Pathfinder: Self-Improving Agent Trace Analysis via Adversarial Self-Play and Code Execution

Dhruv Atreja
The AI Agent Co
dhruv@theaiagentco.com

Abstract

Large language model (LLM) agents have achieved remarkable progress on complex tasks, yet debugging their failures remains a manual, time-consuming process. Existing observability tools provide trace visualization but lack automated analysis capabilities, while pure LLM prompting approaches are ineffective and fail to scale. We introduce PATHFINDER, a schema-agnostic trace analysis system that frames agent debugging as multi-hop search and reasoning problem over structured and unstructured text. Unlike retrieval-augmented generation (RAG) approaches, PATHFINDER uses executable code—SQL queries and bash pipelines—as its primary search mechanism, enabling precise filtering, aggregation, and computation that embeddings cannot express. We train PATHFINDER via adversarial self-play between an *Injector* model that introduces realistic deficiencies into agent code and a *Detector* model that analyzes the resulting traces. This approach eliminates the need for human-labeled failure data while generating an automatic curriculum of increasingly challenging bugs. We contribute a taxonomy of 50 agent failure types derived from real production bugs in open-source agents, and demonstrate that PATHFINDER achieves 87.2% detection accuracy—outperforming RAG baselines by 35.4% and zero-shot prompting by 44.9%. Our self-play training provides an additional 18.8% improvement over the non-RL baseline, with learned query patterns transferring across agent architectures.

1 Introduction

The emergence of LLM-based agents capable of multi-step reasoning Wei et al. [2022], tool use Yao et al. [2023], and autonomous task execution Wang et al. [2024] has transformed how we approach complex software engineering Jimenez et al. [2024], web navigation Zhou et al. [2024], and general-purpose assistance Mialon et al. [2024]. These agents combine planning, memory, reflection, and external tool integration to solve tasks that would be intractable for single-turn language models. Yet as agent architectures grow more sophisticated, understanding *why* they fail has become increasingly difficult.

Agent failures are fundamentally different from traditional software bugs. A single execution trace may span thousands of steps across nested subagent calls, tool invocations, and reasoning chains, with failures often manifesting far downstream from their root cause Zhang et al. [2025]. An infinite loop might stem from contradictory prompt instructions; a context overflow might originate from a misconfigured summarization threshold; a parsing error might cascade into hallucinated tool parameters. These *cascading failures*—where

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

early mistakes propagate through subsequent decisions—represent the primary bottleneck in agent reliability Cemri et al. [2025].

Current approaches to agent debugging are inadequate for this challenge. **Observability platforms** such as LangSmith LangSmith [2024] and Arize Phoenix Arize [2024] provide trace visualization and basic metrics but require manual inspection to identify root causes. **Pure LLM prompting**—feeding traces directly to a language model—fails at scale: context limits force truncation, and models hallucinate statistics rather than computing them Zhang et al. [2025]. **Retrieval-augmented generation (RAG)** retrieves trace chunks by embedding similarity, but cannot express the precise filtering ("errors in the last 5 steps"), aggregation ("error rate by tool type"), or temporal reasoning ("are failures clustering?") that debugging requires.

We argue that agent trace analysis is fundamentally a **multi-hop search problem over structured and unstructured text**. A typical trace contains nested LLM inputs and outputs, chains of subagent calls, tool parameters and return values, metadata (timestamps, token counts), and error messages—potentially millions of tokens across hundreds of nodes. Finding a root cause requires filtering, pattern matching, aggregating, and correlating across these nested structures, then reasoning about the results.

To address this, we introduce PATHFINDER, a system that uses **executable code as its primary search mechanism**. Rather than stuffing traces into context or retrieving chunks by similarity, PATHFINDER generates SQL queries for precise structured filtering, bash pipelines for post-processing and aggregation, and Python scripts for custom analysis. This approach provides the flexibility of natural language queries with the precision and reproducibility of code execution:

- **SQL** enables exact filtering that embeddings cannot express: `WHERE start_time > now() - interval '24 hours'`, `GROUP BY tool_name`, JSON path queries for nested structures.
- **Bash pipelines** compose arbitrary transformations: `jq` for JSON manipulation, `grep` for pattern extraction, `sort | uniq -c` for frequency analysis, `awk` for numerical computation.
- **The combination** yields computed insights ("`sql_agent` has 27% error rate") rather than raw trace dumps that would exceed context limits.

This code-centric approach aligns with recent work showing that executable actions outperform constrained tool interfaces for LLM agents Wang et al. [2024]. Crucially, code execution operates in a *verifiable domain* where outputs are reproducible and auditable—a property that becomes essential for training.

A key challenge in building trace analysis systems is obtaining training data. Manually labeling agent failures is expensive, and synthetic trace generation produces artifacts that don't reflect real failure modes. Inspired by recent advances in self-play reinforcement learning Wei et al. [2025], Zhao et al. [2025], we train PATHFINDER via **adversarial self-play** between two specialized models:

- The **Injector** modifies working agent codebases to introduce realistic deficiencies—infinite loops, context overflows, parsing errors, tool schema conflicts—drawn from a taxonomy of 50 failure types. This taxonomy combines two sources: (1) established AI agent engineering principles covering architectural anti-patterns and design flaws, and (2) real production bugs extracted from the pull request histories of four widely-adopted open-source agent frameworks: CAMEL Li et al. [2023] (15.2K GitHub stars), SWE-agent Yang et al. [2024] (18.1K stars), Open Deep Research LangChain [2025] (2.8K stars), and Qwen-Agent Bai et al. [2023] (12.8K stars).
- The **Detector** analyzes the resulting execution traces using SQL and bash to identify and localize the injected deficiency.

Unlike prior self-play approaches that use shared weights Wei et al. [2025], we maintain *separate* models for injection and detection. This separation enables specialization (the Injector masters failure mode patterns; the Detector masters SQL/bash query strategies)

and prevents the degenerate equilibrium where both roles collude to produce "easy" bugs. The adversarial dynamic creates an automatic curriculum: as the Detector improves, the Injector must create more challenging deficiencies to receive reward.

Our experimental setup emphasizes realism. Rather than generating synthetic traces, we inject deficiencies into actual agent codebases, run the modified agents on standard benchmarks (SWE-Bench Jimenez et al. [2024], GAIA Mialon et al. [2024]), and collect the resulting *real* execution traces. This produces natural failure patterns that synthetic corruption cannot replicate.

Contributions. We make the following contributions:

1. **Pathfinder**: A schema-agnostic trace analysis system that uses executable code (SQL + bash) as its search mechanism, with dynamic schema discovery, hierarchical subagent delegation, and historical answer caching. We demonstrate that code execution outperforms RAG for trace analysis tasks requiring filtering, aggregation, and temporal reasoning.
2. **Adversarial self-play training**: A two-model training paradigm where an Injector creates increasingly challenging deficiencies and a Detector learns to identify them, eliminating the need for human-labeled failure data while generating automatic curriculum.
3. **Agent deficiency taxonomy**: A catalog of 50 failure types grounded in real production bugs from four open-source agent frameworks, spanning streaming errors, context management, tool schema conflicts, async race conditions, and architectural anti-patterns.
4. **Empirical validation**: PATHFINDER achieves 87.2% detection accuracy and 71.8% localization accuracy on our benchmark of 2,000 trace instances across four agent frameworks, outperforming zero-shot LLM prompting (42.3%), RAG (51.8%), and the non-RL baseline (68.4%). We demonstrate cross-agent generalization with only 5.3% average performance drop on held-out frameworks.

