

AIRun: A Production Framework for Failure-Driven MCP Tool Evolution

Abhijeet Apsunde
eBay Inc.
USA

Paul Son
eBay Inc.
USA

Sai Charan Malipeddy
eBay Inc.
USA

Prasad Wali
eBay Inc.
USA

Abstract

Building tools for AI agents is fundamentally different from building human-facing APIs. AI agents make non-obvious, systematic mistakes that are invisible to traditional API observability: they omit critical query constraints, misinterpret ambiguous schema descriptions, and fail silently on timeout. We present a five-phase bootstrap methodology for developing AI-consumable tools by systematically observing agent failure patterns in production and iteratively evolving tool descriptions, schemas, and infrastructure in response. Applied to a production Model Context Protocol (MCP) server at eBay serving AI-driven incident investigation workloads, the methodology improved agent query success rate from 60% to 95%, reduced partition filter omission from 75% to 5%, and reduced investigation time from 20–30 minutes (experienced-practitioner benchmark) to under 3 minutes in documented case studies (N=3; see Section 5). A key contribution is the Closed-Loop Concurrent Agent Development Cycle: one agent session uses the tool in production while a parallel agent session reads structured failure logs and implements fixes—an AI-assisted feedback loop that delivered 15 pull requests in 4 weeks at 32 total human hours. We describe the failure taxonomy, tool description anti-pattern format, correlation ID observability schema, and three-stage async architecture that together constitute a principled engineering methodology for production AI tooling.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **Development frameworks and environments**; **Software development techniques**.

Keywords: AI agents, MCP, tool development, observability, LLM, failure analysis

1 Introduction

The emergence of Model Context Protocol (MCP) servers and LLM function-calling APIs has created a new class of software artifact: tools designed to be consumed by AI agents

rather than human developers. This inversion creates a fundamental challenge that existing API engineering practice does not address.

When a human developer misuses an API, the failure is visible and interpretable—a 400 error, a stack trace, a wrong result that the developer notices. When an AI agent misuses a tool, the failure is often silent or misattributed: the agent generates a syntactically correct SQL query that omits a critical partition filter, causing a full table scan that times out. The agent cannot distinguish between “my query was wrong” and “the infrastructure is slow.” The tool developer sees a timeout metric but cannot tell whether the AI constructed a bad query or the cluster was overloaded.

There is no established methodology for building AI-consumable tools by systematically observing agent failure patterns and evolving tool interfaces in response. Existing work addresses agent self-correction [1]—teaching the agent to recover from its own mistakes—but does not address the complementary tool-developer-side problem: how to build tools that agents are less likely to misuse in the first place.

This paper makes the following contributions:

1. **A five-phase bootstrap methodology** for developing AI-consumable tools through production failure pattern analysis, from minimal viable tool to production-ready via documented agent failures.
2. **A failure taxonomy** for AI agent tool invocations distinguishing infrastructure failures (timeouts, auth errors, connection exhaustion) from semantic failures (wrong query construction, schema misinterpretation, missing constraints).
3. **A correlation ID observability schema** that enables tool developers to debug AI agent failures in under 60 seconds, as demonstrated in production, by providing `correlationId`, `failureStage`, and `recommendations` in every error response.
4. **A tool description anti-pattern format** with explicit correct/incorrect usage examples that demonstrably reduces systematic agent mistakes.

5. **A three-stage async architecture** designed specifically for LLM tool call synchronous timeout constraints, enabling analytical workloads up to 5+ minutes within a 120-second MCP timeout.
6. **A Closed-Loop Concurrent Agent Development Cycle** in which a production usage agent session and a parallel development agent session are connected via structured markdown failure logs, creating an AI-assisted feedback loop for tool evolution.

We validate the methodology through **AIRun**, a production observability MCP server serving incident investigation workloads at eBay, with quantitative results across six months of production operation.

2 Background and Motivation

2.1 Model Context Protocol and LLM Tool Calling

The Model Context Protocol [13] defines a standard interface for exposing tools to LLM agents. An MCP server exposes named tools with descriptions, input schemas, and synchronous call semantics. The agent reads tool descriptions, selects tools, constructs arguments, and interprets responses—all through natural language understanding applied to structured schemas.

The synchronous call constraint is significant: MCP tool calls must complete within 120 seconds by default. For analytical workloads executed over distributed query engines such as Apache Spark, this constraint is frequently violated by legitimate queries.

2.2 The Tool-Developer Problem

Consider a concrete example from our deployment. AIRun exposes an `executeSparkSQL` tool that executes analyst-generated SQL against a distributed event store containing 174 million daily events partitioned by `dateAndHour`. The tool description initially read: *“Execute Spark SQL against the messaging event store.”*

In production, 75% of AI-generated queries omitted the `dateAndHour` partition filter. Each such query triggered a full table scan, consuming cluster resources and timing out after 120 seconds. The agent received a timeout error but could not determine whether the query was wrong or the cluster was slow. From the tool developer’s perspective, the timeout metric gave no signal about query quality.

This failure mode—systematic, invisible to standard observability, rooted in ambiguous tool description—is representative of a class of problems that traditional API engineering does not address.

2.3 Why Existing Approaches Are Insufficient

Agent self-correction approaches such as Reflexion [1] and AgentDebug [11] address agent-side recovery: the agent detects its failure and retries with a corrected approach. These do not prevent systematic mistakes from occurring in the first place and do not provide tool developers with actionable signals.

