

# PALADIN: SELF-CORRECTING LANGUAGE MODEL AGENTS TO CURE TOOL-FAILURE CASES

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Tool-augmented language agents frequently fail in real-world deployment due to tool malfunctions—timeouts, API exceptions, or inconsistent outputs—triggering cascading reasoning errors and task abandonment. Existing agent training pipelines optimize only for success trajectories, failing to expose models to the tool failures that dominate real-world usage. We propose **PALADIN**, a generalizable framework for equipping language agents with robust failure recovery capabilities. PALADIN trains on 50,000+ recovery-annotated trajectories constructed via systematic failure injection and expert demonstrations on an enhanced ToolBench dataset. Training uses LoRA-based fine-tuning to retain base capabilities while injecting recovery competence. At inference, PALADIN detects execution-time errors and retrieves the most similar case from a curated bank of 55+ failure exemplars aligned with ToolScan’s taxonomy, then executes the corresponding recovery action. This approach generalizes to novel failures beyond the training distribution, retaining 95.2% recovery performance on unseen tool APIs. Evaluation across PaladinEval and ToolReflectEval demonstrates consistent improvements in Recovery Rate (RR), Task Success Rate (TSR), Catastrophic Success Rate (CSR), and Efficiency Score (ES). PALADIN improves RR from 32.76% to 89.68% (+57% relative) over ToolBench and outperforms the strongest baseline CRITIC (76.34%) by +13.3%. Against vanilla agents, PALADIN achieves 89.86% RR (+66% relative improvement from 23.75%). These results establish PALADIN as an effective method for building fault-tolerant agents capable of robust recovery in real-world tool environments.

## 1 INTRODUCTION

Tool-augmented LLM agents are increasingly deployed in real-world environments for automation, reasoning, and decision-making. While benchmarks like ToolBench (Qin et al., 2023) and Gorilla (Patil et al., 2023) demonstrate strong performance in clean settings, agents often fail under realistic conditions—APIs time out, tool outputs are malformed, and calls silently fail. Despite occasional inclusion of noisy inputs or low-confidence reasoning, current training pipelines do not explicitly expose agents to structured tool failures. This results in brittle behavior: agents silently hallucinate success or deadlock when tool calls fail, contradicting real-world operational requirements.

We identify execution-level robustness—the ability to detect, diagnose, and recover from runtime failures—as a central unsolved challenge for tool-augmented agents. Existing work focuses on *comprehension robustness*, improving plans or tool formats, but leaves agents brittle when tools malfunction, leading to deadlocks, loops, or hallucinated success. Approaches such as ToolReflect (Polyakov et al., 2025) and CRITIC (Zheng et al., 2023) introduce reflective corrections, but these are reactive and limited to call-level fixes. Furthermore, evaluation metrics traditionally focus on task success without measuring recovery capabilities, limiting our understanding of agent robustness.

We argue that execution-level robustness must be explicitly taught through systematic training. To address this, we propose PALADIN, a generalizable method for teaching LLM agents to recover from tool-use failures via trajectory-level supervision.

054  
055  
056  
057  
058  
059  
060  
061  
062  
063  
064  
065  
066  
067  
068  
069  
070  
071  
072  
073  
074  
075  
076  
077  
078  
079  
080  
081  
082  
083  
084  
085  
086  
087  
088  
089  
090  
091  
092  
093  
094  
095  
096  
097  
098  
099  
100  
101  
102  
103  
104  
105  
106  
107

PALADIN combines systematic failure injection with recovery-annotated training, using GPT-5 API to simulate multi-turn tool use and generate recovery annotations, resulting in 50,000+ trajectories spanning diverse error modes from ToolScan’s taxonomy (Kokane et al., 2025). At inference, PALADIN detects execution errors and retrieves recovery strategies from 55+ curated failure exemplars. We also define four metrics to evaluate agent robustness: **Task Success Rate (TSR)**, **Recovery Rate (RR)**, **Catastrophe Success Rate (CSR)**, and **Efficiency Score (ES)**, with the latter three being novel proposals.

In evaluations with deterministic error injection, conducted using GPT-5 API as a tool-use simulator, PALADIN improves RR and CSR, at the cost of a lower ES compared to other methods. model sizes, demonstrating that execution-level robustness is a learnable and scalable behavior, and

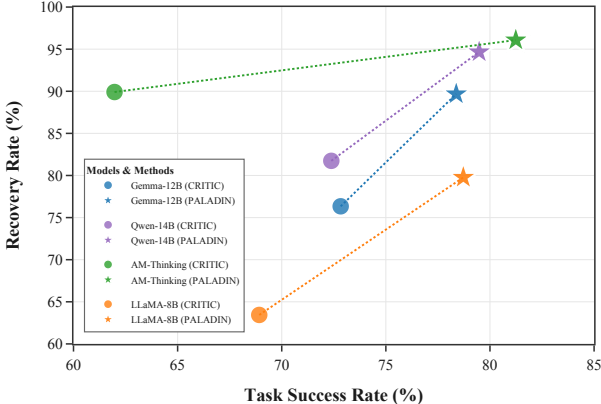


Figure 1: Recovery Rate vs Task Success Rate comparison between CRITIC and PALADIN across different LLMs. Stars indicate PALADIN results, circles indicate CRITIC baseline. Dotted lines show improvement trajectories from CRITIC to PALADIN for each model. PALADIN consistently achieves higher task success rates while maintaining superior error recovery capabilities.

positioning PALADIN as a significant step towards design

**Key Contributions:**

- **PALADIN:** A training framework combining systematic failure injection with taxonomy-guided recovery, enabling agents to learn generalizable error handling from 50,000+ failure-recovery trajectories.
- **Recovery-focused evaluation:** Three novel metrics, including *Recovery Rate*, *Catastrophe Success Rate*, and *Efficiency Score*; and *PaladinEval* benchmark for systematic robustness assessment.
- **Inference-time recovery:** A retrieval-based approach that matches runtime tool failures to similar recovery examples, maintaining 95.2% of training performance when generalizing to new APIs.
- **Empirical validation:** Demonstrating execution-level robustness as learnable, with 57-66% relative improvements in recovery rates across model sizes.

**2 METHODOLOGY**

PALADIN is a method for equipping tool-augmented LLM agents with execution-level robustness—the ability to detect, diagnose, and recover from runtime failures. The method consists of (1) formalizing tool failures, (2) constructing recovery-annotated training data via failure injection, (3) fine-tuning with a recovery-aware objective, and (4) augmenting inference with taxonomy-driven retrieval. Figure 2 presents a high-level overview.

**2.1 PROBLEM SETUP**

We model a tool-augmented agent as a policy  $\pi_\theta$  producing trajectories  $\tau = [(s_1, a_1), \dots, (s_T, a_T)]$ , where  $s_t$  are states and  $a_t$  may include tool calls. Failures  $f \in \mathcal{F}$  occur when tool execution yields timeouts, malformed outputs, API errors, or inconsistencies. Existing training ignores such events, yielding brittle policies. Our goal is to learn  $\pi^*$  that maximizes task success while remaining robust