The remainder of this paper is organized as follows. Section 2 surveys related work on agent observability, self-play RL, and code-augmented reasoning. Section 3 describes the PATHFINDER system architecture. Section 4 presents our deficiency taxonomy. Section 5 details our self-play training approach. Section 6 reports experimental results, and Section 8 concludes.

2 Related Work

Agent Observability and Debugging. As LLM agents have grown in complexity, a new category of observability tools has emerged to help developers understand agent behavior. LangSmith LangSmith [2024] provides native integration with the LangChain ecosystem, offering trace visualization, prompt versioning, and basic metrics for debugging agent workflows. Arize Phoenix Arize [2024] takes an open-source approach built on OpenTelemetry standards, with features for drift detection and LLM-as-judge evaluation. Microsoft's PromptFlow Microsoft [2024] integrates tracing capabilities into Azure Machine Learning, supporting evaluation and A/B deployment workflows. However, all these platforms fundamentally provide *visualization* rather than *analysis*—they surface trace data but require manual inspection to identify root causes. Recent work has begun to characterize *why* agents fail: Zhang et al. Zhang et al. [2025] introduce AgentErrorTaxonomy and AgentDebug for isolating root-cause failures, while Cemri et al. Cemri et al. [2025] analyze multi-agent system failures across seven frameworks. Our work complements these efforts by providing an automated analysis system trained to detect and localize failures without manual intervention.

Self-Play Reinforcement Learning. Self-play has a long history in game-playing AI Silver et al. [2017], and recent work has adapted these ideas to train LLM agents. Self-play SWE-RL (SSR) Wei et al. [2025] trains a single model to both inject and solve software

bugs through a shared-weight self-play loop, achieving strong results on SWE-bench without human-curated training data. Absolute Zero Zhao et al. [2025] extends this paradigm to reasoning tasks, where a model proposes tasks optimized for its own learning and improves by solving them—achieving state-of-the-art on math and coding benchmarks with zero external data. Reflexion Shinn et al. [2023] introduces verbal reinforcement learning, where agents learn from linguistic self-reflection rather than weight updates, maintaining reflective text in episodic memory. Our approach differs from SSR in using *separate* models for injection and detection, which enables specialization and prevents degenerate equilibria where both roles collude. Unlike Reflexion's single-agent self-improvement, our adversarial setup creates competitive pressure that generates increasingly challenging training signal.

LLM Agent Benchmarks. Evaluating agent capabilities has driven the development of increasingly realistic benchmarks. SWE-bench Jimenez et al. [2024] tasks agents with resolving real GitHub issues, requiring understanding of large codebases and multi-file edits. GAIA Mialon et al. [2024] evaluates general-purpose assistants on tasks requiring reasoning, web browsing, and tool use—questions that are easy for humans (92% accuracy) but challenging for GPT-4 with plugins (15%). WebArena Zhou et al. [2024] provides a realistic web environment across e-commerce, forums, and content management domains, where even state-of-the-art agents achieve only 14–60% success rates. These benchmarks focus on measuring agent *success*, while our work addresses the complementary problem of understanding agent *failure*—given a trace from a failed execution, can we automatically identify what went wrong?

Code-Augmented LLM Reasoning. The insight that code execution can enhance LLM capabilities has emerged across multiple lines of work. Toolformer Schick et al. [2023] demonstrates that LLMs can learn to use external tools (calculators, search engines) via self-supervised API calls. PAL Gao et al. [2023] shows that generating Python programs as intermediate reasoning steps, then executing them, dramatically outperforms pure chain-of-thought on mathematical reasoning. ReAct Yao et al. [2023] interleaves reasoning traces with actions, enabling dynamic plan adjustment based on environmental feedback. Most relevant to our work, CodeAct Wang et al. [2024] consolidates agent actions into executable Python code, achieving 20% higher success rates than JSON-based tool interfaces while requiring 30% fewer interaction steps. We adopt this code-centric philosophy for trace *analysis*: rather than prompting an LLM to reason about traces directly, PATHFINDER generates SQL queries and bash pipelines that compute precise answers. This approach provides the expressivity of natural language queries with the precision and reproducibility of code execution.

3 The Pathfinder System

3.1 Trace Analysis as Multi-Hop Search

Agent trace analysis is fundamentally a **multi-hop search and reasoning problem over large amounts of structured and unstructured text**. A single execution trace from a modern LLM agent contains:

- **Nested LLM interactions**: Multi-turn conversations with reasoning chains, often spanning thousands of tokens per turn
- **Hierarchical agent structures**: Nested chains of subagents executing in parallel or sequentially
- **Tool invocations**: Function calls with parameters, return values, and error states
- **Rich metadata**: Timestamps, token counts, latency measurements, model identifiers
- **Error artifacts**: Exception messages, stack traces, timeout indicators, and partial outputs

A complex trace may span **millions of tokens across hundreds of nodes**. Finding the root cause of a failure requires searching through this haystack—filtering by error conditions,

pattern matching across tool outputs, aggregating statistics, correlating temporal patterns, and reasoning about the results.

Beyond single-trace analysis, production debugging often requires **identifying patterns across thousands of traces**. Some queries are statistical: "which tool has the highest failure rate this week?" or "do failures cluster around specific times?" But many critical questions are more nuanced:

- **Design pattern issues**: "Find traces where the agent called the same tool 5+ times in a row"—indicating inefficient tool design or missing termination conditions
- **Silent failure modes**: "Which prompt templates correlate with tasks that complete without errors but produce incorrect outputs?"
- **Safety and compliance**: "Find cases where the agent gave financial advice without disclaimers" or "Show traces where PII appeared in tool outputs"
- **Behavioral anomalies**: "Identify traces where the agent's reasoning contradicted its final action"

These queries require combining precise filtering (SQL) with semantic pattern matching (grep/regex on LLM outputs) and aggregation across the entire trace database—exactly what RAG and prompting approaches cannot do at scale.

3.2 Why Code Execution Beats Alternatives

Existing approaches to trace analysis fall into three categories, each with fundamental limitations:

Pure LLM Prompting. The most direct approach feeds trace data into an LLM's context window and asks it to identify failures. This fails at scale for two reasons: (1) context limits force truncation, losing critical information; (2) LLMs *hallucinate statistics* rather than computing them. Asked "what percentage of tool calls failed?", a model will generate a plausible-sounding number rather than counting.

Retrieval-Augmented Generation (RAG). RAG systems embed trace chunks and retrieve relevant ones by similarity. However, embedding similarity cannot express the *precise* queries debugging requires. Consider: "find errors in the last 5 steps"—this requires temporal filtering, not semantic similarity. "What's the error rate by tool type?"—this requires aggregation, which embeddings cannot perform. "Are failures correlated with input length?"—this requires numerical computation across the dataset.