Standard API observability (latency histograms, error rate dashboards, distributed tracing [4]) provides no signal about whether an AI agent constructed a semantically correct request. A query that times out due to a missing partition filter is indistinguishable from a timeout due to cluster overload.

3 Methodology

The bootstrap methodology comprises five phases that iterate until a target agent success rate is achieved.

3.1 Phase 1: Bootstrap with Instrumented Minimal Viable Tool

Deploy the smallest possible tool that serves the target use case, with comprehensive failure instrumentation from day one.

MVP scope. A junior engineering team can implement the initial AIRun in one week. The MVP exposes: natural language query acceptance, table schema and metadata, SQL execution, result return.

Instrumentation requirements. Every tool invocation—success or failure—must log the fields shown in Listing 1.

Listing 1. Correlation ID observability schema

```
{
  "correlationId": "a7f3c9d1-...",
  "traceId": "x7d2f9ab1c...",
  "status": "ERROR",
  "failureStage": "KYUUBI_TIMEOUT",
  "errorMessage": "Query exceeded 120s timeout",
  "sql": "SELECT COUNT(*) FROM ...",
  "recommendations": [
    "Use executeQueryAsync for queries >60s",
    "Add dateAndHour partition filter"
  ]
}
```

`failureStage` takes one of four values: `VALIDATION` | `CONNECTION_POOL` | `KYUUBI_TIMEOUT` | `EXECUTION`. The `KYUUBI_TIMEOUT` prefix distinguishes query-level timeouts from MCP protocol-level timeout events.

This schema is designed for two consumers: the AI agent (which reads `recommendations` in the response) and the tool developer (who searches logs by `correlationId` to reconstruct what the agent did).

3.2 Phase 2: Production Failure Cataloging

Use an AI agent for real production tasks against the instrumented tool. Document every failure in structured session logs (Listing 2). In our deployment, over Jan–Mar 2026

the corpus comprised: 47 production incident investigations, 1,200+ AI agent queries across Spark SQL, OpenSearch, and application traces, 89 query failures documented with root cause analysis, and 12 major RCA documents produced by AI agents.

Listing 2. Extracted feature/bug request format (produced by Agent 1 from session log on operator instruction)

```
## Session: 2026-01-15
### Observation
AI takes 5-6 exploratory queries per session to
understand
schema and data distribution before writing target query.

### Root Cause
Tool exposes raw table schema with no context on data
patterns, cardinality, or access patterns.

### Failure Category
SEMANTIC -- schema misinterpretation

### Proposed Tool Change
Add source-code-derived annotations to tool descriptions
describing data patterns and typical query shapes.
```

Failure categories:

- **Ambiguous description** – tool description does not convey the constraint
- **Missing example** – agent cannot infer correct usage without examples
- **Wrong constraint** – description conveys a constraint incorrectly
- **Infrastructure** – auth expiry, connection exhaustion, timeout
- **Stale context** – agent’s accumulated session memory or skill file contains a negative capability claim (“tool X cannot do Y”) that has since become false; agent accepts it without re-verification and skips available options

3.3 Phase 3: Tool Description and Schema Evolution

For each cataloged failure, update tool metadata with anti-pattern examples embedded directly in the description (Listing 3). We term these *error prevention annotations*.

Listing 3. Anti-pattern format in tool description

```
CRITICAL: Every query MUST include dateAndHour partition
filter. Without it = full table scan on 174M+ daily
events = catastrophic timeout.

CORRECT:
WHERE dateAndHour = '2026-02-13--14'
WHERE dateAndHour >= '2026-02-13--08'
AND dateAndHour < '2026-02-13--11'

WRONG:
WHERE service = 'ORCH_SVC'
-- missing partition filter -- will timeout!
```

When schema ambiguity is the root cause, add disambiguation tools (e.g., a describeIndices tool that returns structured metadata about available data sources). Measure

improvement rate per failure pattern before and after each update cycle.

3.4 Phase 4: Async Architecture for Timeout Avoidance

The 120-second MCP synchronous timeout is a hard constraint for analytical workloads. We implement a three-stage async architecture:

Stage 1–Synchronous with error messaging. On timeout, return structured error with estimated completion time and explicit recommendation to use the async path.

Stage 2–In-memory async submit/poll. Client submits query and receives queryId. Client polls for status and retrieves result when complete. State held in server memory.

Stage 3–Distributed async. queryId tracked in external key-value store, enabling any pod to serve poll and result requests. Large results (>1 MB) stored in object storage. Handles pod restarts and load-balanced deployments transparently.

The three-stage progression is itself a product of the bootstrap loop: Stage 1 was deployed first; Stage 2 implemented after cataloging timeout failures; Stage 3 after cataloging cross-pod result loss.

3.5 Phase 5: Token Economy Optimization via Handshake Tool

As tool descriptions grow through Phases 3 and 4, token overhead increases substantially. The QueryTool description reached ~3,000–4,000 characters, adding ~1,200 tokens per call. **SessionInitializeTool (handshake)** delivers all operational knowledge once at session start; all other tools are reduced to ~75-character stubs.

A complementary technique is instructing the agent (via the handshake) to prefer summary SQL over row-level retrieval, and to use REST-calling helper scripts for mechanical multi-step extractions. A production attribution pipeline (322 templates, 3-step chain) used 32,100 tokens via REST vs. a 616,500-token MCP equivalent – 95% reduction. The practical rule: *mechanical extraction* (script) vs. *contextual reasoning* (MCP).