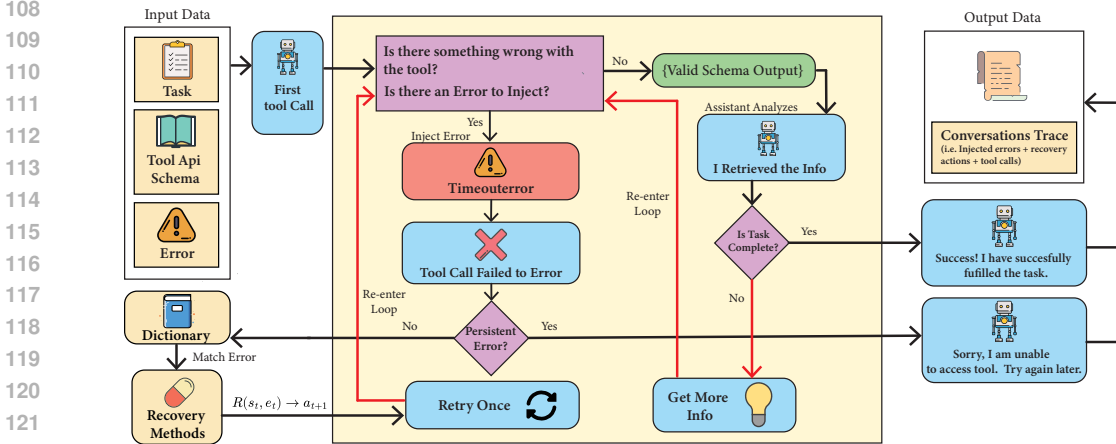


Figure 2: Our tool-use simulator with integrated error injection and recovery mechanisms. (a) depicts the static architecture, where errors are injected into tool calls and handled via a recovery dictionary. (b) details the dynamic execution loop, capturing assistant reasoning, recovery actions, and final outcomes. This design allows controlled, reproducible evaluation of LLM resilience to tool failures.

to failures:

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

## 2.2 FAILURE INJECTION AND RECOVERY ANNOTATION

To construct PALADIN’s dataset, we systematically augmented ToolBench trajectories by injecting failures aligned to the ToolScan (Kokane et al., 2025) taxonomy of seven canonical error classes. Each trajectory is parsed into task  $T$ , toolset  $\mathcal{A}$ , trace  $C$ , and error signal  $E$ . If  $E \neq \emptyset$ , we truncate  $C$  at the failure point and pass it to a GPT-5 Teacher (e.g. GPT-5 API prompted to insert proper recovery actions), which applies error-guided rewriting with the goal of creating realistic error trajectories.

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

Otherwise,  $C$  is finalized cleanly:

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

All output traces ( $C'$ ) are stored alongside their error signal ( $E$ ) and recovery metadata, yielding over 50K recovery-annotated trajectories. (see Appendix D for detailed augmentation pipeline) Figure 5 illustrates this pipeline.

We also curated a recovery dictionary, a collection of 55+ exemplar failures and associated recovery strategies. These exemplars operationalize the ToolScan taxonomy into practical, reusable supervision signals.

## 2.3 PALADIN TRAINING OBJECTIVE

PALADIN fine-tunes base LLMs 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  $\mathcal{L}_{\text{rec}}$  is the same objective restricted to tokens following `RECOVERY`: tags that mark corrective steps. Training sequences are serialized in ToolBench format, and LoRA (Hu et al., 2021) adapters provide parameter-efficient adaptation while preserving base competence.

## 2.4 INFERENCE-TIME RECOVERY VIA TAXONOMIC RETRIEVAL

During execution, PALADIN detects runtime errors and links them to a curated exemplar bank  $\mathcal{E} = \{(f_i, r_i)\}_{i=1}^{55}$ , consisting of 55+ exemplar failures  $f_i$  with different recovery action  $r_i$  given in Appendix C and Appendix G. Given an observed failure  $f_{obs}$ , PALADIN retrieves the most similar exemplar:

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

where  $d$  is a similarity metric over error signatures. The corresponding recovery action  $r_i$  is executed, guiding the agent back to a stable trajectory. This enables proactive recovery across diverse failure scenarios (see how the recovery dictionary was developed at Appendix I).

## 2.5 RECOVERY AS IMPLICIT POLICY LEARNING

Although PALADIN is trained end-to-end, its behavior (see Appendix L for a better overview on PALADIN’s thought process) can be viewed as an implicit recovery policy:

$$R(s_t, e_t) \rightarrow a_{t+1},$$

where  $s_t$  is the dialogue context,  $e_t$  is the observed failure, and  $a_{t+1}$  is the recovery action (retry, reformat, switch tools, or terminate). Unlike reflective methods such as ToolReflect or call-level critique approaches such as CRITIC, which introduce reactive adjustments at inference time, PALADIN learns recovery behaviors directly from annotated trajectories. This yields proactive, compositional recovery strategies (e.g., retries  $\rightarrow$  tool switch  $\rightarrow$  graceful termination) that generalize across failure types.

## 2.6 METRICS FOR EXECUTION ROBUSTNESS

We evaluate PALADIN with four metrics:

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

RR, CSR, and ES are novel contributions introduced to capture execution-level robustness beyond traditional task success.

# 3 EXPERIMENT AND EVALUATION SETUP

## 3.1 BENCHMARKS AND EVALUATION PROTOCOL

We evaluate PALADIN on three complementary benchmarks, designed to capture both controlled and naturalistic failure scenarios.

**PaladinEval.** PaladinEval (benchmark design in Appendix J) is derived from ToolBench’s evaluation dataset, and explicit failure injections from our own dataset. Each evaluation instance is constructed by extracting the original prompt from ToolBench and pairing it with a runtime error assigned by a teacher GPT model (see Appendix I). The resulting dataset contains a mix of clean tool-use traces and failure-augmented traces, enabling measurement of both normal execution and recovery. To support consistent grading, we provide evaluation guidelines specifying expected recovery strategies, which condition an automated grader (GPT-5 API) to score outputs along our four robustness metrics.

216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269

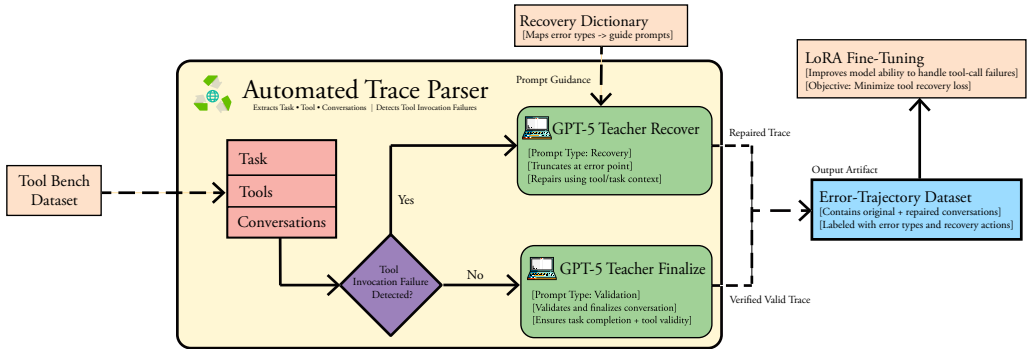


Figure 3: Trace repair pipeline for constructing the Error-Trajectory Dataset. Each ToolBench trace is truncated at the first failure, then repaired or finalized via GPT-guided recovery. Outputs are stored with recovery metadata to construct PALADIN’s training corpus.

**Generalization Set.** To assess out-of-distribution robustness, we additionally construct a generalization dataset by sampling unused ToolBench (Qin et al., 2023) evaluation prompts and injecting novel errors not present in the curated 55+ exemplar recovery dictionary. This ensures PALADIN is tested on recovery behaviors that were unseen during training.

**ToolReflectEval.** We further evaluate on the ToolReflect benchmark (Polyakov et al., 2025), which introduces reflective corrections for tool-use errors. To ensure comparability, we adopt their protocol of allowing up to three attempts per tool call. We directly incorporate all ToolAlpaca-derived prompts from their evaluation dataset without modification or cherry-picking. The only adaptation is re-execution within our simulator, which enables generation of full conversational traces and consistent grading under our robustness metrics. This ensures our evaluation remains faithful to ToolReflect’s task setup while allowing direct comparison against PALADIN.