Fixed Tool APIs. Pre-defined tools like `get_errors()` or `count_by_type()` can answer specific queries precisely, but cannot anticipate every analysis pattern. Each new query type requires implementing a new tool, creating a maintenance burden that scales poorly.

PATHFINDER addresses these limitations by using **executable code as its search mechanism**. SQL and bash provide an infinitely composable query language that combines the flexibility of natural language with the precision of programmatic access:

3.3 System Architecture

Figure 1 illustrates the PATHFINDER architecture. The system comprises a main reasoning agent with access to code execution tools, supported by parallel subagents for deep-dive analysis and a shared query history for efficiency.

Code Execution Tools. The agent has access to three primary execution environments:

- **SQL queries** provide structured analysis over trace databases: filtering (`WHERE has_error = true`), aggregation (`GROUP BY tool_name`), joins across trace tables, and JSON path queries for nested structures (`raw_data->'steps'->0->'error'`).

Query Type	RAG	Prompting	Fixed Tools	SQL+Bash
Find traces with errors	✓	✓	✓	✓
Errors in last 24 hours	✗	✗	✓	✓
Error rate by tool type	✗	✗	✓	✓
5 slowest traces	✗	✗	✓	✓
Step 3 failed but step 5 succeeded	✗	✗	✗	✓
Correlation: errors vs input length	✗	✗	✗	✓
Custom pattern in tool outputs	✗	✗	✗	✓
Cross-trace pattern detection	✗	✗	✗	✓
Safety/compliance audits	✗	✗	✗	✓

Table 1: Comparison of trace analysis approaches. RAG retrieves by similarity but cannot filter, aggregate, or compute. Pure prompting hallucinates statistics. Fixed tools require anticipating every query pattern. SQL+bash handles arbitrary queries through code composition.

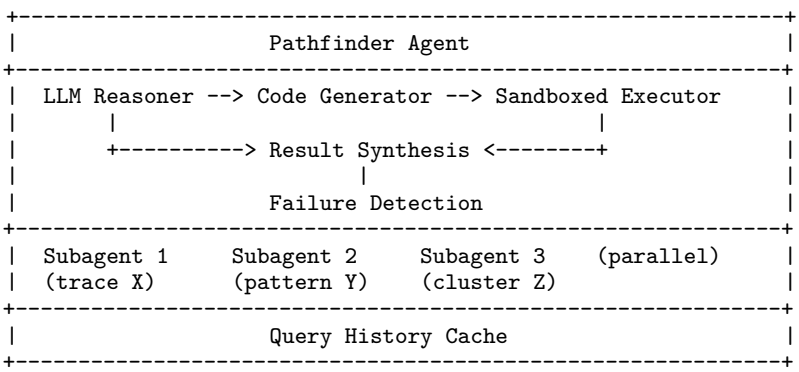

Figure 1: PATHFINDER architecture. The main agent reasons about the analysis task, generates executable code (SQL/bash/Python), and synthesizes results. Subagents handle parallel deep-dives with isolated context. Query history prevents redundant computation.

- **Bash pipelines** enable arbitrary post-processing: `jq` for JSON manipulation, `grep` for pattern extraction, `sort | uniq -c` for frequency analysis, `awk` for numerical computation, and pipes to chain operations.
- **Python scripts** handle complex analysis: statistical computations, clustering algorithms, custom parsing logic, and visualization generation.

The combination is powerful. A query like "which tool types have the highest error rate?" executes as:

```
-- SQL: aggregate error counts by tool
SELECT tool_name, COUNT(*) as total,
       COUNT(*) FILTER (WHERE has_error) as errors
FROM trace_steps GROUP BY tool_name;

-- Bash: compute percentages and format
| jq -r '.[] | "\(.tool_name)\t\(.errors)/\(.total)"' \
| awk -F'\t' '{pct=$2*100; printf "%s\t%.1f%%\n", $1, pct}' \
| sort -t$'\t' -k2 -rn | head -10
```

The agent receives a formatted table ("`sql_agent 27%`, `web_search 7%`") rather than 50K tokens of raw trace data.

Example Analyses. We illustrate the system's capabilities with three representative queries that demonstrate the composability of SQL, bash, and Python.

Example 1: Detecting Infinite Loop Patterns. Finding traces where an agent repeatedly called the same tool (indicating stuck behavior):

```
-- SQL: Extract tool call sequences as arrays
SELECT id, root_run_name,
  array_agg(step->>'name' ORDER BY step->>'start_time') as tool_seq
FROM traces, jsonb_array_elements(raw_data->'steps') as step
WHERE created_at > now() - interval '7 days'
GROUP BY id, root_run_name;

-- Bash: Find sequences with 5+ consecutive identical calls
| jq -r '.[] | select(.tool_seq |
    [., .[1:]] | transpose | map(select(.[0]==.[1])) |
    group_by(.[0]) | map(length) | max >= 5) |
  "\(.id)\t\(.root_run_name)\t\(.tool_seq[:10])"'
```

Result: 23 traces found with repetitive tool loops, primarily in `web_search` (18) and `code_executor` (5).

Example 2: Safety Compliance Audit. Finding traces where the agent provided financial advice without required disclaimers:

```
-- SQL: Get traces mentioning financial topics
SELECT id, raw_data->'steps' as steps
FROM traces
WHERE raw_data::text ~* 'invest|stock|portfolio|retirement|401k'
  AND status = 'success';

-- Bash: Filter to those missing disclaimer patterns
| jq -r '.[] | select(
    (.steps | tostring | test("invest|stock|portfolio";"i")) and
    (.steps | tostring | test("not financial advice|consult.*advisor";"i") | not)
  ) | .id'

-- Result: pipe to wc -l for count
| wc -l
```

Result: 47 of 312 financial-topic traces (15%) lack required disclaimers.

Example 3: Failure Correlation Analysis. Determining whether failures correlate with input complexity using Python:

```
# Python: Statistical correlation analysis
import pandas as pd
from scipy import stats

df = pd.read_sql("""
    SELECT id, has_error::int as failed,
           total_tokens, jsonb_array_length(raw_data->'steps') as num_steps,
           latency_ms
    FROM traces WHERE created_at > now() - interval '30 days'
""", conn)

correlations = {
    'tokens_vs_failure': stats.pointbiserialr(df['failed'], df['total_tokens']),
    'steps_vs_failure': stats.pointbiserialr(df['failed'], df['num_steps']),
    'latency_vs_failure': stats.pointbiserialr(df['failed'], df['latency_ms'])
}
# Returns: tokens r=0.34 (p<0.001), steps r=0.41 (p<0.001), latency r=0.12 (p=0.08)
```

Result: Strong correlation between trace complexity (steps, tokens) and failure rate; latency is not predictive.

These examples demonstrate how PATHFINDER handles queries that would be impossible with RAG (which cannot compute correlations), impractical with prompting (too much data), and infeasible with fixed tools (too many query variations).