4 The Closed-Loop Concurrent Agent Development Cycle

A significant acceleration of the methodology comes from connecting the two agent sessions—production usage and tool development—via structured session logs (Figure 1).

Agent 1 (Usage agent). An engineer uses an AI agent to perform production tasks against the AIRun. All tool calls and responses are recorded as raw session logs. Session logs are not feature requests; they are a complete record that enables going back in time to reconstruct what happened.

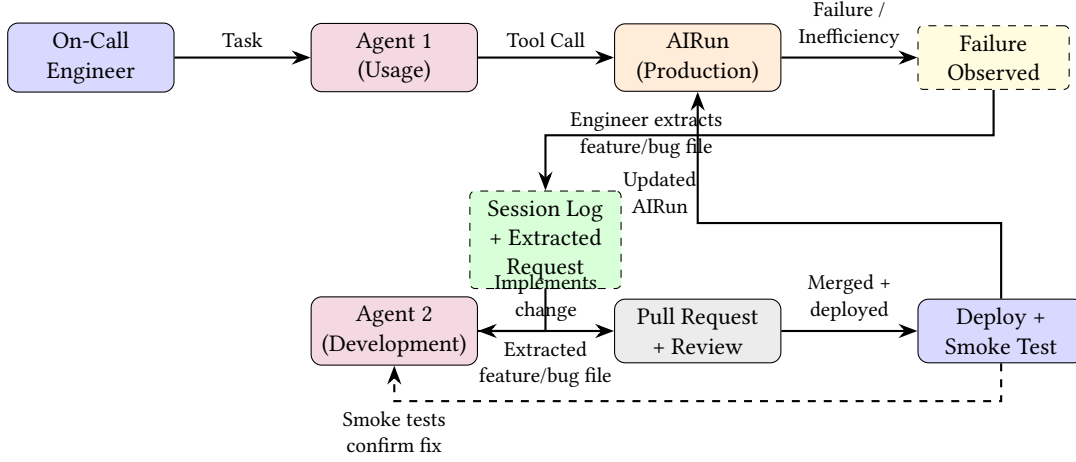


Figure 1. The Closed-Loop Concurrent Agent Development Cycle. Agent 1 uses the AIRun for production tasks; all activity is recorded as raw session logs. When the operator identifies an issue, Agent 1 extracts it into a dedicated feature or bug request file. Agent 2 takes that extracted file as input, implements the change, and creates a pull request reviewed and merged by a human. After deployment, smoke tests confirm the fix, closing the loop.

When the operator identifies an issue worth addressing, they instruct Agent 1 to extract it: the agent reads the relevant portion of the session log and produces a dedicated, structured feature or bug request as a separate .md file committed to the repository.

Agent 2 (Development agent). A separate AI agent session takes a specific extracted feature or bug request file as input – not the raw session log. It implements the described change (tool description update, new tool, architecture modification) and creates a pull request. The agent reads PR review comments via the GitHub CLI, addresses them, and pushes updates until the PR is approved and merged by a human reviewer.

Quality loop. After deployment, a platform health check confirms the target AIRun version is live. The development agent runs smoke tests against production and confirms that the failures documented in the session log no longer occur.

This loop accelerated tool evolution: 15 pull requests merged in 4 weeks, with total human effort of 32 hours across session log authoring, code review, and merge operations (Jan–Feb 2026).

5 Results

We deployed the methodology to a production deployment of AIRun serving AI-driven incident investigation workloads for eBay’s messaging platform. The server provides access to a distributed event store (174M+ daily events partitioned by hour) and an OpenSearch index for point lookups.¹

¹Template metadata access was provided via a separate, locally-deployed MCP server wrapping a MySQL database dump in a Docker container. This

5.1 Quantitative Results

Table 1 summarizes the before/after metrics across six months of production operation.

Query success defined. A query succeeds if it: (1) passes syntactic validation against the schema, (2) executes without runtime errors (no timeouts, no auth failures), (3) returns results when data exists or zero rows when none match, and (4) produces correct results (not timezone-shifted or otherwise semantically wrong). Queries that execute cleanly but omit critical filters (returning plausible but incomplete results) count as success in these metrics; such cases are addressed via tool description improvements tracked in the failure lineage (Table 2).

Baseline clarification. The “before” metrics represent the initial deployment (MVP, Phase 1) after basic Kyuubi and OpenSearch integration. This is not a comparison against a hypothetical hand-engineered baseline but rather a measurement of the methodology itself: how the system improves from a minimal starting point through iterative, failure-driven evolution. The methodology is the product.

5.2 Failure-to-Feature Lineage

Table 2 shows the complete traceability from documented AI agent failure to implemented tool change to measured improvement.

was an ad-hoc stopgap until official support for this data source became available. The MySQL MCP server is not part of AIRun’s tool catalog.

Table 1. Before/after metrics from production deployment

Metric	Before	After	Evidence
Query success rate	60%	95%	Query logs, Jan–May 2026
Partition filter omission	75%	5%	Query pattern analysis
Connection latency	1–2 s	2 ms	HikariCP pool metrics
Auth errors	10/day	0/day	Error logs
Investigation time [†]	20–30 min [*]	<3 min	3 documented case studies
Exploratory queries/session	5–6	1	Session log analysis

[†]Not log-measured; based on 3 documented case studies. ^{*}Experienced-practitioner benchmark, not a measured result.

