. PALADIN trains agents on over 50,000 recovery-annotated trajectories generated via systematic failure injection aligned with the ToolScan taxonomy, while preserving base task competence through LoRA-based fine-tuning. At inference time, agents detect execution failures and condition their responses on a small, curated bank of taxonomy-aligned recovery exemplars, enabling structured diagnosis and repair rather than reactive retries. Across multiple backbones and evaluation settings, PALADIN consistently improves execution-level robustness. On deployment-relevant benchmarks, it raises Recovery Rate from 32.8% to 89.7%, reduces catastrophic (hallucinated) success, and substantially increases task completion, while incurring only modest efficiency overhead. Crucially, PALADIN generalizes to unseen tools and failure types, retaining over 95% recovery rate on out-of-distribution APIs. These results demonstrate that execution-level recovery is a learnable and transferable capability. By treating tool failure as a first-class training signal, PALADIN provides a practical foundation for building reliable, failure-aware language agents and offers a pathway toward safer and more robust LLM-based agentic and multi-agent systems.

Code — https://github.com/33k0/PALADIN-Framework
Dataset —
https://huggingface.co/datasets/E-k-O/PaladinData/

Introduction

Tool-augmented language models are increasingly deployed as agents: systems that plan, invoke external tools, and execute multi-step actions to achieve goals. In clean benchmark settings, such agents appear highly capable. In deployment, however, this competence often silently collapses at scale. APIs timeout, tool outputs are malformed, and partial failures occur silently. When execution deviates from the idealized "happy path," agents frequently deadlock, hallucinate success, or abandon the task altogether.

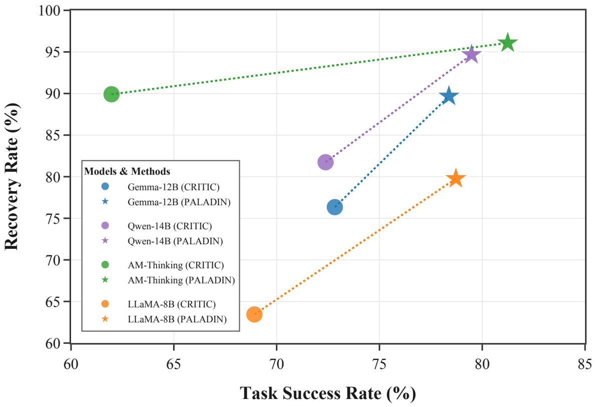


Figure 1: Recovery Rate versus Task Success Rate across backbones for CRITIC and PALADIN. Each dotted line traces the shift from a reflective baseline to PALADIN. Gains in task success closely track gains in recovery, indicating that execution-level recovery is a dominant driver of reliable performance.

This exposes a central paradox in agent design: agents are trained extensively to reason about tools, but not to recover from them. Most training pipelines optimize exclusively over successful trajectories, implicitly assuming reliable execution. As a result, execution-time failures—among the most common causes of real-world agent breakdown—are largely absent from training data. When failures occur, downstream reasoning collapses not because plans are poor, but because recovery behavior is missing.

The problem is especially acute in LLM-based multi-agent systems. In such systems, agents depend on one another's tool-mediated outputs. A single unhandled failure can propagate across agents, corrupt shared state, and derail coordination. Reliable agentic systems therefore require more than accurate planning: they require *execution-level robustness*, the ability to detect, diagnose, and recover from failures as they arise.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Existing approaches address this problem only partially. Reflective and critic-based methods such as ToolReflect (Polyakov et al. 2025) and CRITIC (Zheng et al. 2023) revise individual tool calls after errors occur. While effective in narrow cases, these approaches remain local and reactive. They do not train agents on full recovery trajectories, nor do they yield reusable strategies that generalize across failure modes or multi-step execution.

We argue that recovery must be treated as a first-class learning objective. We introduce **PALADIN**, a framework that teaches language agents explicit, trajectory-level recovery behavior. During training, PALADIN systematically injects realistic execution failures aligned with the ToolScan taxonomy and pairs them with expert recovery demonstrations. This supervision teaches structured behaviors such as diagnosis, replanning, tool substitution, and graceful termination, rather than relying on ad hoc retries or post-hoc correction.

At inference time, PALADIN augments execution with a lightweight retrieval mechanism. When a failure is detected, the agent conditions its response on a small, curated bank of failure–recovery exemplars. This grounds recovery in reusable abstractions, enabling generalization across tools, APIs, and unseen failure conditions without excessive retries.

Figure 1 highlights the core empirical pattern motivating this work. Across models, improvements in recovery capability translate directly into higher task success. Agents that recover reliably complete more tasks—not because they plan better, but because they survive execution failures that derail baseline systems.

We evaluate PALADIN using a deterministic tool-use simulator with controlled failure injection and introduce recovery-focused metrics that directly measure robustness and safety. Across multiple backbones and benchmarks, PALADIN substantially improves recovery and task success with only modest efficiency overhead, and generalizes to unseen tools and failure types.

By making recovery explicit, learnable, and measurable, PALADIN provides a practical foundation for reliable tool-augmented agents and a concrete step toward the robust multi-agent systems.

Our contributions are:

- **Execution-level recovery as a learning objective**, formalized as a trajectory-level capability.
- **PALADIN**, a training and inference framework combining systematic failure injection with taxonomy-guided recovery supervision.
- **Recovery-focused evaluation**, including new metrics and benchmarks that directly measure robustness under execution failures.
- **Empirical generalization**, showing recovery behaviors transfer across models, tools, and unseen failure modes.

Methodology

PALADIN equips tool-augmented language agents with *execution-level robustness*: the ability to detect, diagnose, and recover from runtime tool failures. The framework consists of four components: (i) formalizing execution failures, (ii) constructing recovery-annotated trajectories via systematic failure injection, (iii) fine-tuning agents with recovery-aware supervision, and (iv) augmenting inference with taxonomy-guided retrieval. Figure 2 provides an overview.

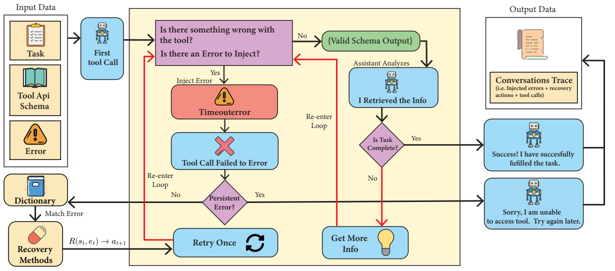


Figure 2: PALADIN's tool-use simulator with integrated failure injection and recovery. (a) Static architecture: tool calls are executed under controlled failure conditions and paired with recovery supervision. (b) Dynamic execution loop: the agent detects failures, applies recovery actions, and resumes task execution. This design enables reproducible training and evaluation of execution-level robustness.

Problem Setup

We model a tool-augmented agent as a policy π_θ that produces a trajectory $\tau = [(s_1, a_1), \ldots, (s_T, a_T)]$, where actions a_t may include tool invocations. Execution failures $f \in \mathcal{F}$ arise when tools time out, return malformed outputs, raise API errors, or produce inconsistent results. Standard training pipelines ignore such events, yielding policies that fail catastrophically when execution deviates from the "happy path."

Our objective is to learn a policy that completes tasks while remaining robust to failures:

$$\pi^* = \arg\max_{\pi} \ \mathbb{E}_{\tau \sim \pi} \big[\mathrm{TSR}(\tau) - \alpha \cdot \mathrm{CSR}(\tau)\big],$$

where TSR measures task completion and CSR penalizes hallucinated success after failure.

Failure Injection and Recovery-Annotated Data

To expose agents to realistic execution faults, we systematically augment ToolBench trajectories with failures aligned to the seven canonical error classes in the ToolScan taxonomy (Kokane et al. 2025). Each trajectory is decomposed into a task T, available tools $\mathcal{A}$, execution trace C, and error signal E. When a failure occurs ($E \neq \varnothing$), the trace is truncated at the failure point and passed to a GPT-5 teacher model, which generates an explicit recovery sequence conditioned on the failure context:

$$f_{\text{repair}}(T, \mathcal{A}, C, E) \rightarrow C'.$$

If no failure is injected, the trace is finalized unchanged:

$$f_{\text{finalize}}(T, \mathcal{A}, C) \rightarrow C'.$$

This process yields over 50,000 recovery-annotated trajectories spanning retries, parameter corrections, tool substitutions, and graceful termination. Failures are injected deterministically at controlled steps, producing multiple variants of the same base task that differ only in fault conditions. This enables reproducible, apples-to-apples comparison across models while preserving identical task structure.

In addition, we curate a compact recovery dictionary of 55+ exemplar failure–recovery pairs. These exemplars operationalize the ToolScan taxonomy into reusable recovery strategies and serve as structured supervision during inference.