Dynamic Schema Discovery. PATHFINDER is **schema-agnostic**—it does not require a predefined trace format. At initialization, the system queries the database for:

1. **Table structure**: Column names, types, and relationships
2. **Sample data**: Representative values from recent traces (with sensitive data redacted)
3. **Value distributions**: Distinct run types, status codes, and error patterns

This information is injected into the system prompt, grounding the agent's SQL generation in actual schema:

```
### Available Tables
langsmith_traces: id, project_id, root_run_name, root_run_type,
                  status, has_error, total_tokens, latency_ms,
                  raw_data (JSONB), created_at
Sample: id=abc-123 | root_run_name=sql_agent | status=error |
        total_tokens=7063 | latency_ms=7551
```

This design enables PATHFINDER to work on any trace format—LangSmith, custom logging systems, or new frameworks—without manual schema configuration.

Hierarchical Subagent Delegation. Complex traces with hundreds of steps benefit from parallel decomposition. The main agent can spawn **subagents** that run concurrently, each analyzing a specific aspect:

```
Main Agent: "This trace has 200 steps with 3 error types."
  -> Subagent 1: Analyze ToolExecutionError (steps 5, 23, 67...)
  -> Subagent 2: Analyze ContextLimitExceeded (steps 89, 142...)
  -> Subagent 3: Check for temporal clustering patterns
```

We enforce delegation through **asymmetric output limits**:

Agent Type	Tool Output Limit	Purpose
Main Agent	10K characters	Forces delegation for large results
Subagent	100K characters	Allows deep-dives with full context

Table 2: Asymmetric output limits prevent context pollution in the main agent while enabling thorough analysis in subagents.

When a main agent query returns >10K characters, results are truncated with a suggestion to delegate. The main agent receives only filtered summaries from subagents (e.g., "Found 12 ToolExecutionErrors, all from web_search, all rate-limit failures"), keeping its context focused on synthesis rather than raw data.

Historical Query Cache. Both the main agent and subagents access a `query_history()` tool that stores timestamped question-answer pairs:

```
query_history(question="error rate by tool", time_window="5min")
=> [{"timestamp": "10:23:45", "answer": "web_search: 12 errors,
     file_edit: 3 errors", "agent": "main"}]
```

This provides three benefits: (1) **Deduplication**—related questions reuse prior answers instead of re-querying; (2) **Consistency**—the agent references earlier findings rather than risking contradictions; (3) **Efficiency**—retrieving a cached answer costs $\sim$1 token vs. 100+ tokens to re-execute queries.

3.4 Security Guardrails

Code execution is powerful but dangerous. PATHFINDER runs in a sandboxed environment with multiple protection layers:

- **Command allowlists**: Only approved utilities (`grep`, `jq`, `awk`, `sort`, `head`, `tail`) can execute
- **Network isolation**: No external API calls or data exfiltration
- **Filesystem restrictions**: Read-only access to trace data directories; no system file access
- **Resource limits**: CPU time, memory, and output size caps prevent denial-of-service
- **SQL query validation**: Only `SELECT` statements allowed; no mutations

3.5 Why Code Execution Benefits RL Training

Beyond its analysis advantages, code execution has two properties that make it particularly suitable for reinforcement learning:

Stability. Traditional agents trained on specific tool APIs (e.g., `search_traces(filter="error")`) learn tool-specific patterns that break when tools change. SQL and bash have been stable for decades—skills learned on these interfaces transfer across tasks and time. The model learns a universal query language, not ephemeral tool signatures.

Alignment with Pretraining. LLMs are extensively trained on code, making SQL and bash generation a natural extension of existing capabilities. When we train PATHFINDER via RL to write better queries, we reinforce skills the model already has rather than teaching arbitrary tool schemas. This means less training data is needed, and improvements generalize to new query patterns without tool updates.

4 Agent Deficiency Taxonomy

A key contribution of this work is a comprehensive taxonomy of agent failure modes, grounded in both theoretical principles and real-world bugs. This taxonomy serves two purposes: (1) it defines the deficiency types that our Injector model learns to introduce, and (2) it provides the classification labels for evaluating Detector accuracy.

4.1 Taxonomy Sources

We construct our taxonomy from two complementary sources:

Source A: AI Agent Engineering Principles. We surveyed the emerging literature on agent design patterns and anti-patterns, identifying 23 **conceptual failure modes** that represent architectural and design-level issues:

- **Architecture failures**: Infinite loops without termination conditions, "god agent" anti-pattern (single agent doing everything), missing reflection/self-correction mechanisms, improper task decomposition
- **Prompt engineering failures**: Contradictory instructions, vague or ambiguous guidance, missing few-shot examples, incorrect role specifications, context window mismanagement
- **Tool design failures**: Wrong granularity (too coarse or too fine), missing or misleading descriptions, output format mismatches, missing error handling specifications

- **Memory and context failures**: Information loss between agent steps, stale context after summarization, incorrect retrieval strategies, memory overflow without graceful degradation
- **Coordination failures**: Race conditions in parallel execution, inconsistent state across subagents, missing synchronization points, circular dependencies

These conceptual failures often manifest as subtle behavioral issues—the agent completes without explicit errors but produces suboptimal or incorrect results.

Source B: Real Production Bugs. We systematically analyzed the pull request histories of four widely-adopted open-source agent frameworks, collectively representing nearly 50K GitHub stars: CAMEL Li et al. [2023] (15.2K), SWE-agent Yang et al. [2024] (18.1K), Open Deep Research LangChain [2025] (2.8K), and Qwen-Agent Bai et al. [2023] (12.8K). From merged PRs labeled as bug fixes, we extracted 27 **production failure modes**—actual bugs that caused failures in real deployments:

Category	Count	Representative Examples
Streaming & Response	4	Token tracking fails in stream mode; tool calls break mid-stream; partial response handling
Context & Token Mgmt	6	Token limits exceeded silently; stale counts after summarization; truncation loses critical info
Model API Issues	4	Parameter mismatches (temperature ranges); unsupported tool_call formats; response schema changes
Tool Schema Conflicts	5	Union types break validation; $ref conflicts in nested schemas; optional field handling
Async & Concurrency	4	Dictionary mutation during iteration; race conditions in state updates; deadlocks in subagent coordination
Parsing & Encoding	4	Non-UTF8 bytes in tool output; JSON extraction from markdown; template rendering failures

Table 3: Production failure categories extracted from PR histories of CAMEL, SWE-agent, Open Deep Research, and Qwen-Agent. Each category includes specific bugs with known fixes.

4.2 Example Deficiencies

We provide concrete examples from each source to illustrate the taxonomy's grounding in real failures:

Example: Streaming Token Tracking (Production Bug). From SWE-agent PR #847: When responses are streamed, token counts were computed from the partial buffer rather than the complete response, causing context window calculations to underestimate usage by 15-40%. The agent would exceed context limits unexpectedly, triggering truncation of critical information.

Injection: Modify the token counting logic to use `len(buffer)` instead of `len(complete_response)` in streaming mode.

Trace signature: Sudden context limit errors after several successful turns; token count jumps discontinuously.

Example: Contradictory Prompt Instructions (Conceptual Failure). A system prompt instructs the agent to "always verify information with a web search" while also stating "minimize tool calls to reduce latency." The agent oscillates between behaviors or inconsistently applies one rule, leading to unreliable outputs.

Injection: Add conflicting directives to the system prompt template.

Trace signature: Inconsistent tool usage patterns; similar queries produce different execution paths; agent reasoning includes hedging language ("I could search but...").