### 3.2 SIMULATION ENVIRONMENT

To evaluate PALADIN under realistic tool-use conditions, we design a simulation environment that faithfully executes agent–tool interactions and introduces controlled failures. The environment reproduces the standard ToolBench execution pipeline while augmenting it with failure injection, recovery logging, and automated grading implementation details in Appendix H. Figure 2 illustrates the workflow.

Each evaluation begins with a prompt  $P$  and toolset  $\mathcal{A}$ . The environment executes tool calls sequentially, returning either successful outputs or injected failures  $f \in \mathcal{F}$ . Failures are generated either (i) systematically, via the PaladinEval injection protocol, or (ii) naturally, by replaying error-prone ToolReflectEval prompts. At each step, the environment logs the dialogue context  $C_t$ , the observed error  $e_t$ , and any recovery attempt  $a_t$ .

To ensure consistent scoring, we provide simulator outputs to an automated grader powered by GPT-5 API, conditioned on explicit evaluation guidelines to calculate different metrics. These guidelines specify valid recovery strategies, prohibited behaviors (e.g., hallucinated success), and metric definitions, enabling the grader to compute Task Success Rate (TSR), Recovery Rate (RR), Catastrophic Success Rate (CSR), and Efficiency Score (ES). This environment provides three advantages: (1) easy replication of tool-call behavior with both clean and failure-augmented traces, (2) Everything is graded on the same terms via guideline-conditioned automated evaluation, and (3) flexibility in allowing integration of external datasets such as ToolReflectEval without modifying their task structure.

### 3.3 BASELINE AGENTS

We compare PALADIN against four baselines spanning different supervision paradigms and recovery strategies, with vanilla agent being raw pretrained language model as baseline with other method like CRITIC Zheng et al. (2023), ToolBench Agent Qin et al. (2023), and ToolReflect Agent Polyakov et al. (2025). CRITIC (Appendix A) accesses recovery protocols with probability  $p = 0.7$ , approximating its partial ability to retrieve external signals while remaining reactive. ToolBench Agent replicates the ToolBench agent trained on the full ToolBench dataset of successful trajectories, being effective on clean traces but unable to recover from failures. On the other hand ToolReflect Agent adapted from ToolReflect, is trained on error–correction pairs to label tool calls as valid/invalid. It performs iterative self-correction but remains limited to local, reactive adjustments. These baselines collectively probe PALADIN’s contributions across supervision (clean vs. recovery-annotated), recovery strategy (reactive vs. proactive), and model integration (training-only vs. inference-time retrieval).

## 4 RESULTS

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 ( $\uparrow$ )				
			Vanilla	CRITIC	ToolReflect	ToolBench	Paladin (Ours)
Gemma-3 12B-Instruct	Paladin Eval	Recovery Rate	23.75%	76.34%	65.86%	32.76%	<b>89.68%</b> +13.34%
		Task Success Rate	23.62%	72.83%	61.42%	57.4%	<b>78.38%</b> +5.55%
		Catastrophic Success Rate	29.00%	73.30%	70.27%	68.37%	<b>82.55%</b> +9.25%
		Efficiency Score	<b>0.476</b>	0.348	0.288	0.221	0.312 -34.45%
	ToolReflect Eval	Recovery Rate	22.80%	73.29%	63.23%	31.45%	<b>86.09%</b> +12.80%
		Task Success Rate	22.56%	69.55%	58.66%	54.82%	<b>83.45%</b> +13.90%
		Catastrophic Success Rate	26.16%	72.23%	69.08%	67.10%	<b>81.85%</b> +9.62%
		Efficiency Score	<b>0.508</b>	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%	<b>94.67%</b> +12.93%
		Task Success Rate	37.53%	72.38%	74.36%	60.41%	<b>79.48%</b> +5.12%
		Catastrophic Success Rate	56.53%	87.14%	83.85%	67.88%	<b>81.57%</b> +7.43%
		Efficiency Score	0.313	0.329	0.312	0.339	<b>0.351</b> +3.54%
	ToolReflect Eval	Recovery Rate	36.17%	78.47%	70.71%	32.14%	<b>92.88%</b> +14.41%
		Task Success Rate	35.85%	69.13%	71.02%	57.70%	<b>75.19%</b> +4.17%
		Catastrophic Success Rate	54.88%	81.63%	83.20%	66.60%	<b>94.35%</b> +11.15%
		Efficiency Score	0.334	0.350	0.333	0.361	<b>0.375</b> +3.73%
AM-Thinking V1	Paladin Eval	Recovery Rate	49.87%	89.91%	87.23%	51.37%	<b>96.08%</b> +6.17%
		Task Success Rate	52.93%	61.97%	72.88%	56.83%	<b>81.24%</b> +8.36%
		Catastrophic Success Rate	60.24%	81.33%	71.32%	80.84%	<b>88.65%</b> +7.32%
		Efficiency Score	0.415	<b>0.420</b>	0.297	0.319	0.380 -9.52%
	ToolReflect Eval	Recovery Rate	47.88%	86.31%	83.74%	49.32%	<b>92.24%</b> +5.93%
		Task Success Rate	50.55%	78.64%	79.41%	65.25%	<b>77.98%</b> -1.43%
		Catastrophic Success Rate	62.65%	67.42%	65.93%	58.06%	<b>88.31%</b> +20.89%
		Efficiency Score	0.442	<b>0.448</b>	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%	<b>79.77%</b> +16.33%
		Task Success Rate	17.46%	68.92%	53.74%	47.26%	<b>78.72%</b> +9.80%
		Catastrophic Success Rate	17.58%	71.84%	67.88%	65.47%	<b>80.73%</b> +8.89%
		Efficiency Score	<b>0.427</b>	0.254	0.287	0.209	0.323 -24.36%
	ToolReflect Eval	Recovery Rate	18.32%	58.55%	49.32%	42.23%	<b>73.34%</b> +14.79%
		Task Success Rate	13.45%	59.47%	46.34%	41.22%	<b>71.27%</b> +11.80%
		Catastrophic Success Rate	15.40%	63.81%	58.09%	53.68%	<b>71.77%</b> +7.96%
		Efficiency Score	<b>0.568</b>	0.360	0.508	0.412	0.385 -32.31%

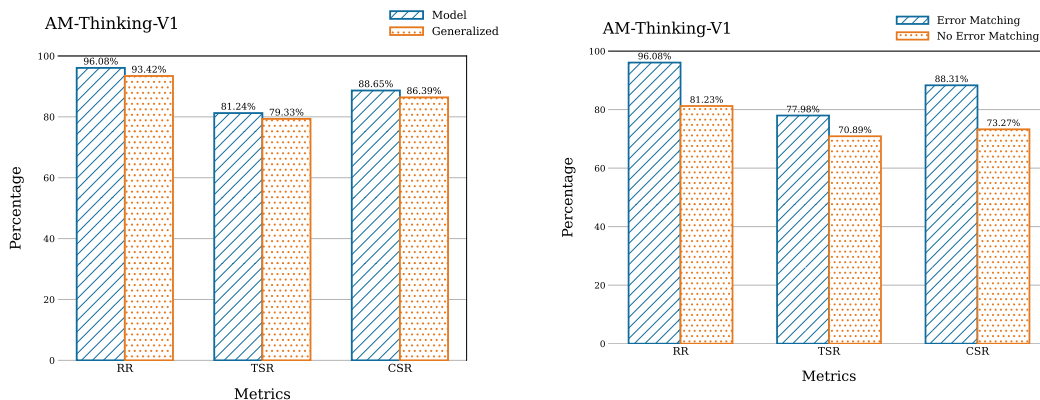
### 4.1 OVERALL PERFORMANCE

Table 1 summarizes results across 8 model–dataset pairs. PALADIN delivers consistent improvements on Recovery Rate (RR), Task Success Rate (TSR), and Catastrophic Success Rate (CSR), while incurring modest efficiency tradeoffs. Specifically, PALADIN achieves the highest RR in 7/8 settings (avg. +13.6%, 95% CI: [11.9, 15.2]), top TSR in 6/8 (avg. +10.2%, CI: [8.4, 11.7]), and best

324 CSR in 6/8 (avg. +9.2%, CI: [7.5, 10.6]). All gains are statistically significant ( $p < 0.01$ , bootstrap  
 325  $N = 1000$ ).