Training Objective

PALADIN fine-tunes base language models using a causal language modeling objective augmented with recovery supervision:

$$\mathcal{L}_{\text{PALADIN}} = \mathcal{L}_{\text{SFT}} + \lambda \, \mathcal{L}_{\text{rec}},$$

where $\mathcal{L}_{\text{SFT}}$ is the standard negative log-likelihood over successful trajectories, and $\mathcal{L}_{\text{rec}}$ applies the same objective to tokens following explicit `Recovery:` markers that denote corrective actions. All trajectories are serialized in ToolBench format. We use LoRA adapters (Hu et al. 2021; Dettmers et al. 2023; Biderman et al. 2024; Liu et al. 2024) for parameter-efficient fine-tuning, preserving base task competence while injecting recovery behavior.

Inference-Time Recovery via Taxonomic Retrieval

At inference time, PALADIN monitors tool execution and triggers recovery only when a failure is detected. Given an observed failure f_{obs}, the agent retrieves the most similar exemplar from the recovery bank $\mathcal{E} = \{(f_i, r_i)\}_{i=1}^{55}$:

$$f_{\text{ref}} = \arg\min_{f_i \in \mathcal{E}} d(f_{\text{obs}}, f_i),$$

where $d(\cdot, \cdot)$ measures similarity between error signatures. The associated recovery action r_i conditions the agent's next steps, guiding execution back to a valid trajectory.

The retrieval operation incurs negligible overhead: the exemplar bank is small, lookup is constant-time, and retrieval is executed only upon failure, not during normal reasoning. Observed efficiency tradeoffs therefore arise from deliberate recovery actions (e.g., retries or tool switching), rather than from the retrieval mechanism itself.

Metrics for Execution Robustness

We evaluate execution-level robustness using four metrics:

$$\begin{aligned}
\text{TSR} &= \frac{\#\,\text{successful tasks}}{\text{total tasks}} \\
\text{RR} &= \frac{\#\,\text{failures recovered}}{\text{failures encountered}} \\
\text{CSR} &= 1 - \frac{\#\,\text{hallucinated successes}}{\#\,\text{failures}} \\
\text{ES} &= \frac{1}{\text{average number of steps per task}}
\end{aligned}$$

RR, CSR, and ES capture recovery effectiveness, safety, and efficiency beyond task success alone. We interpret ES as a proxy for execution cost: improvements in RR and TSR that maintain bounded ES indicate robustness beyond brute-force retrying.

Experiment and Evaluation Setup

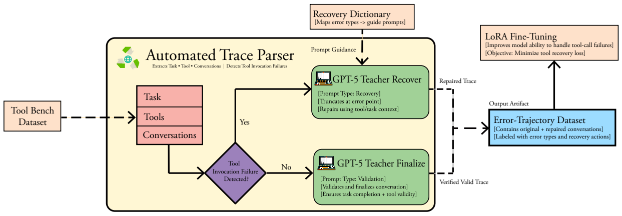

Recovery Dictionary
[Maps error types -> guide prompts]
Prompt Guidance
Automated Trace Parser
Extracts Task • Tool • Conversations | Detects Tool Invocation Failures
Tool Bench Dataset
Task
Tools
Conversations
Yes
Tool Invocation Failure Detected?
No
GPT-5 Teacher Recover
[Prompt Type: Recovery]
[Truncates at error point]
[Repairs using tool/task context]
GPT-5 Teacher Finalize
[Prompt Type: Validation]
[Validates and finalizes conversation]
[Ensures task completion + tool validity]
Repaired Trace
Verified Valid Trace
Output Artifact
Error-Trajectory Dataset
[Contains original + repaired conversations]
[Labeled with error types and recovery actions]
LoRA Fine-Tuning
[Improves model ability to handle tool-call failures]
[Objective: Minimize tool recovery loss]

Figure 3: Construction of the error-trajectory dataset. Each ToolBench trace is truncated at the first injected failure and either repaired with an explicit recovery sequence or finalized unchanged. Resulting traces are stored with failure and recovery metadata.

We evaluate PALADIN under controlled yet deployment-relevant failure conditions, focusing on execution-level robustness rather than clean-task performance alone. All experiments are conducted in a deterministic simulator to ensure reproducibility and fair comparison across methods.

Evaluation Benchmarks

We evaluate on three complementary benchmarks that together measure recovery behavior, generalization, and comparability to prior work.

PaladinEval. PaladinEval is derived from the ToolBench evaluation split and augmented with explicit, labeled execution failures. Each instance consists of a ToolBench prompt paired with a runtime failure injected by a teacher model. The resulting set includes both clean executions and failure-augmented traces, enabling joint measurement of task completion and recovery. To prevent train–test leakage, all tasks, tool schemas, and (failure class, injection step) pairs used for PaladinEval are held out from training.

Generalization Set. To assess out-of-distribution robustness, we construct a separate evaluation set by sampling unused ToolBench prompts and injecting failure types that do not appear in PALADIN's 55+ exemplar recovery dictionary. This benchmark tests whether recovery behavior generalizes beyond memorized failure–recovery pairs.

ToolReflectEval. We additionally evaluate on ToolReflectEval (Polyakov et al. 2025) to enable direct comparison with reflective baselines. We follow their protocol of allowing up to three recovery attempts per tool call and re-execute all ToolAlpaca-derived prompts inside our simulator. No prompts are modified or filtered. This ensures fidelity to the original benchmark while enabling evaluation under a unified grading framework.

Simulation Environment

All evaluations are conducted in a deterministic tool-use simulator that reproduces the ToolBench execution pipeline while adding controlled failure injection, recovery logging, and automated grading. Each episode begins with a prompt P and toolset $\mathcal{A}$. Tool calls are executed sequentially, returning either successful outputs or injected failures $f \in \mathcal{F}$. Failures arise either from PaladinEval's systematic injection protocol or from naturally error-prone ToolReflectEval traces.

At each step, the simulator records the dialogue context, observed failure signal, and recovery action. Final traces are passed to an automated grader to compute Task Success Rate (TSR), Recovery Rate (RR), Catastrophic Success Rate (CSR), and Efficiency Score (ES) under a fixed rubric. Using a single simulator and grader across all methods ensures that differences in performance reflect recovery behavior rather than evaluation artifacts.

We restrict evaluation to this deterministic setting to isolate execution-level robustness from external nondeterminism. Evaluation on live APIs is left to future deployment studies.

Automated Grading and Reproducibility

All metrics are computed using a GPT-5–based auto-grader conditioned on fixed, task-specific rubrics derived from the failure taxonomy. The grader penalizes hallucinated success, rewards valid recovery strategies, and enforces consistent metric definitions across benchmarks.

We validate grader reliability against expert human annotations on a held-out set of 120 traces spanning all ToolScan error classes. Agreement reaches 94.2%, with Cohen's $\kappa = 0.89$. To eliminate evaluation variance, grader prompts are frozen during all experiments. Observed run-to-run variation remains within $\pm 1.5\%$, which is negligible relative to reported gains. Grader prompts and scoring scripts will be released for reproducibility.

Baseline Agents

We compare PALADIN against four baselines spanning different supervision and recovery paradigms:

- **Vanilla**: the pretrained language model without additional supervision.
- **ToolBench Agent** (Qin et al. 2023): trained only on clean ToolBench trajectories, with no recovery capability.
- **ToolReflect Agent** (Polyakov et al. 2025): trained on paired valid/invalid tool calls with iterative self-correction.
- **CRITIC** (Zheng et al. 2023): a reflective agent that accesses external recovery guidance with probability $p = 0.7$.

These baselines isolate the effects of recovery supervision (clean vs. failure-rich), recovery strategy (reactive vs. proactive), and integration point (training vs. inference).

Critic-Style Oracle Baseline

To ensure a strong and conservative comparison, we implement a critic-style oracle baseline closely following the original CRITIC design. After each tool call, an oracle provides a small set of predefined recovery options corresponding to the observed error type. The agent selects and executes the strongest option, with up to three attempts allowed before terminating.

Unlike PALADIN, this baseline does not receive any recovery-focused fine-tuning and relies entirely on inference-time guidance. All critic-style baselines share the same backbone models, simulator, and grader as PALADIN, ensuring comparability.

Illustrative Failure Case. In a service-dependency failure involving multiple APIs (e.g., product retrieval followed by email validation), correct recovery requires identifying the upstream service outage, waiting or retrying appropriately, and resuming execution once the dependency stabilizes. This class of failures highlights the difference between local call correction and trajectory-level recovery.

Results

In agentic systems, tool failures rarely occur in isolation: a single execution error can cascade across planning, retrieval, and coordination modules. We therefore interpret all results through the lens of system reliability rather than isolated task accuracy. In multi-agent or modular agentic systems, these gains directly reduce failure propagation, as successful local recovery prevents downstream agents or modules from operating on corrupted state.

Recovery Is the Primary Driver of Task Success

Across models and benchmarks, improvements in recovery consistently translate into higher task success and reduced silent failure, indicating that execution-level recovery—rather than auxiliary planning or scale alone—is the dominant driver of reliable performance.

Table 1 reports results across eight model–dataset pairs. PALADIN achieves the highest Recovery Rate (RR) in 7/8 settings (average +13.6%), the highest Task Success Rate (TSR) in 6/8 settings (average +10.2%), and the best Catastrophic Success Rate (CSR) in 6/8 settings (average +9.2%). All improvements are statistically significant ($p < 0.01$).