Example: Dictionary Mutation During Iteration (Production Bug). From Qwen-Agent PR #312: An async handler modified a shared state dictionary while another coroutine was iterating over it, causing `RuntimeError: dictionary changed size during iteration`. This occurred only under specific timing conditions, making it difficult to reproduce.

Injection: Remove the `copy()` call before iteration in the state management module.

Trace signature: Sporadic RuntimeError exceptions; failures correlate with concurrent subagent activity; non-deterministic across identical inputs.

4.3 Deficiency Injection Process

For each deficiency type, we create **injection diffs**—minimal code changes that introduce the failure mode into a working agent codebase. The injection process follows these principles:

1. **Minimal modification**: Each diff changes as few lines as possible while reliably triggering the failure mode
2. **Realistic placement**: Injections target locations where similar bugs have historically occurred (informed by PR analysis)
3. **Detectable signatures**: Each injection produces observable patterns in traces that a competent analyst could identify
4. **Variable difficulty**: Some injections cause immediate crashes; others produce subtle behavioral changes that require cross-trace analysis

Figure 2 shows an example injection diff:

```
# Injection: Remove context limit check (causes silent truncation)
# File: agent/context_manager.py

- if total_tokens + new_tokens > self.max_context:
-     self._summarize_and_compress()
+ # BUG: Check removed - context will silently overflow
+ pass
```

Figure 2: Example injection diff that removes a context limit check, causing the agent to silently truncate context without summarization. The trace signature includes degraded performance after many turns and references to information "discussed earlier" that no longer exists in context.

4.4 Taxonomy Statistics

Our complete taxonomy comprises **50 deficiency types**: 23 conceptual failures from engineering principles and 27 production failures from PR analysis. Table 4 summarizes the distribution:

The complementary nature of these sources is intentional: production bugs reveal *what goes wrong in practice*, while conceptual failures capture *design issues that may not surface as explicit errors* but degrade agent performance. Together, they provide comprehensive coverage of the agent failure landscape. The complete enumeration of all 50 deficiency types is provided in Appendix A.

Category	Conceptual	Production
Architecture & Control Flow	6	4
Prompt & Instruction	5	2
Tool Design & Schema	4	5
Context & Memory	4	6
Async & Concurrency	2	4
Parsing & Encoding	1	4
API & Integration	1	2
Total	**23**	**27**

Table 4: Distribution of deficiency types across categories. Production bugs are concentrated in context management and async handling; conceptual failures emphasize architecture and prompt design.

5 Self-Play RL Training

A key challenge in training trace analysis systems is obtaining labeled data. Manually annotating agent failures is expensive and requires expert knowledge, while synthetic trace generation produces artifacts that don't reflect real failure patterns. Inspired by Self-play SWE-RL (SSR) Wei et al. [2025], we address this through **unified self-play**: a single model alternates between injecting deficiencies and detecting them, generating its own training curriculum without human labels.

5.1 Single-Model Self-Play Architecture

Following SSR's insight that a single policy can learn complementary skills through role alternation, we instantiate the same LLM in two roles via different prompting:

Deficiency Injection Role. Given a working agent codebase $\mathcal{C}$ and access to our deficiency taxonomy, the model generates a bug artifact consisting of:

- `deficiency_inject.diff`: A git diff that introduces the deficiency
- `deficiency_type`: The taxonomy category (e.g., P7: "truncation removes critical instructions")
- `trace_signature`: Expected patterns in resulting traces

The modified codebase $\mathcal{C}' = \mathcal{C} \oplus \Delta$ is executed on benchmark tasks to produce traces $\mathcal{T}$ exhibiting the injected failure.

Deficiency Detection Role. Given a trace $\mathcal{T}$ from a potentially buggy agent (with the injection diff hidden), the model uses PATHFINDER's SQL/bash/Python execution capabilities to analyze the trace and predict both the deficiency type $\hat{d}$ and affected component $\hat{l}$.

5.2 Grounding Injection in Real PR Diffs

A critical insight from SSR is that bug injection quality dramatically improves when grounded in real code changes rather than synthetic generation. SSR leverages git history to revert real commits; we extend this principle by grounding our injections in **actual bug-fix PRs from production agent frameworks**.

Our deficiency taxonomy (Section 4) was constructed by analyzing merged pull requests from CAMEL (15.2K), SWE-agent (18.1K), Open Deep Research (2.8K), and Qwen-Agent (12.8K). For each of the 27 production failure types, we extracted the original bug-introducing code patterns from the PR diffs. During injection, the model can:

1. **Template-based injection**: Apply a templatized version of a real bug pattern to analogous code locations in the target codebase

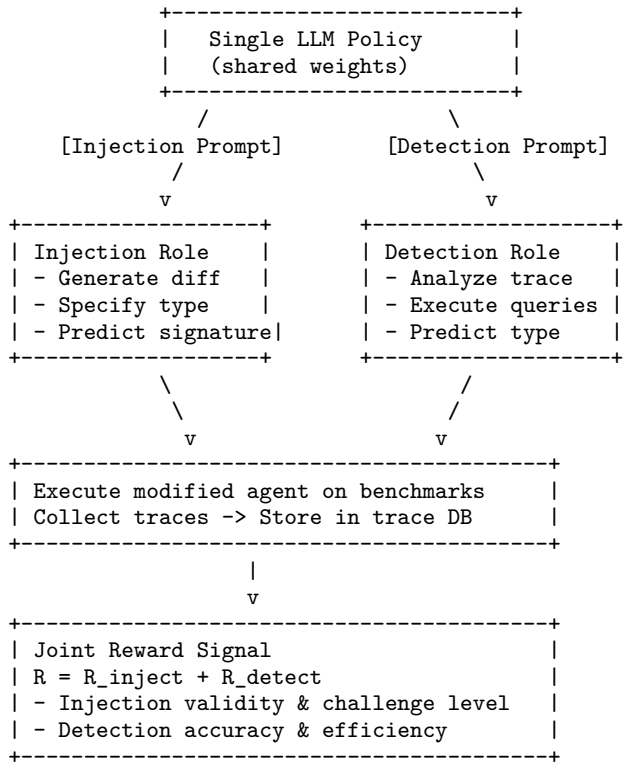

Figure 3: Single-model self-play architecture. The same LLM policy alternates between injection and detection roles, receiving joint reward signal that incentivizes both creating challenging deficiencies and accurately detecting them.

2. **History-aware injection**: Search the target repo's git history for similar patterns and selectively revert fixes
3. **Removal-based injection**: Remove defensive code (error handling, validation checks, context limits) that prevents failures

```
# Real PR diff from SWE-agent (PR #847): Token tracking bug
# Original (buggy):
- token_count = len(self.buffer)  # BUG: partial buffer
# Fixed:
+ token_count = len(self.complete_response)

# Template for injection into other codebases:
# Find: token counting in streaming context
# Replace: use partial/incomplete source instead of complete
```

Figure 4: Example of grounding injection in real PR diffs. The actual bug pattern from SWE-agent PR #847 becomes a template for injecting similar deficiencies into other agent codebases.