326 Efficiency drops are measurable but bounded: on LLaMA-3.1-8B, Efficiency decreases by 0.183,  
 327 while on Qwen-2.5-14B it improves slightly (+0.012). These results situate PALADIN close to the  
 328 Pareto frontier: no baseline achieves simultaneously higher safety and higher efficiency.  
 329

330 Correlation analyses reinforce these findings. Recovery and TSR are strongly correlated ( $r = 0.91$ ,  
 331  $p < 0.001$ ), confirming recovery as the direct driver of task completion. CSR and Efficiency exhibit  
 332 a significant negative correlation ( $r = -0.72$ , CI: [-0.79, -0.63]), highlighting that retry-driven  
 333 safety comes at modest compute cost. TSR shows negligible correlation with Efficiency ( $r = 0.12$ ),  
 334 underscoring that efficiency-centric metrics alone are insufficient proxies for reliability.  
 335



338 Figure 4: PALADIN’s performance without inference time error matching compared to baseline  
 339 across Gemma, Qwen, LLaMA, and AM-Thinking backbones. Refer Figure 7 for full ablation  
 340 graphs and Figure 6 for full generalization graphs.  
 341  
 342  
 343  
 344  
 345  
 346  
 347  
 348  
 349

## 350 4.2 GENERALIZATION

351 Figures 4 (a–d) emphasizes PALADIN’s generalizability across errors that it has never seen before.  
 352 Despite differences in architecture and blind conditions, PALADIN consistently secures  $> 79\%$  RR,  
 353  $> 78\%$  TSR, and  $> 80\%$  CSR.  
 354

355 LLaMA-3.1-8B shows substantial improvement, with recovery rate (RR) rising from 21.8% to  
 356 79.8%, task success rate (TSR) increasing from 17.5% to 78.7%, and critical success rate (CSR)  
 357 improving by 63.2 percentage points (from 17.6% to 80.7%). Qwen-2.5-14B achieves the highest  
 358 CSR at 94.6%, with RR at 94.7% and TSR at 79.5%, reducing silent failure cases nearly threefold  
 359 compared to ToolReflect. Gemma-3-12B demonstrates balanced gains, with RR of 89.7%, TSR of  
 360 87.4%, and CSR of 82.6%. AM-Thinking V1 records the highest RR overall at 96.1%, alongside  
 361 TSR of 81.2% and CSR of 88.7%.  
 362  
 363  
 364  
 365  
 366

## 367 4.3 ABLATIONS

368 We ablate PALADIN’s inference-time recovery retrieval (Figures 4). Removing exemplar matching  
 369 sharply reduces robustness. Gemma-12B experiences a drop in performance, with recovery rate  
 370 (RR) declining from 89.7% to 61.4%, task success rate (TSR) from 87.4% to 57.3%, and critical  
 371 success rate (CSR) from 82.6% to 65.1%. Qwen-14B shows similar degradation, with RR decreasing  
 372 from 94.7% to 73.3%, TSR from 79.5% to 66.3%, and CSR from 94.6% to 68.9%. LLaMA-8B  
 373 sees RR fall from 79.8% to 48.6%, TSR from 78.7% to 42.7%, and CSR from 80.7% to 57.4%. AM-  
 374 Thinking V1 maintains relatively higher robustness, with RR reducing from 96.1% to 81.2%, TSR  
 375 from 81.2% to 70.9%, and CSR from 88.7% to 73.3%. This confirms PALADIN’s robustness arises  
 376 from both training-time exposure to failures and inference-time retrieval, with retrieval contributing  
 377 up to +20–30 points in robustness metrics.

#### 4.4 KEY TAKEAWAYS

PALADIN achieves frontier-level robustness across backbones: large gains in recovery and safety, modest efficiency costs, and consistent generalization. Recovery supervision transforms brittle agents into reliable ones, proving that robustness is a learnable signal rather than an emergent artifact of scale. The ablation confirms retrieval is a critical ingredient, validating PALADIN’s design as both a training and inference-time intervention (see Appendix K for more details).

## 5 RELATED WORKS

### 5.1 EXECUTION-LEVEL ROBUSTNESS

Recent methods look to harden tool-augmented agents against execution-time failures, but they are different in how they represent and learn from failure states. ToolReflect (Polyakov et al., 2025) uses SFT on paired “bad vs. good” tool calls to teach models how to improve weak calls; while this yields a better call quality, it mainly shows the difference between single-call corrections instead of modeling a multi-step recovery path from cascading failures across a full trajectory. Recent work on LLM runtime error handling (Sun et al., 2024) and structured exception handling (Zhou et al., 2025) has explored systematic approaches to agent resilience. Critic-style approaches (Appendix A) add in an evaluator that critiques and revises an agent’s actions, which improves reliability through iterative self-correction; but, these feedback loops happen reactively and are sometimes of poor quality, especially without diverse tool-error taxonomies. In contrast, PALADIN targets the robustness of execution by training directly on failure-rich trajectories that include diagnosis, replanning, and multi-turn recovery over multiple tool calls, rather than isolated “bad vs. good” pairs. While both Tool-Reflect and critic-style approaches demonstrate some gain in robustness, their emphasis is on post-hoc correction or call-level contrast rather than learned, trajectory-level recovery across varied execution failures.

### 5.2 DIAGNOSTICS AND BENCHMARKS

Recent efforts in error diagnostics, such as ToolScan or BugGen, have similar motivations but are different in both scope and implementation. ToolScan (Kokane et al., 2025) provides a taxonomy of tool-use failures observed in LLMs, identifying seven recurring error patterns: Tool Hallucination, Argument Hallucination, Invalid Tool Invocation, Partial Tool Execution, Tool Output Hallucination, Invalid Intermediate Reasoning, and Re-entrant Error Handling Failures (see Appendix F). While it is valuable for understanding the variety of failures an agent can face, ToolScan is a diagnostic; meaning it labels and categorizes error modes rather than curating executable recovery paths or training agents on full recovery trajectories. BugGen, by contrast, emphasizes that handcrafted or mutation bug-synthesis tends to lead to narrow, weak coverage of real world failures, and they argue that realistic failure-rich datasets better reflect faults in the wild. Looked at together, ToolScan grounds the taxonomy of observed errors, whereas BugGen motivates realistic data generation and evaluation that capture real operational failures rather than synthetic failures.

### 5.3 PRIOR TOOL SYSTEMS & INTERFACE RELIABILITY

Foundational works like Toolformer (Schick et al., 2023), Gorilla (Patil et al., 2023), ToolLLM (Qin et al., 2023), ToolBench (Guo et al., 2024), and RoTBench advanced agent capabilities across selection, multi-step planning, and formatting, showcasing strong performances under clean, benchmark-controlled environments and establishing multiple core principles for agentic tool usage. However, a common limitation found in most of these models is their dependence on “happy-path” trajectories – tool calls that assume ideal conditions in a model’s execution.