Table 2. Failure-to-feature lineage: every server capability motivated by a documented production AI agent failure

Observation	Root Cause	Tool Change	Measured Impact
75% queries omit dateAndHour filter	Ambiguous “Execute Spark SQL” description	Anti-pattern examples + warning in description	Omission: 75% → 5%
5–6 exploratory queries per session	No context on data patterns or cardinality	Source-code annotations added to descriptions	Queries per session: 5–6 → 1
Connection latency 1–2 s per query	New Kyuubi connection per request	HikariCP connection pooling	Latency: 1–2 s → 2 ms
Long queries silently timeout	120 s MCP limit < 175 s query time	Three-stage async architecture	Enables 5-min+ queries
JWT 401 after 30-min sessions	Token baked into JDBC URL at startup	RefreshingJwtDataSource per connection	Auth errors: 10/day → 0
AI confused which index to query	Multiple indices, no disambiguation	describeIndices tool added	Eliminated index selection errors
Agent cannot debug its own failures	No failure context in responses	correlationId + failureStage in every response	Debug time: hours → <1 min

5.3 Development Timeline and Investment

The MVP was implemented by two junior engineers in one week. Over the following four weeks, 15 pull requests were merged (1,196 lines of documentation, 340 lines of tests, 152 lines of implementation), all implemented by an AI development agent reading structured session logs. Human effort was limited to session log authoring (~15 min/session), PR review, and merge. All changes were failure-driven.

Traditional incident investigation requires navigating multiple systems (dashboards, OpenSearch, Spark SQL, CAL traces) sequentially – typically 20–30 minutes for an experienced on-call engineer. AIRun reduces this to under 3 minutes in documented cases. Four engineers joined the on-call

rotation during the deployment window and conducted independent investigations without prior system training (qualitative observation; pre-AIRun ramp-up required weeks).

5.4 Case Study 1: Dispatch Traffic Attribution at Scale

Background. Executive stakeholders requested a dashboard showing email volume by business domain, requiring attribution of 778M dispatch events over 7 days to caller pool, template ID, and owning business unit. This class of analysis – correlating dispatch events across sources that share no common key and have no established data pipeline – had previously only been done for 1–2 templates at a time, during active incident investigations where the specific template was already known. At the scale of 235 templates across a 7-day window, the analysis was not considered feasible manually:

the correlation relies on loose matching and engineering judgement across multiple systems, with no programmatic join path between them.

Why no programmatic solution exists. Attribution requires joining application traces (unstructured JSON), Routing and Orchestration service event tables (different schemas, no shared key), and template metadata – each lookup requiring a separate query against a different system. The join logic involves judgement calls on ambiguous attribution that vary per template; no pipeline encodes these rules because the rules themselves are case-dependent. This is ad-hoc analytical work that falls outside what any standing data pipeline could serve.

AI-assisted approach (actual 8 hours). The AI agent automated: Spark SQL queries to extract typeId/contract pairs, batch application log lookups (100 trace IDs at a time), caller pool extraction from trace JSON, Routing↔Orchestration joins across the 7-day window, template metadata enrichment, and CSV updates. Human involvement was limited to defining requirements (1 hr), reviewing results (1 hr), and resolving conflicting attribution decisions (6 hrs).

Results. 207/235 typeIds attributed (88.1%); 715/728 templates attributed (98.2%); complete CSV mapping 778M events to caller pools, template IDs, and owning business units in 8 hours. The meaningful comparison is not time saved but *scale enabled*: this analysis had not been completed at full template scope before because the per-template manual effort made it uneconomical. AIRun made a previously unattempted analysis routine.

Methodology note. This case validated AIRun’s OpenSearch integration (fast dispatch lookups), template metadata access (via auxiliary MCP server), and async query architecture (multi-day Spark SQL aggregations). All three capabilities were integrated via the Closed-Loop Concurrent Agent Development Cycle after being requested in session logs from earlier investigations.

5.5 Case Study 2: Routing Service Retry Amplification RCA

Incident. A multi-hour site degradation affecting several buyer-facing and seller-facing platform services, causing significant latency increases across checkout and listing flows.

Root cause (AI-identified). A two-bug compound failure: (1) a template rendering error caused permanent render failures; (2) WebClient maxAttempts=3 in the dispatch client multiplied each failure exponentially. Stacked retries produced 3^N growth in theory; in practice, tens of millions of buffer events were observed within 6 hours across the affected dispatches.

AI-assisted investigation workflow. AIRun executed a Spark SQL query spanning 5 dateAndHour partitions (requiring 223 seconds of execution time, necessitating the async query architecture). Results identified the highest-volume dispatch with a ratio of over one million buffer events per retry increment versus the expected 1:1 ratio. The top 5 dispatches alone accounted for tens of millions of buffer events across multiple affected template IDs.

Fix progression tracked by AI. Three sequential fixes were correlated against event rate in real time: (1) config change adding error codes to the retry exclusion list (partial relief, did not stop client-level multiplication); (2) a commit reducing maxAttempts: 3→1 (changed exponential to linear growth); (3) a commit setting disableBuffer=true (full mitigation).

Output. Complete RCA document with incident timeline, code analysis, mathematical proof of retry amplification, fix progression, and permanent fix recommendations. The 223-second Spark SQL query that retrieved the smoking-gun data would have silently timed out under the original synchronous MCP architecture, demonstrating the direct dependency between the async query capability and the ability to complete this RCA.

5.6 Case Study 3: Silent Timeout Misread as Data Absence

Scenario. An engineer investigated a notification drop alert triggered for a high-volume email campaign. The agent queried the event store using synchronous executeQuery for a 4-hour window.