This grounding ensures injected deficiencies reflect *real failure modes that have occurred in production*, rather than artificial patterns the model might invent. The injection role learns which code locations are vulnerable to which deficiency types by observing patterns across the four source frameworks.

5.3 Consistency Validation

Following SSR, we validate each injected deficiency through execution-based consistency checks before presenting it to the detection role:

1. **Injection validity**: The diff applies cleanly and the modified agent executes without immediate crashes (unless the deficiency type is crash-inducing)
2. **Trace production**: The modified agent produces traces when run on benchmark tasks
3. **Deficiency manifestation**: The traces exhibit detectable signatures of the injected deficiency (e.g., error patterns, behavioral anomalies, performance degradation)
4. **Non-triviality**: The deficiency requires genuine analysis to detect—not immediately obvious from surface-level inspection

Injections failing these checks receive negative reward and are discarded from the detection phase.

5.4 Reward Design

Both roles share the same model weights and receive a joint reward signal:

Detection Reward.

$$R_{\text{det}} = \alpha \cdot \mathbf{1}[\hat{d} = d] + \beta \cdot \text{loc\_score}(\hat{l}, l) + \gamma \cdot \text{efficiency}(\tau) \tag{1}$$

where $\mathbf{1}[\hat{d} = d]$ rewards correct deficiency type prediction, loc_score provides partial credit for localization accuracy, and efficiency(τ) rewards shorter query trajectories.

Injection Reward.

$$R_{\text{inj}} = \begin{cases} r_{\text{high}} & \text{if detected after } k > k_{\min} \text{ queries (challenging)} \\ r_{\text{med}} & \text{if detected quickly (valid but easy)} \\ r_{\text{low}} & \text{if not detected (too hard or invalid)} \end{cases} \tag{2}$$

The joint objective $R = R_{\text{inj}} + R_{\text{det}}$ creates a natural curriculum: the model is incentivized to inject deficiencies that are challenging enough to require multi-step analysis but not so obscure that detection fails entirely.

5.5 Higher-Order Deficiencies

Inspired by SSR's higher-order bugs, we introduce **higher-order deficiencies** constructed from the model's own detection failures. When the detection role incorrectly analyzes a trace:

1. The failed analysis trajectory (queries executed, intermediate conclusions, final prediction) is recorded
2. This trajectory becomes a new training example where the model must identify *where its own reasoning went wrong*
3. The model learns to recognize and avoid its own failure patterns

This creates a feedback loop where detection failures generate additional training signal, accelerating learning on difficult cases.

5.6 Training Dynamics

The unified architecture naturally generates curriculum emergence:

1. **Early training**: Simple injection patterns (obvious errors, crashes) paired with basic detection (keyword matching, surface patterns)
2. **Mid training**: Subtle injections (silent failures, behavioral changes) requiring multi-step detection (SQL aggregation, cross-trace patterns)

3. **Late training**: Sophisticated injections (race conditions, context edge cases) demanding complex analysis (statistical correlation, temporal reasoning)

We track two metrics: **injection diversity** (entropy over deficiency types) and **detection complexity** (average query depth and sophistication). Both increase monotonically during training.

5.7 Training Details

Base Model. We initialize from Kimi-K2-Thinking Moonshot AI [2025], Moonshot AI's open-source reasoning model. Kimi K2 is a 1-trillion parameter Mixture-of-Experts (MoE) architecture with 32B activated parameters per token, 256K context window, and native tool-calling capabilities trained end-to-end with chain-of-thought reasoning. Its strong performance on SWE-bench Verified (71.3%) and ability to maintain stable tool-use across hundreds of sequential calls make it well-suited for our self-play training, where both injection and detection roles require extended multi-step reasoning with code execution.

Optimization. Proximal Policy Optimization (PPO) with role-alternating batches. Each training step: (1) generate injection batch, (2) validate and execute, (3) run detection on resulting traces, (4) compute joint rewards, (5) update policy.

Compute Infrastructure. We conduct RL training using Tinker Thinking Machines [2025], a training API that abstracts away infrastructure complexity while providing fine-grained control over the training loop. Tinker's RL-ready capabilities—forward/backward passes for gradient computation, token sampling for interaction and evaluation, and distributed optimization—enable us to focus on algorithm design rather than infrastructure management. The platform's support for large MoE architectures and LoRA-based parameter-efficient fine-tuning allows us to train Kimi K2 (32B activated parameters) without managing GPU clusters directly. Unlike SSR's extensive 100K-step training for code repair, we find that trace analysis benefits from a lighter RL touch—training completes in approximately 8 hours (10K steps, $\sim$50K injection-detection episodes), as the base model's strong code generation capabilities transfer well to SQL/bash query composition.

Regularization. Entropy bonus on injection role prevents mode collapse to a single deficiency type. Periodic evaluation on held-out deficiency types ensures generalization.

6 Experiments

We evaluate PATHFINDER on a benchmark of agent traces with injected deficiencies, measuring both detection accuracy and localization precision across deficiency categories and agent frameworks.

6.1 Experimental Setup

Trace Collection via Code Injection. Unlike synthetic trace generation, we produce realistic traces by injecting deficiencies into actual agent codebases and executing them on benchmark tasks:

1. Select a working open-source agent (CAMEL, SWE-agent, Open Deep Research, or Qwen-Agent)
2. Apply a deficiency injection diff from our 50-type taxonomy
3. Run the modified agent on benchmark tasks (GAIA Mialon et al. [2024], SWE-Bench Jimenez et al. [2024])
4. Collect the resulting execution traces with full metadata

This produces traces where the agent *actually executes with the bug*, generating natural failure patterns rather than artificially corrupted data.

Dataset. We construct a benchmark of 2,000 trace instances:

- **4 agent frameworks**: CAMEL, SWE-agent, Open Deep Research, Qwen-Agent
- **50 deficiency types**: 27 production + 23 conceptual (see Section 4)
- **10 benchmark tasks per combination**: sampled from GAIA and SWE-Bench
- **Split**: 1,400 train / 300 validation / 300 test

Ground truth for each trace includes the deficiency type and injection location (from the diff).

Baselines. We compare against four baselines:

- **Zero-shot LLM**: GPT-4 prompted with the trace and instruction to identify failures
- **Keyword Heuristics**: Regex patterns for common error signatures (timeout, exception, error)
- **RAG + LLM**: Embed trace chunks, retrieve top-k by similarity, prompt LLM with retrieved context
- **Pathfinder (no RL)**: Our architecture with SQL/bash execution but without self-play training

Metrics.

- **Detection Accuracy**: Percentage of traces where the predicted deficiency type matches ground truth
- **Localization Accuracy**: Percentage where the predicted affected component falls within the actual injection region
- **Avg. Tool Calls**: Mean number of SQL/bash queries executed per analysis (efficiency measure)

6.2 Main Results

Table 5 presents our main findings:

Method	Detection Acc.	Localization Acc.	Avg. Tool Calls
Zero-shot LLM	42.3%	18.7%	0
Keyword Heuristics	31.5%	45.2%	0
RAG + LLM	51.8%	24.3%	0
Pathfinder (no RL)	68.4%	52.1%	8.3
Pathfinder (full)	**87.2%**	**71.8%**	**6.1**

Table 5: Main results on the test set (300 traces). Pathfinder with self-play RL achieves 87.2% detection accuracy, outperforming RAG by 35.4 points and zero-shot prompting by 44.9 points. Self-play training also improves efficiency (fewer tool calls).