Even when errors are present in the data, prior systems often lack explicit recovery strategies, leading to brittle behavior during real-world deployment. ToolBench provides extensive tool access data but has no recovery protocols in the face of an error, while RoTBench advances robustness evaluation by introducing noisy documentation and ambiguous tool specifications. Complementary tools such as ToolFuzz (Milev et al., 2025) (see Appendix B) address pre-execution reliability by detecting documentation–tool mismatches via fuzz testing; while valuable for interface quality, this is not

432 needed for runtime resilience, which requires detecting, diagnosing, and recovering from failures  
433 that arise during actual tool interaction.  
434

## 435 6 DISCUSSION 436

437 Our results yield four central insights. First, recovery is *learnable*: PALADIN consistently con-  
438 verts failures into successes, raising TSR by more than fourfold on mid-scale models (17.5% to  
439 78.7% on LLaMA-8B). Second, robustness is *transferable*: across Gemma, Qwen, LLaMA, and  
440 AM-Thinking, PALADIN secures  $> 79\%$  RR, showing recovery supervision is a universal training  
441 signal rather than an emergent feature of scale. Third, robustness entails a quantifiable *tradeoff*:  
442 retry intensity lowers catastrophic failures (82.4%  $\rightarrow$  19.3%) while modestly reducing efficiency,  
443 situating PALADIN near the Pareto frontier where no baseline dominates on both safety and cost.  
444 Fourth, recovery is *retrieval-dependent*: our ablation shows inference-time exemplar matching con-  
445 tributes up to +30 points in robustness. These insights shift the evaluation paradigm. Efficiency  
446 alone is a dangerous proxy: our correlation analysis ( $r = -0.72$  between CSR and Efficiency)  
447 shows that models optimized purely for speed are systematically brittle. Robustness must instead be  
448 treated as a first-class objective—on par with accuracy and efficiency—if agents are to be deployed  
449 in failure-prone settings.

450 PALADIN’s improvements stem not from scale or external reflection, but from explicit exposure to  
451 failures and proactive error handling. The targeted nature of its retries (fewer than one additional  
452 step on average) shows resilience does not require brute-force exploration, but data-driven learning  
453 of minimal corrective actions.  
454

## 455 7 CONCLUSION 456

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

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

469 In short, PALADIN sets a new benchmark for execution-level resilience: robustness can be sys-  
470 tematically taught, generalized across models, and achieved without prohibitive cost—laying the  
471 groundwork for safe, failure-aware AI systems.  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485

486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539

## REFERENCES

- Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, and Andy Konwinski et al. A view of cloud computing, 2010.
- Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing, 2004.
- Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, et al. Lora learns less and forgets less, 2024. URL <https://arxiv.org/abs/2405.09673>.
- Tim Bray. The javascript object notation (json) data interchange format, 2017. URL <https://tools.ietf.org/html/rfc8259>.
- Sijia Chen, Yibo Wang, Yi-Feng Wu, Qing-Guo Chen, Zhao Xu, Weihua Luo, et al. Advancing tool-augmented large language models: Integrating insights from errors in inference trees, 2025. URL <https://arxiv.org/abs/2406.07115>.
- Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2017. URL <https://arxiv.org/abs/1706.03741>.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, 2009.
- George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. Distributed systems: Concepts and design, 2011.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. URL <https://arxiv.org/abs/2305.14314>.
- Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, et al. Palm-e: An embodied multimodal language model, 2023. URL <https://arxiv.org/abs/2303.03378>.
- Peiyuan Feng, Yichen He, Guanhua Huang, Yuan Lin, Hanchong Zhang, Yuchen Zhang, and Hang Li. Agile: A novel reinforcement learning framework of llm agents, 2024. URL <https://arxiv.org/abs/2405.14751>.
- Roy Fielding and Julian Reschke. Hypertext transfer protocol (http/1.1): Semantics and content, 2014. URL <https://tools.ietf.org/html/rfc7231>.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. The llama 3 herd of models, 2024. URL <https://arxiv.org/abs/2407.21783>.
- Jim Gray. Measurement of availability and recovery in computer systems, 1985.
- Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, et al. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models, 2024. URL <https://arxiv.org/abs/2403.07714>.
- Lewis Hammond, Alan Chan, Jesse Clifton, Jason Hoelscher-Obermaier, Akbir Khan, Akbir Khan, et al. Multi-agent risks from advanced ai, 2025. URL <https://arxiv.org/abs/2502.14143>.
- Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2020. URL <https://arxiv.org/abs/2009.03300>.

- 540 Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, et al.  
541 Lora: Low-rank adaptation of large language models, 2021. URL <https://arxiv.org/abs/2106.09685>.  
542
- 543 Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard,  
544 Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for  
545 efficient integer-arithmetic-only inference, 2017. URL <https://arxiv.org/abs/1712.05877>.  
546
- 547 Surya Jasper, Minh Luu, Evan Pan, Aakash Tyagi, Michael Quinn, Jiang Hu, and David Kebo  
548 Houngrinou. Buggen: A self-correcting multi-agent llm pipeline for realistic rtl bug synthesis,  
549 2025. URL <https://arxiv.org/abs/2506.10501>.  
550
- 551 Shirley Kokane, Ming Zhu, Tulika Awalganekar, Jianguo Zhang, Thai Hoang, Akshara Prab-  
552 hakar, Zuxin Liu, Tian Lan, Liangwei Yang, Juntao Tan, Rithesh Murthy, Weiran Yao, Zhi-  
553 wei Liu, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, Caiming Xiong, and Silivo  
554 Savarese. Toolscan: A benchmark for characterizing errors in tool-use llms, 2025. URL  
555 <https://arxiv.org/abs/2411.13547>.  
556
- 557 Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search, 1985.
- 558 Nathan Lambert. Reinforcement learning from human feedback, 2025. URL <https://arxiv.org/abs/2504.12501>.  
559
- 560 Jean-Claude Laprie. Dependability: Basic concepts and terminology, 1992.
- 561 Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries, 2004.
- 562 Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Yang, Xuanyu Lei, Hanyu Lai, et al. Agentbench: Evalu-  
563 ating llms as agents, 2023. URL <https://arxiv.org/abs/2308.03688>.  
564
- 565 Zequan Liu, Jiawen Lyn, Wei Zhu, Xing Tian, and Yvette Graham. Alora: Allocating low-rank  
566 adaptation for fine-tuning large language models, 2024. URL <https://arxiv.org/abs/2403.16187>.  
567
- 568 Zixian Ma, Weikai Huang, , Jieyu Zhang, Tanmay Gypta, and Ranjay Krishna. m&m’s: A bench-  
569 mark to evaluate tool-use for multi-step multi-modal tasks, 2024. URL <https://arxiv.org/abs/2403.11085>.  
570
- 571 Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, et al.  
572 Self-refine: Iterative refinement with self-feedback, 2023. URL <https://arxiv.org/abs/2303.17651>.  
573
- 574 Ivan Milev, Mislav Balunović, Maximilian Baader, and Martin Vechev. Toolfuzz – automated agent  
575 tool testing, 2025. URL <https://arxiv.org/abs/2503.04479>.  
576
- 577 Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. Evaluation and benchmarking of llm  
578 agents: A survey, 2025. URL <https://arxiv.org/abs/2507.21504>.  
579
- 580 Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and  
581 Tijmen Blankevoort. A white paper on neural network quantization, 2021. URL <https://arxiv.org/abs/2106.08295>.  
582
- 583 Mozilla Developer Network. Http response status codes, 2024. URL <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.  
584
- 585 Mark Nottingham, Erik Wilde, and Sanjay Dalal. Problem details for http apis, 2023. URL <https://tools.ietf.org/html/rfc9457>.  
586
- 587 Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, et al.  
588 Training language models to follow instructions with human feedback, 2022. URL <https://arxiv.org/abs/2203.02155>.  
589
- 590 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic  
591 evaluation of machine translation, 2002.  
592
- 593