Agent behavior (before async). The synchronous query timed out at 300 s. The agent received an empty response with no timeout signal—no error code, no message, no failure stage. It concluded: “No drop events found for this campaign in the specified window.” This was wrong.

Root cause. The 300-second MCP synchronous timeout was shorter than the Spark execution time required for a 4-hour partition scan (2–5 min). The agent had no mechanism to distinguish timeout from genuine data absence.

Fix. Failure logged in a session document. The development agent implemented Phase 2 async architecture (executeQueryAsync, getQueryStatus, getQueryResult) in the same session.

Result. Async query returned thousands of drop events across the 4-hour window—data the synchronous path reported as nonexistent. Drops were attributed to template rendering errors caused by null upstream data from a dependent service, accounting for over 90% of total drops across multiple alert windows (from query results). Root cause identified in under 5 minutes.

5.7 Case Study 4: Observability Gap Blocks Debugging

Scenario. During a February 2026 investigation, an AI agent query failed. The agent logged: “MCP query timed out silently—how do I investigate? Where can I find application traces? What’s the traceId?”

Root cause. AIRun returned no correlation identifier on failure. The tool developer had no log to search. The investigation stalled.

Fix. Development agent implemented the correlation ID schema (PR #26, PR #24). Every invocation now returns:

Listing 4. Correlation ID in every response

```
{
  "correlationId": "a7f3c9d1-...",
  "traceId": "x7d2f9ab1c...",
  "failureStage": "KYUUBI_TIMEOUT",
  "recommendations": ["Use executeQueryAsync for queries >60s"]
}
```

Result. Debug time: hours → under 60 seconds via single application log search by correlationId.

5.8 Case Study 5: Stale Context Suppresses a Valid Investigation Path

Scenario. A June 2026 batch attribution session required 322 OpenSearch lookups. The project memory file stated: “OpenSearch is MCP-only pending the feature proposal in [...]”. The agent accepted this as authoritative.

Root cause. The memory entry was a *negative capability claim* written before the REST endpoint was deployed. It became false when the endpoint shipped, with no observable signal to the agent. Negative capability claims decay faster than positive ones because any deployment silently invalidates them.

Fix. Instead of accepting the stale record, the agent probed the endpoint: four HTTP calls, ~10 seconds. Three returned 404; one returned 400 (not 404)—a signal that the path exists but requires a POST body. A correctly-formed POST confirmed functionality. The full REST path was taken (642 calls across all 3 steps), saving an estimated 584,400 tokens versus the MCP path (Section 3.5).

Lesson. Negative capability claims in session memory should trigger re-verification, not acceptance. The probe cost is negligible; the cost of acting on a stale claim is not. A natural tool-level response is a capability-check tool that lets the agent verify what a server supports before committing to a query strategy.

5.9 Deployment and Adoption

AIRun is deployed as a Java/Spring Boot service on eBay’s internal platform-as-a-service (Kubernetes-based, multi-region). It serves a production messaging infrastructure processing

174M+ daily events across email, push notification, and message center channels.

Usage patterns. The core on-call team (8 engineers) uses the AIRun daily for: (1) incident response—investigating production alarms, running multi-step RCA workflows where the AI agent proposes hypotheses, queries real production data, validates or discards each theory, and produces a structured RCA document for human review; (2) incident timeline reconstruction—the agent correlates events across services using distributed trace IDs to establish causal chains; (3) production support—tracing ownership of communications when users report missing or incorrect content in notifications they received, requiring multi-hop attribution from user complaint to template owner to dispatch record to rendering service.

Expansion. Three sister teams (~20 engineers) began using AIRun following a team-internal presentation. One contributed a data source connector extending retention from 30 days to 3+ years. New on-call engineers conducted independent investigations without prior system training; this is a qualitative observation, not a measured result.

6 Distinguishing Properties

We identify five properties that distinguish this methodology from prior work. Of the closest related work surveyed, no single work combines all five.

(1) Tool-developer-side feedback target. US12423064B2 [14] addresses prompt A/B experimentation with production feedback—the feedback target is the agent’s *prompt*. Reflexion [1] targets agent *behavior*. This work targets the *tool interface*: descriptions, schemas, and infrastructure. The feedback target is entirely different, and the party responsible for acting on feedback is the tool developer, not the agent operator.

(2) Failure stage classification for AI agent debugging. Distributed tracing systems [4] provide correlation IDs for performance debugging but carry no semantics about AI agent behavior. Our failureStage enum (VALIDATION | CONNECTION_POOL | KYUUBI_TIMEOUT | EXECUTION) enables a tool developer to determine in under 60 seconds—via a single log search by correlationId—whether an agent failure was caused by a bad query, infrastructure, or timeout. No prior work we identified addresses AI-agent-specific tool invocation observability with this structure.

(3) Tool description as error prevention surface. Standard API documentation explains *what* a tool does. Error prevention annotations in tool descriptions explain *why* a constraint exists and what happens when it is violated. This is a different authoring objective that produces a different artifact. The ✓/× (correct/incorrect) example format embedded in tool metadata, as distinct from external documentation, is a novel application surface.

(4) Empirical bootstrap loop with traceable lineage.

Every server capability in our deployment is motivated by a documented production AI agent failure. This creates a traceable lineage from session log entry to tool change to measured improvement rate. Traditional API development is requirements-driven; this methodology is failure-driven, with the agent’s behavior in production as the source of truth for what the tool needs.