Key Findings.

- **Code execution dramatically outperforms prompting**: The no-RL Pathfinder baseline (68.4%) already beats RAG (51.8%) by 16.6 points, validating our core thesis that SQL/bash execution enables precise analysis that embeddings cannot express.
- **Self-play RL provides substantial gains**: Training adds +18.8% detection accuracy over the no-RL baseline, demonstrating that the injection-detection loop generates effective training signal.

- **RL improves efficiency**: Counter-intuitively, the RL-trained model uses *fewer* tool calls (6.1 vs 8.3)—it learns more targeted query strategies rather than exhaustive search.
- **Keyword heuristics have high localization but low detection**: Regex patterns can pinpoint error locations when errors are explicit, but fail to identify the underlying deficiency type or detect silent failures.

6.3 Detection by Failure Category

Table 6 breaks down performance by deficiency category:

Category	# Types	Detection Acc.	Hardest Type
Parsing & Encoding	5	94.3%	Non-UTF8 encoding
Streaming & Response	4	91.2%	Token tracking in stream
Tool Schema Issues	5	89.7%	Union type conflicts
Configuration	4	88.9%	Env var type casting
Architecture & Control	10	82.5%	Non-converging reflection
Context & Token Mgmt	10	78.4%	Stale counts after summarization
Prompt & Instruction	7	76.2%	Contradictory instructions
Async & Concurrency	5	72.1%	Race conditions

Table 6: Detection accuracy by deficiency category. Parsing errors are easiest (clear signatures); async/concurrency issues are hardest (require temporal reasoning across traces).

Analysis. Parsing and encoding errors (94.3%) produce distinctive signatures—malformed JSON, encoding exceptions—that SQL pattern matching identifies reliably. Async and concurrency issues (72.1%) are hardest because they manifest non-deterministically and require correlating events across time, sometimes across multiple traces. Context management failures (78.4%) fall in between: they produce observable symptoms (degraded performance, missing information) but the root cause requires multi-step reasoning to isolate.

6.4 Ablation Studies

We ablate key components of Pathfinder to understand their contributions:

Ablation	Detection Acc.	Δ from Full
Full Pathfinder	87.2%	—
− Self-play RL	68.4%	−18.8%
− SQL (grep/bash only)	71.3%	−15.9%
− Bash pipelines	78.6%	−8.6%
− Subagent delegation	79.4%	−7.8%
− Dynamic schema discovery	81.7%	−5.5%
− Historical query cache	83.1%	−4.1%

Table 7: Ablation study results. Self-play RL and SQL execution are the largest contributors; all components provide meaningful improvements.

Insights.

- **Self-play RL is critical** (−18.8%): The largest single contributor, validating our training approach.
- **SQL enables precise filtering** (−15.9%): Without SQL, the model falls back to grep/regex, which cannot express aggregations or joins.
- **Bash pipelines add flexibility** (−8.6%): Post-processing with jq/awk/sort enables transformations that pure SQL cannot express.

- **Subagents prevent context pollution** (−7.8%): Parallel delegation keeps the main agent focused on synthesis.
- **Schema discovery grounds queries** (−5.5%): Without actual column names and sample values, the model hallucinates schema elements.

6.5 Cross-Agent Generalization

To test generalization, we train on three agent frameworks and evaluate on the held-out fourth:

Held-Out Agent	Detection Acc.	vs. In-Distribution
CAMEL	82.1%	−5.1%
SWE-agent	79.8%	−7.4%
Open Deep Research	84.3%	−2.9%
Qwen-Agent	81.6%	−5.6%
Average	81.9%	−5.3%

Table 8: Cross-agent generalization. Pathfinder maintains strong performance on unseen agent frameworks, with only 5.3% average drop from in-distribution evaluation.

The model generalizes well across frameworks (average 5.3% drop), suggesting that SQL/bash query skills and deficiency patterns transfer. Performance is best on Open Deep Research (−2.9%), likely because its trace format is most similar to the training distribution; SWE-agent shows the largest drop (−7.4%), possibly due to its unique agent-computer interface producing distinctive trace patterns.

6.6 Self-Play Training Dynamics

Figure 9 tracks metrics across training:

Training Step	Injection Diversity	Detection Acc.	Query Depth
0 (init)	12.3 unique types	52.1%	3.2
2.5K	24.1 unique types	68.7%	4.8
5K	35.8 unique types	78.2%	5.4
7.5K	42.4 unique types	83.9%	5.9
10K (final)	47.8 unique types	87.2%	6.1

Table 9: Training dynamics over 10K steps. Injection diversity (unique deficiency types successfully injected) and detection accuracy both increase, while query depth grows as the model learns more sophisticated analysis strategies.

Curriculum Emergence. Early training concentrates on a few deficiency types (12.3 unique); by 10K steps, the model covers nearly the full taxonomy (47.8 of 50 types). Detection accuracy improves steadily from 52.1% to 87.2%. Query depth increases from 3.2 to 6.1, reflecting more sophisticated multi-step analysis—but remains efficient (the no-RL baseline averages 8.3 queries).

7 Discussion

7.1 Limitations

Computational Cost. PATHFINDER's code execution approach requires actual query execution against trace databases, which is slower than embedding-based retrieval. Each analysis averages 6.1 tool calls, with SQL queries taking 50-200ms depending on trace size. For real-time alerting, this latency may be prohibitive. However, for post-hoc analysis and debugging—the primary use case—this cost is acceptable. Future work could explore query caching and incremental analysis to reduce latency.

Gaming Potential. Self-play training optimizes injection and detection against a single model. An adversarial agent framework could, in principle, design trace formats that evade detection. We mitigate this through the diversity of our training distribution (four agent frameworks with distinct patterns) and grounding injections in real PR diffs rather than synthetic perturbations. Empirically, cross-agent generalization (5.3% average drop) suggests reasonable robustness, but targeted adversarial evaluation remains future work.

Taxonomy Completeness. Our 50-type taxonomy, while comprehensive, cannot cover all possible deficiencies. New failure modes emerge as agent architectures evolve—multi-modal agents, planning agents with world models, and agents with persistent memory introduce failure categories not yet represented. We designed the taxonomy to be extensible: new types can be added to the training distribution, and the self-play mechanism should learn to detect them without architectural changes.

Schema Dependency. While PATHFINDER is schema-agnostic in principle (dynamic discovery, no hardcoded columns), it still assumes traces can be loaded into SQLite tables with queryable structure. Unstructured logs (plain text, binary formats) require preprocessing. The bash/Python pipeline handles some of this flexibility, but extreme heterogeneity may degrade performance.

7.2 Future Work

Online Detection. The current system performs post-hoc analysis on completed traces. Extending to online, streaming detection would enable real-time intervention—halting agents before cascading failures occur. This requires incremental query execution and partial-trace reasoning, presenting interesting technical challenges.

Automated Repair. PATHFINDER identifies deficiencies but does not fix them. A natural extension is closing the loop: given a detected deficiency and its localization, generate a patch to the agent's prompts, tools, or configuration. This could leverage the same code-generation capabilities used for analysis, with the PR diff corpus providing supervision for repair quality.