Correlation analysis supports a mechanistic relationship between recovery and task completion. RR and TSR are strongly correlated ($r = 0.91$, $p < 0.001$), while TSR exhibits negligible correlation with efficiency ($r = 0.12$), indicating that efficiency alone is a poor proxy for reliability. CSR and Efficiency show a significant negative correlation ($r = -0.72$, CI: $[-0.79, -0.63]$), reflecting an inherent but bounded safety–efficiency tradeoff.

Efficiency reductions are therefore not incidental. They arise from deliberate recovery behavior—such as retries, validation, or tool switching—rather than uncontrolled looping. No baseline simultaneously achieves higher safety and higher efficiency, situating PALADIN near the Pareto frontier.

Table 1: Main results across models and datasets. All metrics (Recovery, Task Success, CSR, Efficiency) are normalized such that higher is better. PALADIN achieves consistently strong safety and recovery performance, with modest efficiency tradeoffs

Pretrained LLM	**Evaluation Datasets**	**Evaluation Metrics**	**Scores (↑)**				
			Vanilla	CRITIC	ToolReflect	ToolBench	Paladin (Ours)
Gemma-3 12B-Instruct	Paladin Eval	Recovery Rate	23.75%	76.34%	65.86%	32.76%	**89.68%** (+13.34%)
		Task Success Rate	23.62%	72.83%	61.42%	57.4%	**78.38%** (+5.55%)
		Catastrophic Success Rate	29.00%	73.30%	70.27%	68.37%	**82.55%** (+9.25%)
		Efficiency Score	**0.476**	0.348	0.288	0.221	0.312 (-34.45%)
	ToolReflect Eval	Recovery Rate	22.80%	73.29%	63.23%	31.45%	**86.09%** (+12.80%)
		Task Success Rate	22.56%	69.55%	58.66%	54.82%	**83.45%** (+13.90%)
		Catastrophic Success Rate	26.16%	72.23%	69.08%	67.10%	**81.85%** (+9.62%)
		Efficiency Score	**0.508**	0.370	0.307	0.235	0.332 (-34.65%)
Qwen-2.5-14B-Instruct	Paladin Eval	Recovery Rate	37.68%	81.74%	73.66%	33.48%	**94.67%** (+12.93%)
		Task Success Rate	37.53%	72.38%	74.36%	60.41%	**79.48%** (+5.12%)
		Catastrophic Success Rate	56.53%	87.14%	83.85%	67.88%	**94.57%** (+7.43%)
		Efficiency Score	0.313	0.329	0.312	0.339	**0.351** (+3.54%)
	ToolReflect Eval	Recovery Rate	36.17%	78.47%	70.71%	32.14%	**92.88%** (+14.41%)
		Task Success Rate	35.85%	69.13%	71.02%	57.70%	**75.19%** (+4.17%)
		Catastrophic Success Rate	54.88%	81.63%	83.20%	66.60%	**94.35%** (+11.15%)
		Efficiency Score	0.334	0.350	0.333	0.361	**0.375** (+3.73%)
AM-Thinking V1	Paladin Eval	Recovery Rate	49.87%	89.91%	87.23%	51.37%	**96.08%** (+6.17%)
		Task Success Rate	52.93%	61.97%	72.88%	56.83%	**81.24%** (+8.36%)
		Catastrophic Success Rate	60.24%	81.33%	71.32%	80.84%	**88.65%** (+7.32%)
		Efficiency Score	0.415	**0.420**	0.297	0.319	0.380 (-9.52%)
	ToolReflect Eval	Recovery Rate	47.88%	86.31%	83.74%	49.32%	**92.24%** (+5.93%)
		Task Success Rate	50.55%	78.64%	79.41%	65.25%	**77.98%** (-1.43%)
		Catastrophic Success Rate	62.65%	67.42%	65.93%	58.06%	**88.31%** (+20.89%)
		Efficiency Score	0.442	**0.448**	0.316	0.340	0.405 (-9.60%)
Llama-3.1-8B-Instruct	Paladin Eval	Recovery Rate	21.83%	63.44%	56.32%	49.2%	**79.77%** (+16.33%)
		Task Success Rate	17.46%	68.92%	53.74%	47.26%	**78.72%** (+9.80%)
		Catastrophic Success Rate	17.58%	71.84%	67.88%	65.47%	**80.73%** (+8.89%)
		Efficiency Score	**0.427**	0.254	0.287	0.209	0.323 (-24.36%)
	ToolReflect Eval	Recovery Rate	18.32%	58.55%	49.32%	42.23%	**73.34%** (+14.79%)
		Task Success Rate	13.45%	59.47%	46.34%	41.22%	**71.27%** (+11.80%)
		Catastrophic Success Rate	15.40%	63.81%	58.09%	53.68%	**71.77%** (+7.96%)
		Efficiency Score	**0.568**	0.360	0.508	0.412	0.385 (-32.31%)

Recovery Generalizes to Unseen Failures

PALADIN's recovery behavior generalizes beyond the failure types and conditions observed during training, maintaining strong performance under out-of-distribution execution faults.

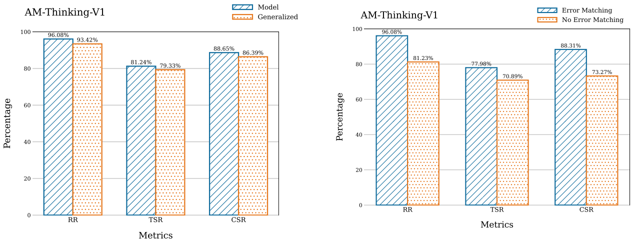


Figure 4: PALADIN's performance without inference-time error matching compared to baselines across Gemma, Qwen, LLaMA, and AM-Thinking backbones. Full results are provided in the supplementary material.

Figure 4 reports results on unseen failure types across all backbones. Despite architectural differences, PALADIN consistently achieves $> 79\%$ RR, $> 78\%$ TSR, and $> 80\%$ CSR. Notably, LLaMA-3.1-8B improves from 21.8% to 79.8% RR and from 17.5% to 78.7% TSR, reducing silent failure by over 60 percentage points. Qwen-2.5-14B attains the highest CSR (94.6%), while AM-Thinking-V1 achieves the highest RR (96.1%).

These results demonstrate that recovery is not memorized at the level of individual error cases. Instead, PALADIN learns reusable recovery strategies that transfer across unseen failures, tools, and model families.

Inference-Time Retrieval Is the Mechanism Enabling Robustness

Ablation experiments identify inference-time retrieval as a critical mechanism underlying PALADIN's robustness. Retrieval does not introduce new reasoning steps; it conditions existing recovery behavior on structured failure exemplars.

Removing exemplar-based error matching sharply degrades recovery across all models (Figure 4). For example, Gemma-12B's RR drops from 89.7% to 61.4%, TSR from 87.4% to 57.3%, and CSR from 82.6% to 65.1%. Similar declines are observed for Qwen-14B (RR -21.4%), LLaMA-8B (RR -31.2%), and AM-Thinking-V1 (RR -14.9%).

The magnitude of these drops—often 20–30 points—indicates that training-time exposure to failures alone is insufficient. These results indicate that inference-time retrieval is not a performance booster but a structural component: without it, recovery policies fail to activate reliably under execution faults.

Implications for Agent Reliability

Taken together, these results show that execution-level robustness is a learnable and transferable capability. PALADIN consistently improves recovery, safety, and task success across models, generalizes to unseen failures, and exhibits predictable safety–efficiency tradeoffs driven by intentional recovery behavior. These findings establish recovery as a first-class optimization objective for tool-augmented agents.

Related Work

Execution-Level Robustness

A growing body of work seeks to improve the robustness of tool-augmented language agents under execution-time failures. However, most existing approaches treat failures as *local anomalies*—events to be patched after they occur—rather than as *trajectory-level operating conditions* that must be modeled and learned explicitly.

ToolReflect (Polyakov et al. 2025) improves tool-call quality by contrasting invalid and corrected calls through supervised fine-tuning. While effective for call-level correction, this formulation does not model recovery as a multi-step process: failures are addressed in isolation rather than as part of a cascading execution trace that may require diagnosis, replanning, or tool substitution. Related work on LLM runtime error handling (Sun et al. 2024) and structured exception handling (Zhou et al. 2025) introduces systematic mechanisms for error detection, but stops short of training agents on full recovery trajectories.

Critic-style approaches (Zheng et al. 2023) introduce an external evaluator that critiques and revises agent actions, improving reliability through iterative self-correction. These methods remain fundamentally reactive: feedback is applied post hoc, recovery quality depends on inference-time judgment, and recovery strategies are not internalized as reusable behaviors.

In contrast, PALADIN treats recovery as a first-class learning objective. By training directly on failure-rich trajectories that include diagnosis, replanning, and multi-turn repair, PALADIN learns reusable recovery policies that operate at the level of execution traces rather than individual calls.