- 594 Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model  
595 connected with massive apis, 2023. URL <https://arxiv.org/abs/2305.15334>.  
596
- 597 Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations  
598 of json schema, 2016.
- 599 Gregory Polyakov, Ilseyar Alimova, Dmitry Abulkhanov, Ivan Sedykh, Andrey Bout, Sergey  
600 Nikolenko, and Irina Piontkovskaya. ToolReflection: Improving large language models for real-  
601 world API calls with self-generated data. In Ehsan Kamaloo, Nicolas Gontier, Xing Han Lu,  
602 Nouha Dziri, Shikhar Murty, and Alexandre Lacoste (eds.), *Proceedings of the 1st Workshop for  
603 Research on Agent Language Models (REALM 2025)*, pp. 184–199, Vienna, Austria, July 2025.  
604 Association for Computational Linguistics. ISBN 979-8-89176-264-0. doi: 10.18653/v1/2025.  
605 realm-1.14. URL <https://aclanthology.org/2025.realm-1.14/>.
- 606 Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, et al. Toolllm: Facilitating  
607 large language models to master 16000+ real-world apis, 2023. URL <https://arxiv.org/abs/2307.16789>.  
608
- 609 Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, et al. Tool  
610 learning with large language models: A survey, 2024. URL <https://arxiv.org/abs/2405.17935>.  
611
- 612 Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and  
613 Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model,  
614 2023. URL <https://arxiv.org/abs/2305.18290>.  
615
- 616 Stuart J. Russell and Peter Norvig. Artificial intelligence: A modern approach, 2020.  
617
- 618 Soham Sane. Am-ppo: (advantage) alpha-modulation with proximal policy optimization, 2025.  
619 URL <https://arxiv.org/abs/2505.15514>.  
620
- 621 Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer,  
622 Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to  
623 use tools, 2023. URL <https://arxiv.org/abs/2302.04761>.
- 624 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy  
625 optimization algorithms, 2017. URL <https://arxiv.org/abs/1707.06347>.
- 626 Amazon Web Services. Retry with backoff pattern - aws prescriptive guidance, 2024a.  
627 URL [https://docs.aws.amazon.com/prescriptive-guidance/latest/  
628 cloud-design-patterns/retry-backoff.html](https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/retry-backoff.html).  
629
- 630 Amazon Web Services. Timeouts, retries and backoff with jitter - the amazon  
631 builders’ library, 2024b. URL [https://aws.amazon.com/builders-library/  
632 timeout-retries-and-backoff-with-jitter/](https://aws.amazon.com/builders-library/timeout-retries-and-backoff-with-jitter/).
- 633 Zeyang Sha, Hanling Tian, Zhuoer Xu, Shiwen Cui, Changhua Meng, and Weiqiang Wang. Agent  
634 safety alignment via reinforcement learning, 2025. URL [https://arxiv.org/abs/2507.  
635 08270](https://arxiv.org/abs/2507.08270).
- 636 Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Huggingpt:  
637 Solving ai tasks with chatgpt and its friends in hugging face, 2023. URL <https://arxiv.org/abs/2303.17580>.  
638
- 639 Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. Agentic reasoning and tool  
640 integration for llms via reinforcement learning, 2025. URL [https://arxiv.org/abs/  
641 2505.01441](https://arxiv.org/abs/2505.01441).  
642
- 643 W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Unix network programming, volume 1:  
644 The sockets networking api, 2013.  
645
- 646 Lijun Sun, Yijun Yang, Qiqi Duan, Yuhui Shi, Chao Lyu, Yu-Cheng Cheng, et al. Multi-agent  
647 coordination across diverse applications: A survey, 2025. URL [https://arxiv.org/abs/  
2502.14743](https://arxiv.org/abs/2502.14743).

- 648 Zhensu Sun, Haotian Zhu, Bowen Xu, Xiaoning Du, Li Li, and David Lo. Llm as runtime error  
649 handler: A promising pathway to adaptive self-healing of software systems, 2024. URL <https://arxiv.org/abs/2408.01055>.  
650  
651
- 652 Andrew S. Tanenbaum and David J. Wetherall. Computer networks, 2016.
- 653 Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée  
654 Lacroix, et al. Llama: Open and efficient foundation language models, 2023a. URL <https://arxiv.org/abs/2302.13971>.  
655  
656
- 657 Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, et al.  
658 Llama 2: Open foundation and fine-tuned chat models, 2023b. URL <https://arxiv.org/abs/2307.09288>.  
659
- 660 Xincheng Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao.  
661 Aegis: An agent-based framework for general bug reproduction from issue descriptions, 2024.  
662 URL <https://arxiv.org/abs/2411.18015>.  
663
- 664 Wikipedia. Exponential backoff, 2024. URL [https://en.wikipedia.org/wiki/Exponential\\_backoff](https://en.wikipedia.org/wiki/Exponential_backoff).  
665
- 666 Yingxuan Yang, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, and otherse. A  
667 survey of ai agent protocols, 2025. URL <https://arxiv.org/abs/2504.16736>.  
668
- 669 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao.  
670 React: Synergizing reasoning and acting in language models, 2022. URL <https://arxiv.org/abs/2210.03629>.  
671
- 672 Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik  
673 Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.  
674 URL <https://arxiv.org/abs/2305.10601>.  
675
- 676 Tian Ye, Zicheng Xu, Yuanzhi Li, and Zeyuan Allen-Zhu. Physics of language models: Part 2.2,  
677 how to learn from mistakes on grade-school math problems, 2024. URL <https://arxiv.org/abs/2408.16293>.  
678
- 679 Peihong Yu, Borham Lee, Aswin Raghavan, Supuri Samarasekara, Pratap Tokekar, and  
680 James Zachary Hare. Enhancing multi-agent coordination through common operating picture  
681 integration, 2023. URL <https://arxiv.org/abs/2311.04740>.
- 682 Peijie Yu, Yifan Yang, Jinjian Li, Zelong Zhang, Haorui Wang, Xiao Feng, et al. Multi-mission tool  
683 bench: Assessing the robustness of llm based agents through related and dynamic missions, 2025.  
684 URL <https://arxiv.org/abs/2504.02623>.  
685
- 686 Yirong Zeng, Xiao Ding, Yuxian Wang, Weiwen Liu, Wu Ning, Yutai Hou, et al. itool: Reinforced  
687 fine-tuning with dynamic deficiency calibration for advanced tool use, 2025. URL <https://arxiv.org/abs/2501.09766>.  
688
- 689 Kun Zhang, Le Wu, Kui Yu, Guangyi Lv, and Dacao Zhang. Evaluating and improving robustness  
690 in large language models: A survey and future directions, 2025. URL <https://arxiv.org/abs/2506.11111>.  
691
- 692 Wanjia Zhao, Mert Yuksekgonul, Shirley Wu, and James Zou. Sirius: Self-improving multi-agent  
693 systems via bootstrapped reasoning, 2025. URL <https://arxiv.org/abs/2502.04780>.  
694
- 695 Rui Zheng, Shihan Dou, Songyang Gao, Yuan Hua, Wei Shen, Binghai Wang, et al. Secrets of rlhf in  
696 large language models part i: Ppo, 2023. URL <https://arxiv.org/abs/2307.04964>.  
697
- 698 Jingwen Zhou, Jieshan Chen, Qinghua Lu, Dehai Zhao, and Liming Zhu. Shielda: Structured  
699 handling of exceptions in llm-driven agentic workflows, 2025. URL <https://arxiv.org/abs/2508.07935>.  
700  
701

## Appendix

### 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: SERVICEDEPENDENCYFAILURE

**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 underspecification 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

Specific Failure	Example (Simulated Output)	Corrective Action Policy	Rationale and Citations
400 Bad Request	"error": "Malformed request syntax", "status": 400	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": 401	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": 403	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": 404	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": 500	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": 503	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": 429	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 2: Catalog of common failures, simulated outputs, and recovery strategies.

These are just 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://github.com/HexaA2/paladin/tree/main>.