(5) Three-stage async architecture for MCP timeout constraints. The graduated progression from synchronous execution with error messaging through in-memory async to distributed cross-pod async is specifically designed for the 120-second LLM tool call timeout constraint. General async job queue literature does not address this constraint or the agent-readable error messaging requirements of Stage 1.

Table 3 summarizes how these five properties compare against the closest prior work. These five dimensions were identified as the novel contributions of this work; they are not a comprehensive scoring of prior work across all properties. Each prior work cited addresses many other aspects of agent systems not shown here.

6.1 System Architecture

Figure 2 shows the deployed system. User actions trigger a three-stage message pipeline (Routing → Orchestration → Delivery) before reaching the last-mile channel. All pipeline stages emit events to a centralized observability platform backed by Spark SQL (174M+ events/day, 30-day retention) and a search index (sub-second point lookups). AIRun provides unified AI agent access to this platform via the Model Context Protocol.

6.2 The Tool Description as Error Prevention Surface

A key insight is that tool descriptions are not documentation—they are error prevention surfaces. The purpose of a description is not to explain what the tool does; it is to prevent the agent from calling the tool incorrectly. This reframes the tool description as a first-class engineering artifact, not an afterthought.

Anti-pattern examples embedded in descriptions are more effective than constraint validation in schemas because they convey *why* a constraint exists (“without this filter = full table scan on 174M events”) rather than just *what* it is. Constraint validation fails silently from the agent’s perspective; descriptive warnings are visible at description-read time.

6.3 Generalizability

The methodology is agnostic to tool domain and query language. LLM provider and agent framework portability is an architectural property of the design (standard MCP interfaces, plain markdown skill files) but has not been empirically

evaluated across providers – all sessions described here used a single model and agent harness (see Section 6.5). The core principle—instrument for agent failure visibility, catalog failures by root cause, evolve tool descriptions with anti-pattern examples—applies to any data source or API placed behind an MCP interface. Three successive iterations within our own deployment demonstrate this concretely.

Three successive iterations demonstrate this. **Iteration 1** (Spark SQL over Kyuubi) drove the partition filter warnings, connection pooling, async architecture, and JWT refresh described in this paper. **Iteration 2** (OpenSearch integration) surfaced a distinct failure class – eBay-specific auth – and resolved it through the same loop using Iteration 1’s failure catalog as a template. **Iteration 3** (historical data source, contributed by a sister team without central coordination) extended retention from 30 days to 3+ years, enabling trend analysis across annual email seasons.

These three iterations span REST APIs, distributed SQL engines, search indices, and data warehouses. The failure taxonomy, anti-pattern annotation format, and failure-to-feature loop are reusable regardless of underlying protocol.

6.4 Limits of Description-Driven Error Prevention

The methodology is calibrated for *systematic* failures that recur across sessions. Clear limits: (1) **One-time failures** do not justify the session-log documentation investment. (2) **Reasoning failures** stem from model limitations, not tool descriptions; they require model improvement or query decomposition at the agent workflow level. (3) **Handshake token cost** dominates for very short sessions (1–2 queries); the optimal approach is embedded descriptions for rarely-called tools and the handshake for frequently-called suites. (4) **Description drift** occurs as schema evolves; stale descriptions actively mislead the agent and the bootstrap loop must include a description review step on underlying tool changes.

6.5 Domain Knowledge vs. Tool Interface Knowledge

A key architectural insight from production deployment is that two distinct knowledge layers are required for effective AI-assisted investigation, and they must be managed separately.

Tool interface knowledge is what AIRun’s handshake tool delivers: how to call each tool, what parameters are required, what constraints apply, what errors mean. This knowledge lives at the tool boundary and is owned by the tool developer.

Domain workflow knowledge is orthogonal: given that tools work correctly, how should an investigation proceed? Three recurring patterns illustrate this: (1) *delivery trace* –

Table 3. Comparison against closest prior art across the five properties introduced by this work. ✓ = addressed, ~ = partially addressed, × = not addressed. These dimensions were selected as the novel contributions of this work; prior work addresses many properties not shown here.

Work	(1) Tool-dev side feedback	(2) Failure stage classification	(3) Error prevention annotations	(4) Bootstrap lineage	(5) Async for MCP
Reflexion [1]	×	×	×	×	×
AgentDebug [11]	×	~	×	×	×
ToolLLM [2]	×	×	×	×	×
Srinivasan [12]	×	~	×	×	×
Auld et al. [14]	~	×	×	×	×
Dapper [4]	×	~	×	×	×
This work	✓	✓	✓	✓	✓