Diagnostics and Benchmarks

Several efforts focus on characterizing tool-use failures rather than resolving them. ToolScan (Kokane et al. 2025) introduces a taxonomy of seven recurring error patterns in LLM tool use, providing a diagnostic lens on where and how agents fail. However, ToolScan is explicitly descriptive: it categorizes failures but does not curate executable recovery trajectories or provide supervision for learning recovery behavior.

BugGen argues that handcrafted or mutation-based bug synthesis yields narrow coverage of real-world failures, motivating realistic failure-rich datasets. Together, these works establish the *structure* of agent failures but leave open how agents should be trained to recover during execution. PALADIN bridges this gap by operationalizing failure taxonomies into recoverable trajectories that directly supervise recovery behavior.

Prior Tool Systems and Interface Reliability

Foundational systems such as Toolformer, Gorilla, ToolLLM, ToolBench (Qin et al. 2023; Patil et al. 2023; Schick et al. 2023; Guo et al. 2024), and RoTBench advanced tool selection, formatting, and multi-step planning, but largely assume clean execution or focus on pre-execution uncertainty. As a result, these systems are optimized for "happy-path" trajectories in which tool calls succeed as intended.

When execution errors occur, prior systems typically lack explicit recovery strategies, leading to brittle deployment behavior. ToolBench provides broad tool coverage but no runtime recovery protocols, while RoTBench probes robustness via documentation noise rather than execution faults. Complementary tools such as ToolFuzz (Milev et al. 2025) improve interface reliability before execution, but do not address failures that arise during tool interaction.

PALADIN targets this missing layer: runtime execution breakdowns that occur despite correct tool selection and formatting. By explicitly modeling, training on, and evaluating recovery under such failures, PALADIN reframes robustness as an execution-time competence rather than a pre-execution filtering problem.

Discussion

The results show that execution-level recovery is a prerequisite for reliable agentic systems: when failures are not handled explicitly, errors propagate across reasoning steps and—by extension—across agents in multi-agent settings. We interpret PALADIN's gains not as isolated performance improvements, but as evidence for concrete design principles for building failure-aware agents.

Importantly, PALADIN does not aim to eliminate tool failures. Instead, it treats failure as an expected operating condition and optimizes recovery behavior accordingly.

Recovery as a Teachable System Capability

PALADIN demonstrates that recovery from tool failures is not an emergent side effect of scale or reflection, but a capability that can be explicitly taught. Unlike prior approaches such as ToolReflect or CRITIC, which rely on local post-hoc corrections, PALADIN trains agents on full recovery trajectories that include failure detection, diagnosis, and multi-step repair. This reframes robustness as a system-level behavior rather than a patch applied to individual tool calls.

From a design perspective, this suggests that robust agents should be trained not only on what successful execution looks like, but on how execution breaks and how to resume progress afterward. Recovery supervision functions as a control layer over execution, enabling agents to remain operational under non-ideal conditions rather than collapsing when assumptions are violated.

Generalization via Failure Abstraction

PALADIN's ability to generalize across tools and unseen failure modes follows from a simple engineering principle: recovery policies should be conditioned on *failure structure*, not tool identity. By aligning supervision with a failure taxonomy, PALADIN learns reusable recovery behaviors (e.g., retry with backoff, reformat outputs, substitute tools) that apply across heterogeneous APIs.

Practically, this means that robustness does not require memorizing tool-specific fixes. A compact set of failure abstractions, paired with exemplar recoveries, is sufficient to cover a broad tool surface. This keeps recovery infrastructure small and stable even as the number of tools or agents grows—an important consideration for scalable agent systems. Because recovery is conditioned on failure structure rather than tool identity, the supporting exemplar bank remains compact and stable, scaling with the diversity of failure classes rather than the number of tools, tasks, or agents.

Failure Injection as Robustness-Oriented Training

Systematic failure injection serves as a targeted mechanism for teaching agents how to operate under adverse execution conditions. Rather than relying on rare naturally occurring failures or synthetic noise, PALADIN injects canonical failure types at controlled points in execution. This exposes agents to realistic breakdowns while preserving task structure.

Viewed as an engineering strategy, failure injection provides coverage guarantees: agents encounter the failure modes they are most likely to face in deployment. The resulting behavior is not merely tolerant of errors, but structured—agents learn when to retry, when to replan, and when to terminate gracefully. The modest efficiency overhead observed reflects intentional recovery actions rather than uncontrolled retries.

Managing the Safety–Efficiency Tradeoff

The observed tradeoff between efficiency and robustness reflects an inherent property of failure-aware systems: recovering from errors incurs cost, but avoiding recovery incurs silent failure. PALADIN operates near this tradeoff boundary by converting small increases in execution steps into large gains in task success and catastrophic failure avoidance.

For system designers, this suggests that efficiency should be treated as a constrained resource rather than a primary objective. In deployment, bounded inefficiency may be preferable to brittle speed. PALADIN's behavior makes this tradeoff explicit and measurable, enabling future systems to adapt recovery intensity based on task criticality or latency constraints.

Limits of Deterministic Evaluation

All results in this work are obtained in a deterministic simulator, which isolates recovery behavior from external nondeterminism and enables controlled comparison. While real APIs introduce stochastic delays and evolving interfaces, the underlying failure modes—timeouts, malformed outputs, partial execution—remain structurally consistent.

We therefore view deterministic evaluation as a necessary abstraction for studying execution-level robustness. Live deployment studies are a complementary step, but not a prerequisite for establishing recovery as a learnable system capability.

Implications for Multi-Agent Systems

Although PALADIN is evaluated in a single-agent setting, its abstractions directly extend to multi-agent systems. In such systems, tool-mediated failures are amplified: an unhandled error in one agent can corrupt shared state or derail downstream agents.

PALADIN's recovery policies and exemplar bank can be shared across agents, providing a common language for diagnosing and repairing execution failures. This reduces the likelihood of cascading breakdowns and supports more stable coordination. From a multi-agent system perspective, this positions execution-level recovery as foundational infrastructure for scalable and reliable multi-agent systems, rather than an optional enhancement.

Deployment and Future Directions

PALADIN provides a modular pathway toward failure-aware agents that can operate under realistic conditions. Its separation of recovery supervision, exemplar retrieval, and base task competence makes it compatible with existing agent architectures and scaling trends.

Future work may explore adaptive recovery policies that modulate retry behavior based on task criticality, confidence, or system load, as well as online incorporation of failure traces from deployment logs. More broadly, PALADIN highlights a shift in how robustness should be treated: not as an emergent property of better planning, but as an explicit, trainable dimension of agent behavior.

Conclusion

PALADIN demonstrates that execution-level robustness is not emergent but a *teachable, scalable capability*. By unifying recovery-rich fine-tuning with inference-time exemplar retrieval, it transforms brittle, "happy-path" agents into resilient problem-solvers. Across backbones, PALADIN achieves $> 79\%$ RR, $> 78\%$ TSR, and $> 80\%$ CSR, while keeping efficiency penalties within one step. These contributions extend beyond benchmarks. PALADIN shows recovery behaviors generalize to unseen failures, proving robustness is not tied to specific error types but can be abstracted into transferable policies.

The implications are twofold. For research, robustness supervision provides a new axis for evaluation and learning, complementary to scaling and alignment. For practice, PALADIN enables safer deployment of LLM agents in high-stakes domains where silent failures are unacceptable. Future work should explore adaptive controllers that modulate retry intensity based on task difficulty or model confidence, and integration with error logs from production environments.

In short, PALADIN sets a new benchmark for execution-level resilience: robustness can be systematically taught, generalized across models, and achieved without prohibitive cost—laying the groundwork for safe, failure-aware AI systems. More broadly, we view PALADIN as a building block for safer LLM-based multi-agent systems: equipping each agent with robust, instrumented recovery behavior reduces the chance that a single tool fault silently corrupts shared state, enabling more stable planning and collaboration in multi-agent workflows.

References

Biderman, D.; Portes, J.; Ortiz, J. J. G.; Paul, M.; Greengard, P.; Jennings, C.; King, D.; et al. 2024. LoRA Learns Less and Forgets Less. arXiv:2405.09673.

Dettmers, T.; Pagnoni, A.; Holtzman, A.; and Zettlemoyer, L. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. arXiv:2305.14314.

Guo, Z.; Cheng, S.; Wang, H.; Liang, S.; Qin, Y.; Li, P.; et al. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. arXiv:2403.07714.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685.

Kokane, S.; Zhu, M.; Awalgaonkar, T.; Zhang, J.; Hoang, T.; Prabhakar, A.; Liu, Z.; Lan, T.; Yang, L.; Tan, J.; Murthy, R.; Yao, W.; Liu, Z.; Niebles, J. C.; Wang, H.; Heinecke, S.; Xiong, C.; and Savarese, S. 2025. ToolScan: A Benchmark for Characterizing Errors in Tool-Use LLMs. arXiv:2411.13547.

Liu, Z.; Lyn, J.; Zhu, W.; Tian, X.; and Graham, Y. 2024. ALoRA: Allocating Low-Rank Adaptation for Fine-tuning Large Language Models. arXiv:2403.16187.

Milev, I.; Balunović, M.; Baader, M.; and Vechev, M. 2025. ToolFuzz – Automated Agent Tool Testing. arXiv:2503.04479.