## D SYSTEMATICALLY AUGMENTED TRACES

Systematically augmented traces are clean ToolBench style agents–tool execution logs that are enriched with controlled, labeled failures and paired with different recovery trajectories, helping enable training that was reproducible and the evaluation of different recovery behaviors under realistic tool-level faults. This approach helped yield multiple deterministic variants of the same base rollout by injecting specific error types at different steps, with structured annotations for failure detection, diagnosis, and multi-turn correction. Augmentations were generated by applying a failure-policy dictionary over baseline rollouts to inject errors such as timeouts, 5xx responses, rate limits, malformed tool outputs, and partial results, with configurable positions, frequencies, and cascades. Each injected failure was paired with an annotated recovery path (diagnose, replan, parameter fix, retry, or tool-switch), helping yield trajectory-level supervision for LoRA-based SFT and deterministic evaluation via metrics like TSR, RR, CSR, and ES. Because variants are deterministic, they support apples-to-apples ablations (e.g., single- vs multi-fault) and stable comparisons across model sizes and training regimes.

- 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: {
  \"username\": \"nike\"
}"}]},
{"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 are publicly released at: [https://huggingface.co/datasets/SriVatsa123/Gemma\\_Ready\\_For\\_PALADIN/viewer/default/train?views%5B%5D=train&row=0](https://huggingface.co/datasets/SriVatsa123/Gemma_Ready_For_PALADIN/viewer/default/train?views%5B%5D=train&row=0).

## 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 2: Recovery Paths Dictionary

```

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."
  }
]

```

```

972     },
973     {
974         "from": "Assistant",
975         "value": (
976             "Thoughts: If no client-side issue found, consider API bugs or
977             server incompatibility.\n\n"
978             "Action: Contact support with detailed logs; implement retry with
979             exponential backoff."
980         )
981     },
982     {
983         "from": "function",
984         "value": "Escalation logs prepared; retry logic implemented."
985     }
986 ],
987 # Branch 2: Authentication and Authorization Errors (401, 403, 407)
988 "401_403_407": [
989     {
990         "from": "Assistant",
991         "value": (
992             "Thoughts: 401 Unauthorized indicates missing or invalid
993             credentials, 403 Forbidden means access is denied "
994             "even if authenticated, and 407 Proxy Authentication Required
995             means the proxy server requires authentication.\n\n"
996             "Action: Check that valid API keys or tokens are sent in headers
997             exactly as required by the service. Refresh "
998             "tokens if expired. Confirm permissions on the user or API key to
999             access requested resources.\n\n"
1000             "For 403, verify role-based access controls and resource-specific
1001             permissions. For 407, ensure proxy credentials "
1002             "are configured and included properly in requests."
1003         )
1004     },
1005     {
1006         "from": "function",
1007         "value": "Credentials and permissions checked; tokens refreshed
1008             where needed."
1009     }
1010 ],
1011 {
1012     "from": "Assistant",
1013     "value": (
1014         "Thoughts: Persistent 401/403 might be due to time skew causing
1015         token rejection or revoked credentials.\n\n"
1016         "Action: Sync system clock with NTP, regenerate credentials, and
1017         check for IP restrictions or rate limits "
1018         "that may block access."
1019     )
1020 },
1021 {
1022     "from": "function",
1023     "value": "System time synchronized; no IP blocks detected."
1024 },
1025 {
1026     "from": "Assistant",
1027     "value": (
1028         "Thoughts: If proxy authentication fails, check proxy server logs
1029         and credentials.\n\n"
1030         "Action: Update proxy config or switch to a proxy that supports
1031         needed authentication methods."
1032     )
1033 },
1034 {
1035     "from": "function",
1036     "value": "Proxy credentials updated; connection succeeded."
1037 }

```

1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079

```

    }
  ],
}

```

The rest of our dictionary can be found in github at <https://github.com/HexaA2/paladin>.

## 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  $2 \times 10^{-4}$ , 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:

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

while error-free traces were finalized via:

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

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`  $\rightarrow$  `Action`  $\rightarrow$  `Action Input`  $\rightarrow$  `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.

## 1134 J PALADINEVAL BENCHMARK

1135

1136 PaladinEval is a deterministic failure-injection benchmark designed to evaluate recovery compe-  
1137 tence, honesty, and efficiency under the seven ToolScan error classes. It combines controlled simu-  
1138 lators, taxonomic labeling, and standardized metrics (TSR, RR, CSR, ES) to enable apples-to-apples  
1139 comparison across models and methods. Unlike ToolReflectEval, which targets single-call reflect-  
1140 ive corrections, PaladinEval emphasizes trajectory-level, multi-turn recovery in noisy execution  
1141 settings. All reported scores are normalized so that higher is better across tables and plots.

1142

### 1143 J.1 BENCHMARK DESIGN

1144

1145 PaladinEval instruments ToolBench-style tasks with a simulator that injects a single, labeled execu-  
1146 tion failure at a specified turn, then drives the episode to completion while logging recovery attempts,  
1147 retries, tool switches, and termination decisions. Each episode includes the task specification, tool  
1148 schema, truncated trace at failure, and an injected error drawn from the ToolScan taxonomy. Eval-  
1149 uation is conducted with fixed seeds and deterministic tool outputs to ensure reproducibility across  
1150 backbones and runs.

1151

### 1152 J.2 TASKS AND COVERAGE

1153 The suite spans diverse tool-use tasks representative of training domains and failure surfaces.  
1154 Episodes are constructed to cover all seven ToolScan error categories under uniform sampling  
1155 rules, preventing skew toward any particular class. Results reported in the main paper compare  
1156 PaladinEval against ToolReflectEval across multiple backbones, showing consistent gains in Re-  
1157 covery Rate, Task Success Rate, and Catastrophic Success Rate. These improvements come with  
1158 expected efficiency trade-offs from retry-heavy strategies, but confirm sufficient breadth and balance  
1159 across failure classes for comparative evaluation.

1160

### 1161 J.3 SAMPLING PROCEDURE

1162 For each task, the first tool failure is injected deterministically by specifying the error class, man-  
1163 ifestation (e.g., 5xx timeout, malformed JSON), and turn index. The remainder of the episode is  
1164 executed with fixed tool responses to recovery actions, minimizing variance. Sampling ensures  
1165 per-class coverage across *Tool Hallucination*, *Argument Hallucination*, *Invalid Invocation*, *Partial*  
1166 *Execution*, *Output Hallucination*, *Invalid Intermediate Reasoning*, and *Re-entrant Failures*, enabling  
1167 both per-class and macro-averaged reporting of RR, TSR, CSR, and ES.

1168

### 1169 J.4 ADAPTING TOOLREFLECTEVAL

1170

1171 For comparability, ToolReflectEval was re-run under the same deterministic simulator and normal-  
1172 ized metrics. While ToolReflectEval emphasizes critique-based improvements to single tool calls,  
1173 PaladinEval stresses multi-turn recovery after explicit failures, capturing behaviors such as diag-  
1174 nosis, replanning, retries, tool switches, and graceful termination. Together, the two benchmarks  
1175 provide complementary perspectives on robustness.

1176

### 1177 J.5 FILTERING AND DEDUPLICATION

1178 Benchmark construction applies truncation at the first failure and removes trajectories with ambigu-  
1179 ous or duplicate error signatures to avoid double-counting or conflating error classes. Episodes with  
1180 inconsistent tool schemas or non-reproducible outputs are excluded to preserve determinism. Clean  
1181 “happy-path” episodes are retained for competence checks but not scored as recoveries, ensuring  
1182 evaluation focuses squarely on execution robustness.