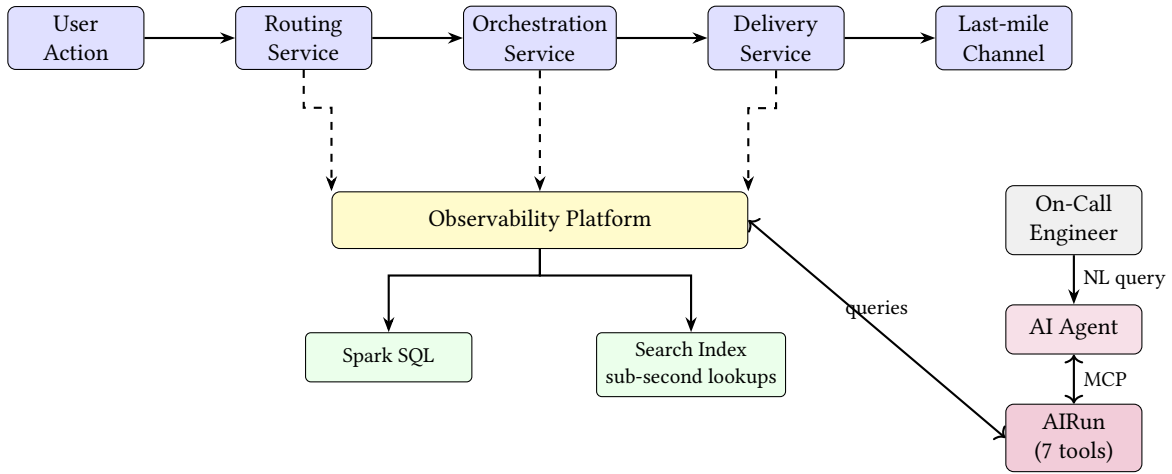


Figure 2. Deployed system architecture. User actions trigger a three-stage message pipeline whose services all emit events to a centralized observability platform. AIRun exposes the platform to an AI agent via 7 MCP tools, enabling on-call engineers to investigate incidents using natural language.

query search index by message ID, join pipeline event tables on the dispatch ID, check for BUFFERED events, correlate timing with downstream health; (2) *system health sweep* – aggregate event counts by status code across current and prior hour partitions, cross-reference anomalous codes for samples; (3) *alert-driven investigation* – isolate affected dispatch IDs, retrieve application traces for a sample, group error messages to identify the dominant failure mode. Sequencing, join strategy, and sampling are domain knowledge no tool description can carry.

In our deployment, this knowledge lives in a versioned repository of structured markdown skill files. The skill repository evolves by the same failure-driven loop as tool descriptions: when agents investigate in the wrong order or miss a critical join, those failures are documented and the affected skill file is updated.

This two-layer architecture – tool interface knowledge in the MCP server, domain workflow knowledge in a versioned skill repository – generalizes to any multi-service observability domain. Unlike fine-tuning, the artifacts (tool descriptions, skill files, the correlation ID schema) are plain text, version-controlled, and carry no model dependency – any domain engineer can update them without ML expertise.

6.6 Limitations

Measurement snapshot and current ceiling. The initial 85% success rate (Jan–Mar 2026) improved to 95% following two additional fixes identified and implemented via the same bootstrap methodology: (1) SQL comment stripping—queries prefixed with `-` or `/* */` comments were incorrectly rejected by validation logic that checked the raw SQL before stripping comments; (2) null rowCount handling—an NPE in the async

result path when Kyuubi returns a null row count for large or long-running result sets.

The remaining 5% failure rate falls outside the reach of the bootstrap methodology: these failures occur at the same rate when SQL is executed by other means and are not attributable to tool description quality. Two categories account for this floor: (1) Spark SQL resource exhaustion—queries that fail late into execution after consuming cluster resources; (2) Kyuubi connection lifecycle failures—queries that fail when the JVM holding the JDBC connection is recycled mid-execution due to pod restarts or rolling deployments. A fix exists (submitting via a dedicated service account that manages the full connection lifecycle outside the application pool) but is pending implementation; the current workaround is resubmission. Both failure classes land in the EXECUTION failure stage and are distinguishable from agent-caused failures via the correlation ID schema.

Session log quality dependency. The Closed-Loop Concurrent Agent Development Cycle is only as effective as the session logs it consumes. Vague logs produce vague tool changes. A structured log format (observation, root cause, proposed change) is necessary for reliable AI-implemented fixes.

Evaluation scope. Case studies represent investigations where agent output was verified by a human and found correct—not a random sample. We lack a false-conclusion rate because AIRun was adopted organically into an existing workflow; sessions where the agent was corrected were not systematically logged at the outset. Pre-AIRun Jira time data exists but covers a different operational regime (fewer services, different failure modes) and would introduce confounds as a direct baseline. The 20–30 minute pre-AIRun estimate is an experienced-practitioner benchmark consistent with the workflow steps described in Section 4, not a measured result.

Measurement validity. Query success rate is measured from server-side logs, not investigation outcome quality. A query that succeeds but returns wrong results is counted as a success in our metrics.

Description-driven prevention vs. server-side enforcement. Server-side safeguards (partition filter enforcement, query rewriting) would provide stronger guarantees: a query missing the dateAndHour filter could be rejected or auto-completed. Tool descriptions are probabilistic—an agent that misreads or ignores the description will still submit an invalid query. The current architecture uses descriptions as the primary prevention layer because they generalize across failure modes without code changes: adding a new anti-pattern requires updating a string, not a validator. Adding server-side enforcement for high-cost failure modes is the natural next step as agent reliability in production matures.

7 Related Work

Agent self-correction. Reflexion [1] and similar approaches teach agents to recover from failures through self-reflection. AgentDebug [11] provides a failure taxonomy for agent self-correction. Our methodology is the inverse: the tool developer improves the tool based on observed agent behavior, rather than the agent improving its behavior. The feedback target is different.

LLM tool use. ToolLLM [2] and related work studies how LLMs select and invoke tools. This work characterizes agent tool use behavior but does not address how tool developers should evolve tools in response to observed failures.

MCP production patterns. Srinivasan [12] addresses structured error semantics for agent self-correction in MCP deployments. It does not cover developer-side failure cataloging, tool description evolution, or the bootstrap methodology.