Patil, S. G.; Zhang, T.; Wang, X.; and Gonzalez, J. E. 2023. Gorilla: Large Language Model Connected with Massive APIs. arXiv:2305.15334.

Polyakov, G.; Alimova, I.; Abulkhanov, D.; Sedykh, I.; Bout, A.; Nikolenko, S.; and Piontkovskaya, I. 2025. ToolReflection: Improving Large Language Models for Real-World API Calls with Self-Generated Data. In Kamalloo, E.; Gontier, N.; Lu, X. H.; Dziri, N.; Murty, S.; and Lacoste, A., eds., *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)*, 184–199. Vienna, Austria: Association for Computational Linguistics. ISBN 979-8-89176-264-0.

Qin, Y.; Liang, S.; Ye, Y.; Zhu, K.; Yan, L.; Lu, Y.; et al. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:2307.16789.

Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761.

Sun, Z.; Zhu, H.; Xu, B.; Du, X.; Li, L.; and Lo, D. 2024. LLM as Runtime Error Handler: A Promising Pathway to Adaptive Self-Healing of Software Systems. arXiv:2408.01055.

Zheng, R.; Dou, S.; Gao, S.; Hua, Y.; Shen, W.; Wang, B.; et al. 2023. Secrets of RLHF in Large Language Models Part I: PPO. arXiv:2307.04964.

Zhou, J.; Chen, J.; Lu, Q.; Zhao, D.; and Zhu, L. 2025. SHIELDA: Structured Handling of Exceptions in LLM-Driven Agentic Workflows. arXiv:2508.07935.

Supplementary Material

(for: *PALADIN: Self-Correcting Language Model Agents to Cure Tool-Failure Cases*)

Anonymous

11/02/2025

A Critic-Style Approach

We implemented a critic-style agent baseline with an oracle-assisted loop as a benchmark for comparison to PALADIN. The critic module (Gemma-12B) was invoked after each tool call to detect execution errors and propose recovery actions. Whenever an error was flagged, the executor was directed to an oracle dataset with pre-defined recovery actions for the detected error type. The most appropriate action among them was then selected to continue the trajectory. A maximum of 3 recovery attempts were permitted per error before execution continued with the latest attempt or failed gracefully.

Our implementation differs from prior CRITIC systems in that, rather than requiring the model to autonomously generate new candidate recovery actions or perform external reasoning (e.g., web search), we provide a structured set of recovery actions from a curated oracle dataset. The model's role was to choose the most appropriate recovery, rather than generate one from scratch.

Evaluation Example:

Chosen error: ServiceDependencyFailure
Justification: The request involves two services: fetching hot products from AliExpress and validating email domains. A realistic failure is that one upstream service is temporarily unavailable. The model must recognize this is an external failure, not due to malformed input.
Expected recovery actions: Identify the failing service, wait until it is healthy, and retry the request.

B ToolFuzz

While PALADIN focuses on runtime robustness during tool execution, complementary work such as ToolFuzz addresses pre-execution reliability by improving the alignment between tool documentation and model expectations.

ToolFuzz applies automated fuzz testing to API schemas in order to detect inconsistencies between declared documentation and actual behavior. It uncovered over 20x more specification-related failures than prompt engineering baselines across 32 LangChain and 35 custom tools—revealing widespread documentation under specification as a root cause of tool-use errors.

Although PALADIN assumes that documentation is accurate at test time (as each tool is provided with correct instructions), ToolFuzz supports our broader vision: robust real-world tool use requires not only runtime adaptability, but also upstream validation of the tool interfaces themselves. We view ToolFuzz as complementary infrastructure—ensuring that PALADIN's recovery logic is exercised on meaningful failures, not avoidable documentation bugs.

C Runtime Errors

The following are some of the many common errors found in API faults, tool calls, or in daily life. For the rest of the errors used to train this model, check out our github: `https://anonymous.4open.science/r/PALADIN-Framework/README.md/`.

Specific Failure	Example (Simulated Output)	Corrective Action Policy	Rationale and Citations
400 Bad Request	`"error": "Malformed request syntax", "status":`	Re-examine tool documentation, check parameter formatting, and re-issue the call.	The error originates from the client's request. The only path to recovery is for the client (the agent) to correct its own mistake.
401 Unauthorized	`"error": "Invalid API key provided", "status":`	Check for a valid API key. If missing or invalid, terminate the task and report the failure.	A non-recoverable authentication error; retrying with invalid credentials is futile.
403 Forbidden	`"error": "User does not have permission for this resource", "status":`	Terminate the task and report the lack of permissions. Do not retry.	This is an authorization failure. Retrying will not change permissions.
404 Not Found	`"error": "The requested resource does not exist", "status":`	Verify the request URL. If correct, assume the resource is unavailable and try an alternative tool.	A common error caused by typos or moved resources. Check self-error first.
500 Internal Server Error	`"error": "Unexpected server error", "status":`	Retry using exponential backoff. If failure persists after 3–4 attempts, terminate and report.	A catch-all for transient server-side issues. Retrying is industry standard.
503 Service Unavailable	`"error": "Service unavailable due to overload or maintenance", "status":`	Follow the Retry-After header if present; otherwise, use exponential backoff.	Retry is expected, as the issue is temporary.
Request Timeout	`requests.exceptions.Timeout`	Retry with exponential backoff; distinguish connection vs. read timeouts.	Timeouts are usually transient and should be retried.
DNS Resolution Error	`requests.exceptions.ConnectionError: getaddrinfo_failed`	Check hostname for typos. If correct, wait and retry.	Could be a typo or a temporary DNS failure.
429 Too Many Requests	`"error": "Rate limit exceeded", "status":`	Respect the Retry-After header or apply exponential backoff.	Standard API rate enforcement. Ignoring leads to blocking.
Malformed JSON	`SyntaxError:JSON.parse_error`	Retry first. If repeated, use a lenient parser or fallback tool.	Often due to truncated or corrupted responses. Retry is the simplest fix.
JSON Schema Violation	`ValidationError: ’abc’is_not_of_type ’number’`	Report a data quality issue. Coerce if allowed, otherwise discard.	Data violates schema expectations. Must be handled gracefully.

Table 1: Catalog of common failures, simulated outputs, and recovery strategies.

D Systematically Augmented Traces

Systematically augmented traces are clean ToolBench-style agents–tool execution logs enriched with controlled, labeled failures and paired with recovery trajectories. This enables reproducible training and evaluation of different recovery behaviors under realistic tool-level faults. The method supports generating deterministic variants of a base rollout by injecting specific error types at different steps, with structured annotations for failure detection, diagnosis, and correction.

Augmentations were created using a failure-policy dictionary over baseline rollouts, injecting errors such as timeouts, 5xx responses, rate limits, malformed tool outputs, and partial results, with configurable positions and cascades. Each injected failure was paired with an annotated recovery path (diagnose, replan, parameter fix, retry, or tool-switch), enabling trajectory-level supervision and deterministic evaluations via metrics like TSR, RR, CSR, and ES. Because variants are deterministic, they support controlled ablations and consistent comparisons across model sizes and training settings.

- Base trace: plan → call Tool A with parameters p → receive valid response → continue.
- Augmented variant: identical plan → call Tool A with p → injected timeout at step t → agent detects timeout → backs off with jittered retry and parameter validation → successful response → proceed to completion. A second variant might inject a malformed JSON response instead of a timeout, prompting schema validation, selective re-query, or tool-switch. Each variant includes labels for failure type, step index, recovery rationale, and final outcome, enabling targeted training and evaluation.

Listing 1: Paladin trace with recovery and Finish

```
[
  {
    "role": "system",
    "content": [
      {
        "type": "text",
        "text": "You are Paladin, an error-resilient agent ..."
      }
    ]
  },
  {
    "role": "user",
    "content": [
      {
        "type": "text",
        "text": "I’m conducting a research project on social media influencers ..."
      }
    ]
  },
  {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "Thought: I need to gather user information ...
Action: userinfo_for_instagram_cheapest
Action Input: {\n \"username\": \"nike\"\n}"
      }
    ]
  },
  {
    "role": "function",
    "content": [
      {
        "type": "text",
        "text": "{\"error\": \"\", \"response\": \"{’data’: {’user’: {’biography’:
        ’Spotlighting athlete* ..."}}"
      }
    ]
  },
  {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "Recovery: Thought: The API call to retrieve user information ...
Action: user_medias_for_instagram_cheapest
Action Input: {\"user_id\": \"113294420064920\"}"
      }
    ]
  },
  {
    "role": "function",
    "content": [
      {
        "type": "text",
        "text": "{\"error\": \"\", \"response\": \"{\\\"data\\\":   [
        {\\\"media_id\\\": \\\"1234567890\\\",
\\\"caption\\\": \\\"Just Do It\\\" ...\"}"
      }
    ]
  },
  {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "Thought: I have successfully retrieved the recent media posts ...
Action: Finish
Action Input: {\"return_type\": \"give_answer\", \"final_answer\":
\"User information and recent
media posts for ’nike’ ...\"}"
      }
    ]
  }
]
```

The full corpus, schema, and documentation will be publicly released upon acceptance.