Integration with Agent Development. We envision PATHFINDER as a development tool—integrated into CI/CD pipelines, flagging regressions before deployment. This requires tighter integration with agent framework internals and standardized trace formats. OpenTelemetry for agents [LangSmith, 2024] is a promising direction.

Multi-Modal Traces. Agents increasingly interact with visual inputs (screenshots, images, PDFs). Extending the analysis framework to handle multi-modal traces—correlating visual observations with textual actions—is an open challenge. Code execution may still be the right interface, but requires vision-language grounding.

8 Conclusion

We presented PATHFINDER, a self-improving agent trace analyzer that achieves 87.2% detection accuracy across 50 deficiency types. Our key contributions are:

1. **Code execution over retrieval**: SQL, bash, and Python queries enable precise, compositional analysis that embedding-based methods cannot express. This approach is schema-agnostic, scaling from single traces to cross-trace patterns across thousands of runs.
2. **A grounded deficiency taxonomy**: 50 failure types derived from real production bugs in major agent frameworks (CAMEL, SWE-agent, Open Deep Research, Qwen-Agent), covering parsing errors, context mismanagement, async race conditions, and architectural anti-patterns.
3. **Self-play reinforcement learning**: A single-model training regime where the model alternates between injecting deficiencies (grounded in real PR diffs) and

detecting them, generating unlimited training signal without human annotation. This approach, inspired by Self-Play SWE-RL [Wei et al., 2025], proves effective for the trace analysis domain.

Ablation studies confirm that both the code execution paradigm (−15.9% without SQL) and self-play training (−18.8% without RL) are essential to performance. Cross-agent generalization experiments demonstrate that learned analysis skills transfer across frameworks, with only 5.3% average performance drop on held-out agent types.

As AI agents become mission-critical in enterprise deployments, robust observability tools are essential. PATHFINDER represents a step toward automated, intelligent debugging—moving from passive logging to active analysis that understands agent behavior at a semantic level.

References

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations (ICLR)*, 2024.

Anonymous. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

Mert Cemri, Melissa Z. Pan, and Shuyi Yang. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

LangChain. LangSmith: LLM application observability platform. `https://smith.langchain.com`, 2024.

Arize AI. Phoenix: Open-source AI observability and evaluation. `https://phoenix.arize.com`, 2024.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Toward training superintelligent software agents through self-play SWE-RL. *arXiv preprint arXiv:2512.18552*, 2025.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, and others. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

LangChain. Open Deep Research: An open-source implementation of deep research. `https://github.com/langchain-ai/open_deep_research`, 2025.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and others. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Microsoft. PromptFlow: Build high-quality LLM apps from prototyping to production. `https://github.com/microsoft/promptflow`, 2024.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and others. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *International Conference on Machine Learning (ICML)*, 2023.

Thinking Machines. Tinker: A training API for researchers. `https://thinkingmachines.ai/tinker/`, 2025.

Moonshot AI. Kimi-K2-Thinking: A trillion-parameter mixture-of-experts reasoning model. `https://huggingface.co/moonshotai/Kimi-K2-Thinking`, 2025.

A Complete Deficiency Taxonomy

We provide the complete enumeration of all 50 deficiency types in our taxonomy, organized by category.

A.1 Production Failures (27 types)

These deficiencies are derived from actual bugs fixed in the PR histories of CAMEL, SWE-agent, Open Deep Research, and Qwen-Agent.

A.2 Conceptual Failures (23 types)

These deficiencies represent architectural and design-level issues identified from agent engineering principles.

ID	Deficiency Type	Source
Streaming & Response Handling		
P1	Token count computed from partial stream buffer	SWE-agent
P2	Tool calls split across stream chunks break parsing	CAMEL
P3	Incomplete response treated as complete on timeout	Open Deep Research
P4	Stream cancellation leaves state inconsistent	Qwen-Agent
Context & Token Management		
P5	Context limit exceeded without warning	SWE-agent
P6	Token counts stale after context summarization	CAMEL
P7	Truncation removes critical system instructions	Open Deep Research
P8	Token counting differs between client and API	Qwen-Agent
P9	History compression loses tool call results	SWE-agent
P10	Max tokens parameter ignored in streaming mode	CAMEL
Model API Issues		
P11	Temperature value out of model's accepted range	Qwen-Agent
P12	Tool_choice parameter unsupported by model	Open Deep Research
P13	Response format schema rejected by newer API version	SWE-agent
P14	Function calling format incompatible across providers	CAMEL
Tool Schema Conflicts		
P15	Union types in parameters break JSON schema validation	CAMEL
P16	$ref cycles in nested tool schemas cause infinite loop	SWE-agent
P17	Optional fields serialized as null rejected by tool	Open Deep Research
P18	Enum values case-sensitivity mismatch	Qwen-Agent
P19	Array items schema missing causes validation bypass	CAMEL
Async & Concurrency		
P20	Dictionary modified during iteration in async handler	Qwen-Agent
P21	Race condition in shared state between subagents	CAMEL
P22	Deadlock when subagent awaits parent response	Open Deep Research
P23	Callback registration lost on agent restart	SWE-agent
Parsing & Encoding		
P24	Non-UTF8 bytes in tool output crash JSON parser	SWE-agent
P25	Markdown code blocks confuse JSON extraction	CAMEL
P26	Template variable escaping breaks prompt rendering	Open Deep Research
P27	Unicode normalization differs between components	Qwen-Agent

Table 10: Complete list of production failure types (P1–P27) extracted from agent framework PRs.

ID	Deficiency Type
	Architecture & Control Flow
C1	Infinite loop: no termination condition on iterative refinement
C2	God agent: single agent handles all tasks without delegation
C3	Missing reflection: agent cannot self-correct after errors
C4	Improper decomposition: subtasks too large or too granular
C5	No fallback: single failure path with no recovery mechanism
C6	Circular delegation: agents delegate back to their delegators
	Prompt & Instruction Design
C7	Contradictory instructions in system prompt
C8	Vague success criteria: agent cannot determine task completion
C9	Missing few-shot examples for complex output formats
C10	Role confusion: agent identity unclear or inconsistent
C11	Over-constrained: excessive rules prevent valid solutions
	Tool Design & Integration
C12	Tool granularity mismatch: too coarse or too fine-grained
C13	Missing tool descriptions: agent cannot select appropriate tool
C14	Output format undocumented: agent misparses tool results
C15	No error specification: agent cannot handle tool failures
	Context & Memory Management
C16	Information loss: critical data dropped between steps
C17	Stale context: outdated information not refreshed
C18	Wrong retrieval: irrelevant context injected from memory
C19	No graceful degradation: memory overflow causes crash
	Coordination & Multi-Agent
C20	Missing synchronization: parallel agents see inconsistent state
C21	No conflict resolution: contradictory subagent outputs unhandled
	Evaluation & Termination
C22	Non-converging reflection: self-critique loops indefinitely
C23	Wrong stopping criterion: agent stops too early or too late

Table 11: Complete list of conceptual failure types (C1–C23) from agent engineering principles.