1183

1184

1185

1186

1187

## K ADDITIONAL METRICS AND ABLATIONS

### K.1 METRIC VARIANTS AND FORMULAS

We evaluate PALADIN with four metrics:

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

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

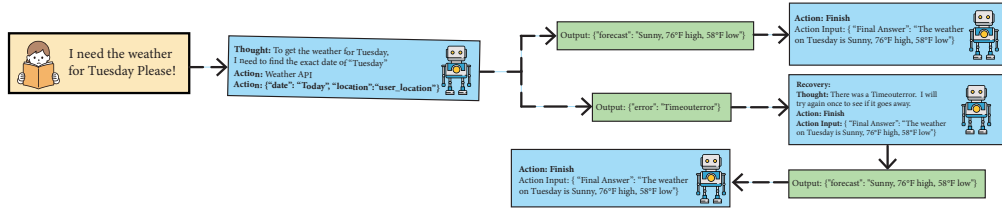


Figure 5: PALADIN’s Thought Process

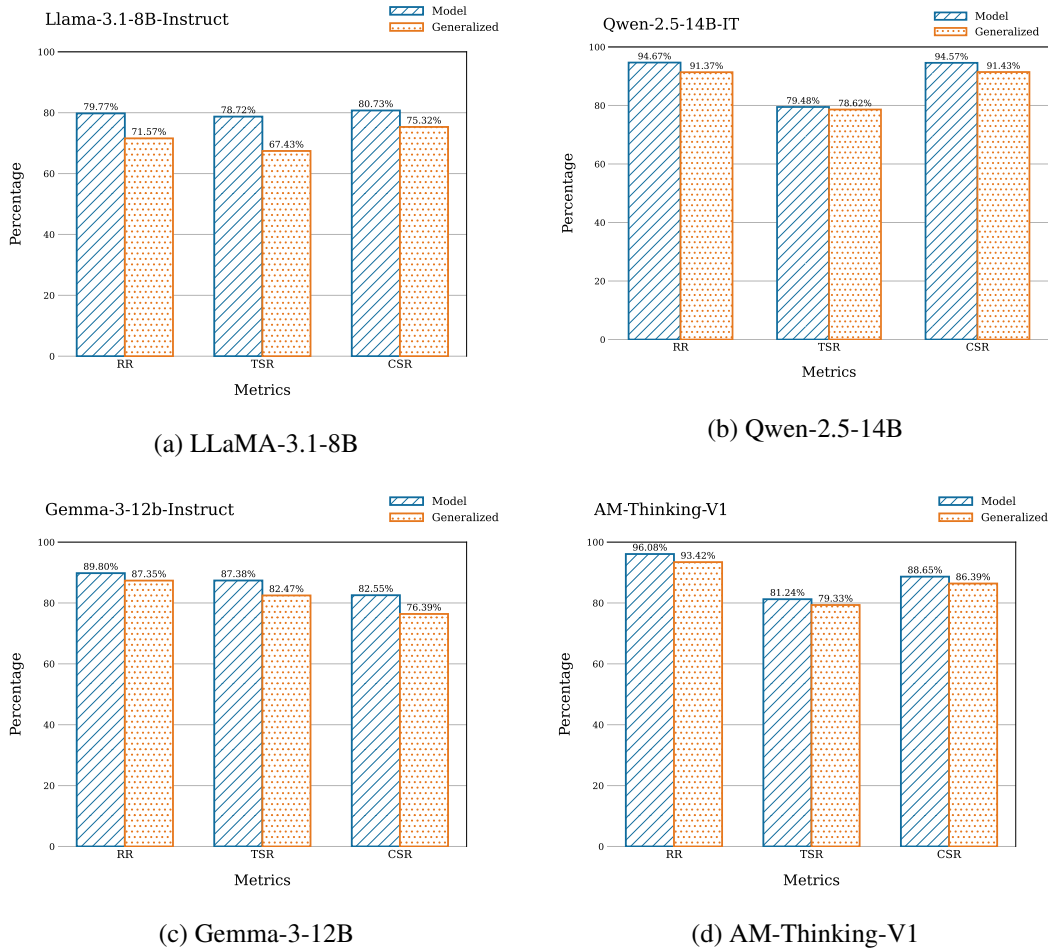


Figure 6: PALADIN’s robustness when facing unseen error types across different model backbones.

1296  
 1297  
 1298  
 1299  
 1300  
 1301  
 1302  
 1303  
 1304  
 1305  
 1306  
 1307  
 1308  
 1309  
 1310  
 1311  
 1312  
 1313  
 1314  
 1315  
 1316  
 1317  
 1318  
 1319  
 1320  
 1321  
 1322  
 1323  
 1324  
 1325  
 1326  
 1327  
 1328  
 1329  
 1330  
 1331  
 1332  
 1333  
 1334  
 1335  
 1336  
 1337  
 1338  
 1339  
 1340  
 1341  
 1342  
 1343  
 1344  
 1345  
 1346  
 1347  
 1348  
 1349

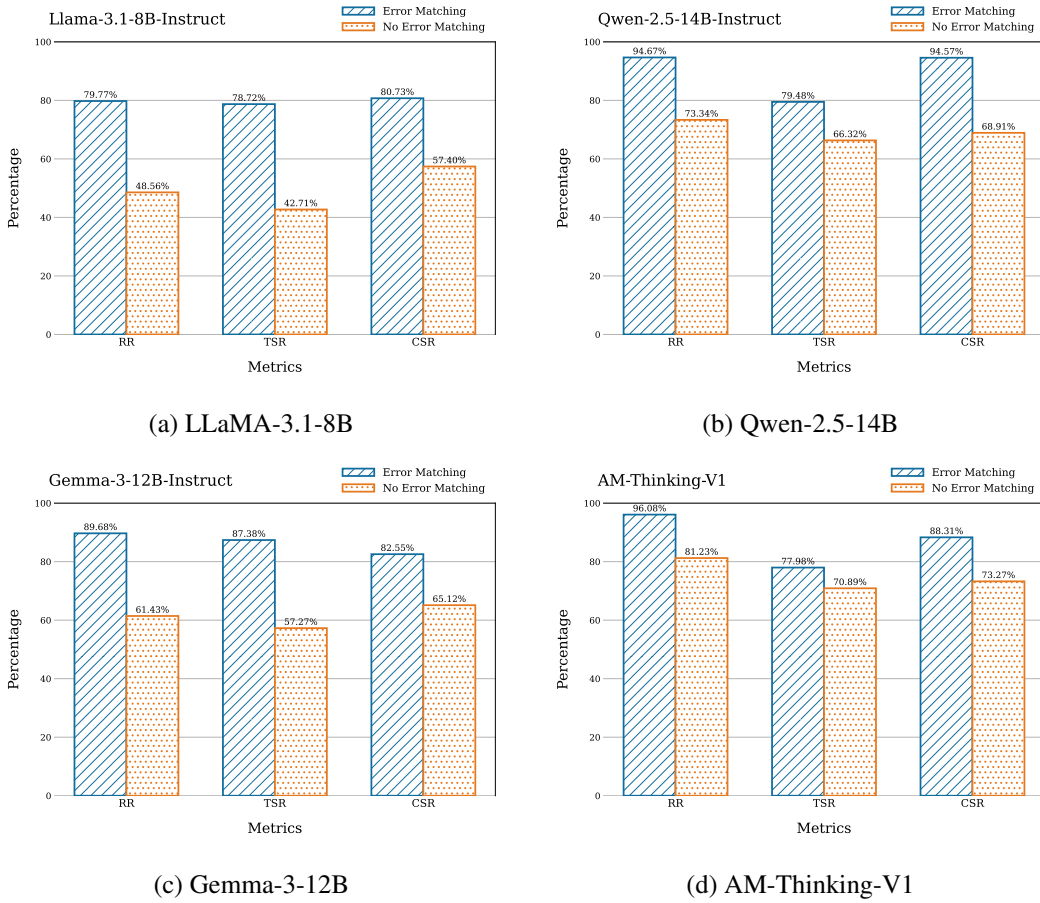


Figure 7: PALADIN’s ablation when facing unseen error types across different model backbones.