E Recovery Dictionary

At the core of PALADIN's learning process is a recovery dictionary—a curated collection of over 50 execution-level failure types and corresponding recovery strategies. This dictionary was not arbitrarily constructed. Instead, it was built through an extensive review of real-world sources, including:

- Developer forums (e.g., Stack Overflow, GitHub Issues)
- Toolchain documentation (e.g., LangChain, Zapier, API reference manuals)
- Industry engineering blogs detailing agent failure cases (e.g., Google's Bard, OpenAI function calls)
- Academic surveys and benchmarks (e.g., ToolScan, ShieldA, Healer)

We structured our final recovery dictionary around ToolScan's taxonomy of common agentic failures, enriched with patterns synthesized from forums, documentation, and engineering logs. Each error type in the dictionary is paired with recovery actions grounded in real debugging and fallback strategies reported by practitioners.

These error-response mappings were then transformed into example trajectories using the ToolBench format, providing PALADIN with richly annotated failure contexts and realistic correction paths. By rooting this dictionary in actual agent failure logs and developer strategies, we ensured the training data reflected authentic, actionable recovery behavior, not synthetic artifacts.

F ToolScan Error Types

Tool Hallucination → tool not found errors
Argument Hallucination → missing or invalid tool input
Invalid Tool Invocation → tool call failed
Partial Execution → incomplete tool outputs requiring continuation
Output Hallucination → tool output not valid
Invalid Intermediate Reasoning → corrected plans and backtracking in multi-turn dialogues
Re-entrant Failures → handled via explicit retry and fallback logic

G Python Dictionary

Here is part of our dictionary:

Listing 1: Recovery Paths Dictionary (wrapped for two-column layout)

```
recovery_paths = {

  # Branch 1: Client Request Syntax and Format Errors (400, 422)
  "400_422": [
    {
      "from": "Assistant",
      "value": (
        "Thoughts: The 400 Bad Request and 422 Unprocessable Entity "
        "errors indicate client-side issues. 400 often means malformed "
        "syntax or invalid request formatting; 422 means the syntax is "
        "correct but the content is semantically invalid.\n\n"
        "Action: Start by examining the request URL and endpoint "
        "carefully. Check for typos, extra slashes, or illegal characters "
        "that violate URL standards. Use URL validators or linters. Then, "
        "validate your payload format; if JSON, ensure it is well-formed, "
        "properly bracketed, and matches the API schema. Check required "
        "fields and types.\n\n"
        "Verify that all required headers like Content-Type and "
        "Authorization are present and correct. For 422, focus on "
        "semantic validation such as correct value ranges, formats, and "
        "dependencies."
      )
    },
    {
      "from": "function",
      "value": "Validated URL and headers; payload verified with no syntax errors."
    },
    {
      "from": "Assistant",
      "value": (
        "Thoughts: If syntax is good, server-side validation may enforce "
        "strict formats or rules not obvious in docs.\n\n"
        "Action: Test with minimal payloads including only required "
        "fields; gradually add optional fields to isolate problematic "
        "data. Confirm API version matches docs, and check server logs "
        "if available."
      )
    },
    {
      "from": "function",
      "value": "Minimal payload testing performed; error persists."
    },
    {
      "from": "Assistant",
      "value": (
        "Thoughts: Persistent error suggests possible issues with query "
        "parameters or headers.\n\n"
        "Action: Verify all query parameters for correctness and allowed "
        "values. Confirm proper encoding (UTF-8) and escaping of special "
        "characters. Check multipart boundaries if uploading files."
      )
    },
    {
      "from": "function",
      "value": "Query parameters and encoding verified."
    },
    {
      "from": "Assistant",
      "value": (
        "Thoughts: If no client-side issue found, consider API bugs or "
        "server incompatibility.\n\n"
        "Action: Contact support with detailed logs; implement retry with "
        "exponential backoff."
      )
    },
    {
      "from": "function",
      "value": "Escalation logs prepared; retry logic implemented."
    }
  ],

  # Branch 2: Authentication and Authorization Errors (401, 403, 407)
  "401_403_407": [
    {
      "from": "Assistant",
      "value": (
        "Thoughts: 401 Unauthorized indicates missing or invalid "
        "credentials, 403 Forbidden means access is denied even if "
        "authenticated, and 407 Proxy Authentication Required means the "
        "proxy server requires authentication.\n\n"
        "Action: Check that valid API keys or tokens are sent in headers "
        "exactly as required by the service. Refresh tokens if expired. "
        "Confirm permissions on the user or API key to access requested "
        "resources.\n\n"
        "For 403, verify role-based access controls and resource-specific "
        "permissions. For 407, ensure proxy credentials are configured "
        "and included properly in requests."
      )
    },
    {
      "from": "function",
      "value": "Credentials and permissions checked; tokens refreshed where needed."
    },
    {
      "from": "Assistant",
      "value": (
        "Thoughts: Persistent 401/403 might be due to time skew causing "
        "token rejection or revoked credentials.\n\n"
        "Action: Sync system clock with NTP, regenerate credentials, and "
        "check for IP restrictions or rate limits that may block access."
      )
    },
    {
      "from": "function",
      "value": "System time synchronized; no IP blocks detected."
    },
    {
      "from": "Assistant",
      "value": (
        "Thoughts: If proxy authentication fails, check proxy server logs "
        "and credentials.\n\n"
        "Action: Update proxy config or switch to a proxy that supports "
        "needed authentication methods."
      )
    },
    {
      "from": "function",
      "value": "Proxy credentials updated; connection succeeded."
    }
  ],

}
```

The rest of our dictionary can be found in github at `https://anonymous.4open.science/r/PALADIN-Framework/README.md`.

H Implementation Details

H.1 Model Instantiations

PALADIN was fine-tuned on multiple open-weight backbones to test cross-architecture generality: Gemma-27B, Qwen-2.5-14B-Instruct, AM-Thinking V1, and LLaMA-3.1-8B-Instruct. All models were trained on the full 50K recovery-annotated corpus, enabling direct comparison of robustness transfer across scales and inductive biases. For LLaMA-3.1-8B, extended context via RoPE scaling ensured parity on long recovery traces relative to Gemma and Qwen, avoiding truncation artifacts.

H.2 LoRA Configuration

All runs adopted LoRA adapters with rank 16, scaling $\alpha = 32$, and dropout 0.0. Adapters were injected into attention projections (`q_proj`, `k_proj`, `v_proj`, `o_proj`) and MLP projections (`up_proj`, `down_proj`, `gate_proj`), equipping models with recovery skills while preserving base competence.

H.3 Optimizer and Scheduler

Training employed paged AdamW (32-bit) with `bf16` precision and gradient checkpointing. We used a base learning rate of 2×10^{-4}, with standard AdamW defaults for warmup and weight decay, and a constant schedule over one epoch, appropriate for single-epoch SFT with LoRA.

H.4 Batching, Context, and Epochs

Experiments used a context length of 8192 tokens. Training was performed with micro-batch size 1 and gradient accumulation 8 (effective batch size 8), for a single epoch over 50K trajectories (80% failure-rich, 20% clean "happy paths"). This composition balanced recovery supervision with baseline tool-use competence.

H.5 Hardware and Runtime

All fine-tuning ran on `h200sxm` GPUs. The combination of `bf16`, paged AdamW, and checkpointing enabled stable 8K-token SFT with LoRA on these accelerators. RoPE scaling extended LLaMA-3.1-8B's effective context to 128K tokens with modest additional memory cost, remaining tractable under `h200`-class footprints. End-to-end fine-tuning completed within a single epoch per backbone, with wall-clock time increasing monotonically with parameter count (8B < 14B < 27B).

I Dataset Construction Details

I.1 Failure Injection Procedure

We began from ToolBench tasks and tool schemas, discarding original rollouts, and applied an automated trace parser to detect the first execution error. Each trajectory was truncated at that failure point to create a repair target. For every truncated trace, a controller supplied the task, tool schema, error signal, and dialogue context to a GPT-5 Teacher equipped with a recovery dictionary. The Teacher then generated multi-turn `Recovery:` segments consisting of retries, reformulations, fallbacks, or graceful termination, producing repaired trajectories:

$$f_{\text{repair}}(T, A, C, E) \quad \rightarrow \quad C',$$

while error-free traces were finalized via:

$$f_{\text{finalize}}(T, A, C) \quad \rightarrow \quad C'.$$

The simulator injected failures deterministically by specifying error type, manifestation (e.g., malformed output, silent failure), and turn index. A Python controller executed each scenario by providing tool documentation, applying the designated error, and simulating tool responses to the agent's recovery actions.

Injected error examples.

- **Timeouts and 5xx/503:** Transient server failures triggering capped backoff-and-retry before graceful termination.
- **Malformed/invalid outputs:** Truncated JSON, schema violations, or null fields designed to elicit re-queries, lenient parsing, or tool switching.
- **Auth/permission errors (401/403/407):** Non-recoverable or credentials-refresh scenarios; repeated failures prompted terminate-with-explanation policies.

I.2 Recovery Annotation Process

Recovery supervision combined two sources: (i) a curated recovery dictionary aligned to ToolScan's seven error classes, and (ii) GPT-guided rewriting conditioned on truncated traces and error signals. For each failure, the Teacher expanded dictionary-level strategies into situated multi-turn recoveries in ToolBench format (`Thought` → `Action` → `Action Input` → `Tool Output`), prefixing corrective steps with `Recovery:` tags and concluding with `Finish` plus either a user-facing answer or graceful-failure explanation. Clean "happy path" traces (about 20%) were also audited and, when necessary, rewritten to ensure fully successful interactions, preserving base competence while keeping recovery central in the remaining 80% of traces.

I.3 Recovery Exemplar Distribution

PALADIN's retrieval bank contains over 55 recovery exemplars derived from the dictionary and aligned to ToolScan's seven canonical error types (see (see Appendix E)). These cover *Tool Hallucination*, *Argument Hallucination*, *Invalid Tool Invocation*, *Partial Execution*, *Output Hallucination*, *Invalid Intermediate Reasoning*, and *Re-entrant Failures*, each paired with exemplar failures and recovery protocols. During execution, observed failures are matched to the closest exemplar via signature distance d, with the associated recovery action steering trajectories back to stability. This enables generalization across diverse failure surfaces.

I.4 Data Split

The final corpus comprises roughly 50K trajectories serialized in ToolBench format with explicit `Recovery:` tags, with an 80/20 composition of recovery-rich to clean traces. Training sequences were processed under a single-epoch CLM SFT regime. Evaluations were conducted in a sandboxed environment with deterministic error injection to ensure controlled, reproducible assessment using TSR, RR, CSR, and Efficiency metrics. For LLaMA-3.1-8B, RoPE scaling extended effective context to 128K tokens, preventing truncation of long recovery trajectories and ensuring parity across backbones under the same split design.

J PaladinEval Benchmark

PaladinEval is a deterministic failure-injection benchmark designed to evaluate recovery competence, honesty, and efficiency under the seven ToolScan error classes. It combines controlled simulators, taxonomic labeling, and standardized metrics (TSR, RR, CSR, ES) to enable apples-to-apples comparison across models and methods. Unlike ToolReflectEval, which targets single-call reflective corrections, PaladinEval emphasizes trajectory-level, multi-turn recovery in noisy execution settings. All reported scores are normalized so that higher is better across tables and plots.

J.1 Benchmark Design

PaladinEval instruments ToolBench-style tasks with a simulator that injects a single, labeled execution failure at a specified turn, then drives the episode to completion while logging recovery attempts, retries, tool switches, and termination decisions. Each episode includes the task specification, tool schema, truncated trace at failure, and an injected error drawn from the ToolScan taxonomy. Evaluation is conducted with fixed seeds and deterministic tool outputs to ensure reproducibility across backbones and runs.

J.2 Tasks and Coverage

The suite spans diverse tool-use tasks representative of training domains and failure surfaces. Episodes are constructed to cover all seven ToolScan error categories under uniform sampling rules, preventing skew toward any particular class. Results reported in the main paper compare PaladinEval against ToolReflectEval across multiple backbones, showing consistent gains in Recovery Rate, Task Success Rate, and Catastrophic Success Rate. These improvements come with expected efficiency trade-offs from retry-heavy strategies, but confirm sufficient breadth and balance across failure classes for comparative evaluation.

J.3 Sampling Procedure

For each task, the first tool failure is injected deterministically by specifying the error class, manifestation (e.g., 5xx timeout, malformed JSON), and turn index. The remainder of the episode is executed with fixed tool responses to recovery actions, minimizing variance. Sampling ensures per-class coverage across *Tool Hallucination*, *Argument Hallucination*, *Invalid Invocation*, *Partial Execution*, *Output Hallucination*, *Invalid Intermediate Reasoning*, and *Re-entrant Failures*, enabling both per-class and macro-averaged reporting of RR, TSR, CSR, and ES.

J.4 Adapting ToolReflectEval

For comparability, ToolReflectEval was re-run under the same deterministic simulator and normalized metrics. While ToolReflectEval emphasizes critique-based improvements to single tool calls, PaladinEval stresses multi-turn recovery after explicit failures, capturing behaviors such as diagnosis, replanning, retries, tool switches, and graceful termination. Together, the two benchmarks provide complementary perspectives on robustness.

J.5 Filtering and Deduplication

Benchmark construction applies truncation at the first failure and removes trajectories with ambiguous or duplicate error signatures to avoid double-counting or conflating error classes. Episodes with inconsistent tool schemas or non-reproducible outputs are excluded to preserve determinism. Clean “happy-path” episodes are retained for competence checks but not scored as recoveries, ensuring evaluation focuses squarely on execution robustness.

K Additional Metrics and Ablations

K.1 Metric Variants and Formulas

We evaluate PALADIN with four metrics:

$$\begin{aligned}
\text{Task Success Rate (TSR)} &= \tfrac{\text{\# successful tasks}}{\text{\# total tasks}},\\
\text{Recovery Rate (RR)} &= \tfrac{\text{\# failures recovered}}{\text{\# failures encountered}},\\
\text{Catastrophe Success Rate (CSR)} &= 1 - \tfrac{\text{\# hallucinated successes}}{\text{\# total failures}},\\
\text{Efficiency Score (ES)} &= \tfrac{1}{\text{average \# steps to complete task}}.
\end{aligned}$$

RR, CSR, and ES are novel contributions that capture execution-level robustness beyond traditional task success. For diagnostic checks, we also experimented with call-level efficiency and normalized efficiency variants; these preserved model rankings and are omitted from main results for clarity.

K.2 Ablation: Removing Inference-Time Retrieval

PALADIN uses taxonomic retrieval over a curated bank of 55+ recovery exemplars aligned to ToolScan to map observed failures to prototypical recovery actions. Removing retrieval disables exemplar matching and forces purely end-to-end behavior. Across backbones, this sharply reduces robustness:

- **Gemma-12B:** RR 89.7% → 61.4%, TSR 87.4% → 57.3%, CSR 82.6% → 65.1%.
- **Qwen-14B:** RR 94.7% → 73.3%, TSR 79.5% → 66.3%, CSR 94.6% → 68.9%.
- **LLaMA-8B:** RR 79.8% → 48.6%, TSR 78.7% → 42.7%, CSR 80.7% → 57.4%.

- **AM-Thinking V1:** RR 96.1% → 81.2%, TSR 81.2% → 70.9%, CSR 88.7% → 73.3%.

Drops of 20–30 points highlight that learned recovery patterns are substantially amplified by exemplar-guided retrieval at inference.

K.3 Ablation: Training Data Composition

PALADIN is trained on recovery-rich traces (80%) plus clean "happy-path" traces (20%) to preserve tool competence and avoid overfitting to failure-only dynamics. Training solely on injected failures (without Teacher-authored `Recovery:` continuations) degrades multi-turn behavior: agents overfit to "retry-once" heuristics, miss plan-shift and tool-switch transitions, and exhibit decreased CSR due to missing supervised end states. Quantitatively, RR and TSR fall relative to full recovery-annotated training, with larger ES penalties from inefficient repeated retries. This confirms that explicit recovery annotations are essential for learning stable, compositional recovery behaviors.

K.4 Robustness: Zero-Shot Transfer to Unseen Tools

To evaluate generalization, we tested on held-out APIs under the same simulator and error taxonomy. Zero-shot transfer preserves a large fraction of recovery performance, with smaller absolute drops in CSR than RR/TSR. This suggests PALADIN preserves honesty under uncertainty even when recovery is incomplete. Failure analyses show strong transfer of schema-mismatch handling and malformed-output repair, while tool-specific authentication and pagination account for most residual errors. Retrieval mitigates these by guiding toward nearest exemplars, even when the exact tools are unseen.

K.5 Takeaways

- Inference-time retrieval is a primary driver of robustness, complementing training-time exposure to diverse failures.
- Recovery-annotated trajectories are critical; failure-only training under-specifies chained recovery, degrading end-to-end completion and safety.
- Zero-shot transfer demonstrates that execution-level recovery behaviors generalize, with retrieval providing a safety net for novel APIs.

L Expanded Figures

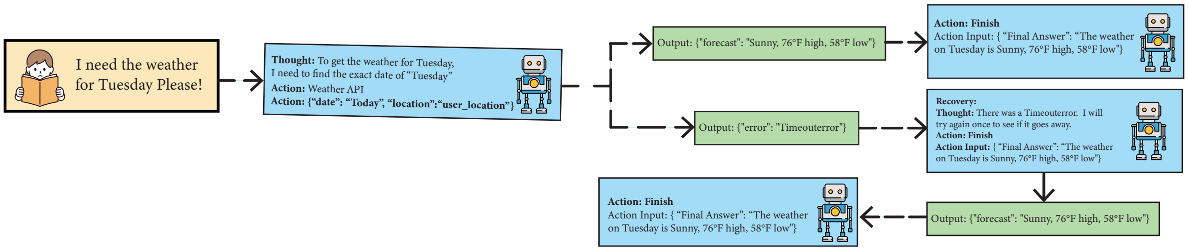
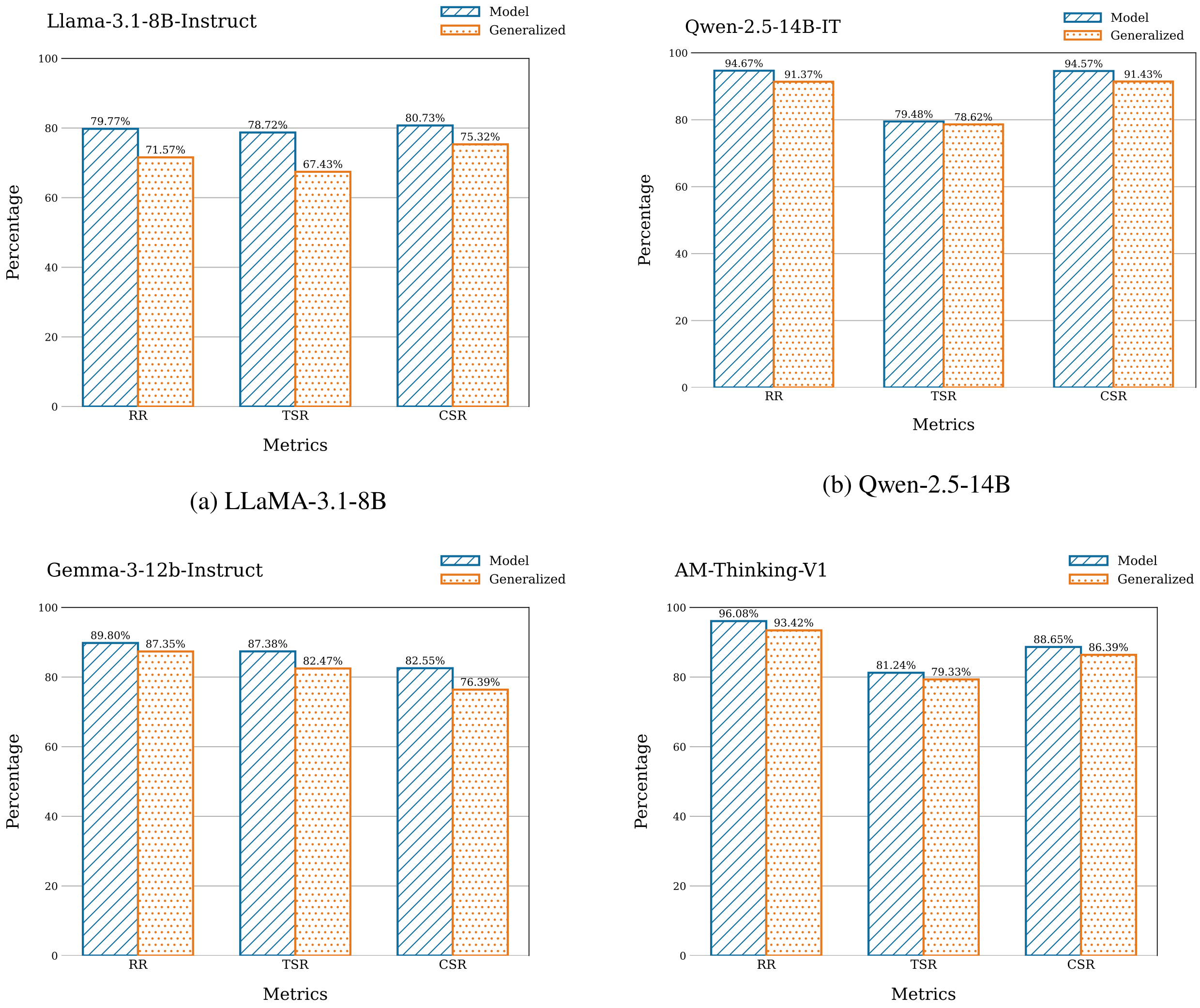

I need the weather for Tuesday Please!
Thought: To get the weather for Tuesday, I need to find the exact date of "Tuesday"
Action: Weather API
Action: {"date": "Today", "location":"user_location"}
Output: {"forecast": "Sunny, 76°F high, 58°F low"}
Action: Finish
Action Input: { "Final Answer": "The weather on Tuesday is Sunny, 76°F high, 58°F low"}
Output: {"error": "Timeouterror"}
Recovery:
Thought: There was a Timeouterror. I will try again once to see if it goes away.
Action: Finish
Action Input: { "Final Answer": "The weather on Tuesday is Sunny, 76°F high, 58°F low"}
Output: {"forecast": "Sunny, 76°F high, 58°F low"}
Action: Finish
Action Input: { "Final Answer": "The weather on Tuesday is Sunny, 76°F high, 58°F low"}

Figure 1: PALADIN's Thought Process



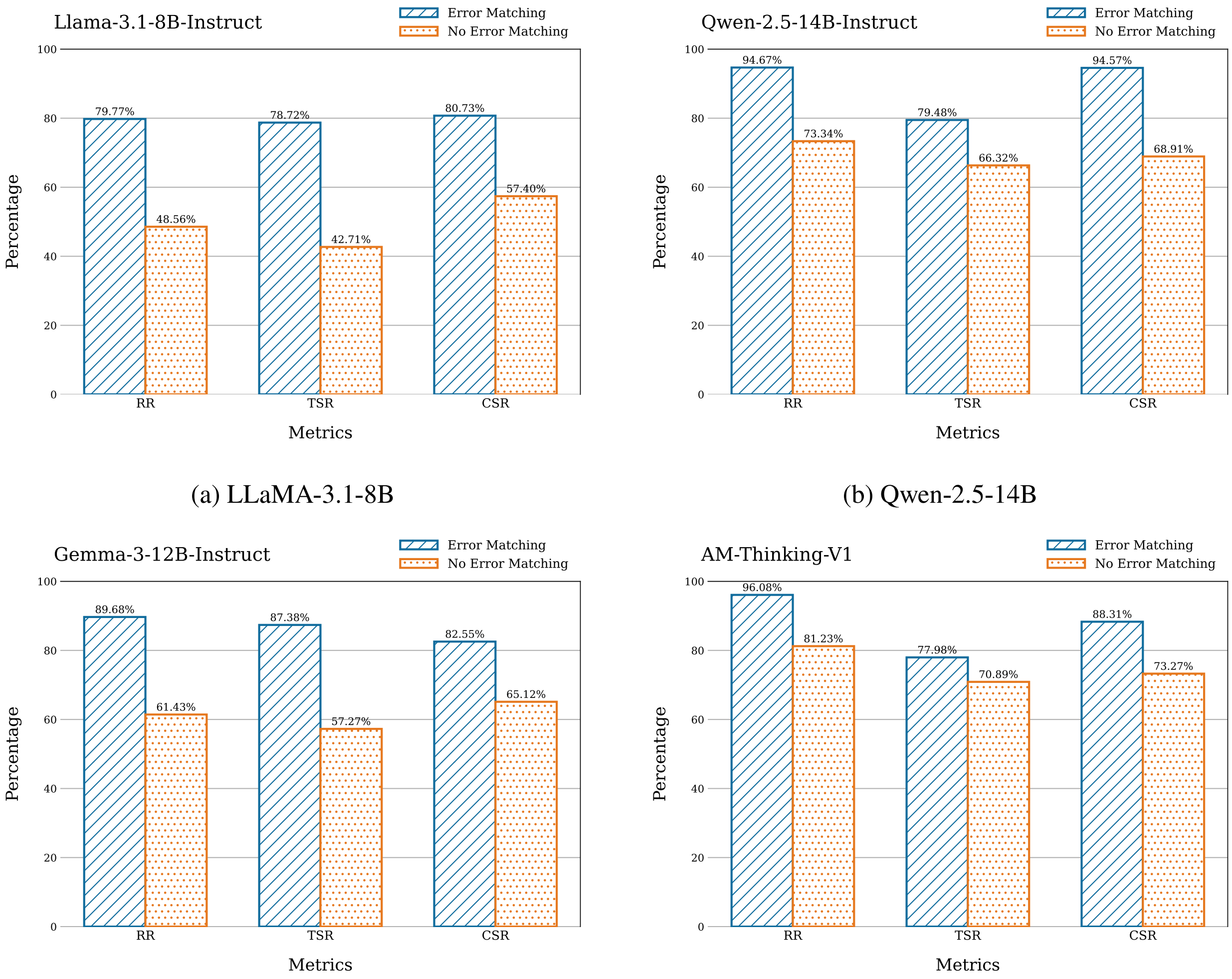
(a) LLaMA-3.1-8B

(b) Qwen-2.5-14B

(c) Gemma-3-12B

(d) AM-Thinking-V1

Figure 2: PALADIN's robustness when facing unseen error types across different model backbones.



(a) LLaMA-3.1-8B

(b) Qwen-2.5-14B

(c) Gemma-3-12B

(d) AM-Thinking-V1

Figure 3: PALADIN's abalation when facing unseen error types across different model backbones.