Tool description optimization. Hasan et al. [15] empirically analyze 856 tools across 103 MCP servers and find that 97.1% contain at least one description quality issue, with over half failing to clearly state their purpose. Their work confirms at scale the problem we address: tool descriptions are systematically deficient. They propose static augmentation heuristics; our methodology derives description improvements from observed production failures—grounding each change in a specific agent failure rather than a general quality rubric. This is a complementary approach: static analysis surfaces potential issues at design time; our loop surfaces actual failures in production. Neither Meta-Harness (Lee et al., COLM 2026) nor any other cited prior work targets the same feedback target: live production execution traces driving tool-developer-side interface changes. Meta-Harness optimizes agent scaffolding code (system prompts, retrieval logic, context management) via benchmark-driven search; it does not touch tool interface definitions, schemas, or server infrastructure. Pan [16] proposes an Agent-First Tool API paradigm with a six-verb semantic protocol and normalized tool contract for enterprise AI agents, achieving 88% task success versus 64% for CRUD baselines. This work is specification-driven; our methodology is empirically driven—tool interface changes are motivated by documented production failures rather than upfront design principles.

Prompt engineering. Chain-of-thought [3] and related techniques improve agent reasoning. Tool descriptions are a distinct surface from agent prompts, owned by tool developers rather than agent operators, and evolve independently.

API observability. Dapper [4] and distributed tracing systems provide correlation IDs for performance debugging. Our schema extends this pattern specifically for AI agent debugging, adding failureStage classification and agent-readable recommendations.

LLM observability. AgentOps [5] proposes a lifecycle-wide observability taxonomy for LLM agents. Moshkovich et al. [6] address observability challenges in non-deterministic agentic systems. Our work is complementary but distinct: rather than observing agent behavior for analysis, we use observed failures to drive tool interface evolution on the developer side.

Tool invocation reliability. Huang et al. [7] propose a 12-category error taxonomy for tool invocation failures in multi-agent systems. Our failure taxonomy (Section 5) is independently derived from production failures and focuses specifically on the developer-actionable root causes—ambiguous descriptions, missing examples, infrastructure constraints—rather than agent-side error classification.

AI Ops. Saha and Hoi [8] apply NLP to mine root cause knowledge from cloud service incident reports. This work targets post-hoc knowledge extraction from text; AIRun provides real-time data access during live investigations. SuperBench [9] validates cloud AI infrastructure proactively. Our work is reactive but complementary: AIRun accelerates the human-in-the-loop investigation that follows anomaly detection.

Natural language database interfaces. Hong et al. [10] survey LLM-based text-to-SQL systems. These systems translate NL queries to SQL but assume a clean, well-documented schema. AIRun’s tool description evolution addresses the harder problem of schemas with 50+ fields, undocumented constraints, and complex partition requirements that cause systematic AI agent failures in production.

API usability and failure-driven evolution. Murphy-Hill et al. [17] describe StopMotion, a technique that discovers API usability problems at Google scale by analyzing successive file-level changes made by developers—using observed developer behavior to surface misuse patterns. This is the closest non-AI precedent for our approach: using observed usage failures to drive interface improvement. Our work applies the same principle to AI agents rather than human developers, with the distinction that agent failures are systematic and silent where human failures are visible and idiosyncratic. Peldszus et al. [18] identify eight factors influencing REST API usability through interviews with 16 industry experts, noting that guidelines must evolve with organizational practice. Our failure taxonomy and anti-pattern format operationalize this insight for AI agent consumers specifically.

8 Lessons Learned

Six months of production operation surface the following generalizable principles about building and operating AI-consumable tooling:

Production failures are a better specification than requirements. Every capability in AIRun was motivated by a documented production failure. The partition filter warning exists because 75% of real queries timed out. The async architecture exists because a live RCA produced a 223-second query. A requirements-driven approach would not have discovered these constraints; only production usage did. This suggests a general principle: for observability tooling whose usage patterns are unknowable upfront, empirical bootstrapping outperforms specification-driven design.

Tool descriptions are load-bearing artifacts. Reducing partition filter omission from 75% to 5% required no code change—only a rewrite of a tool description. The description is not documentation; it is the primary mechanism for communicating operational constraints to an AI agent. It should be treated with the same engineering rigor as the API contract itself.

Silent failures are worse than loud ones. Three of the five case studies involved silent failures: a timeout with no signal, an empty response misread as absent data, and a query failure with no correlation ID. In each case, the investigation stalled not because the answer was unavailable but because the failure mode was invisible. Observability of the tool itself—not just the system it queries—is a first-class requirement for AI-consumable tooling.

AI agents normalize across experience levels—in both directions. Junior engineers tended to accept agent conclusions at face value; senior engineers occasionally overrode correct conclusions based on tribal knowledge. One incident: the agent correctly identified an “at-least-once” delivery assumption being violated, but the system’s original designer dismissed the finding. The mitigation was a prompt constraint in a versioned skill file requiring the agent to ground every conclusion in production data, state unexplained-but-plausible scenarios as hypotheses, and defer to the operator on ambiguous cases. This normalized both failure modes by making the evidence chain explicit.

Session logs are write-ahead logs for human-AI collaboration. A structured session log enables resumption when an agentic session ends (context limits, SSO timeouts, operator interruption) without repeating prior work. Logs also serve as raw material for extracting reusable skills: patterns that recur become .md skill files in a versioned repository, enabling consistent repetition of complex workflows despite LLM non-determinism.

The development loop improves with its tooling. eBay’s internal AI PR review tooling was integrated with the development agent via the gh CLI. When a higher-quality internal review tool became available mid-project, the same integration allowed the development agent to fetch and address the new comments autonomously—no loop changes

required. The Closed-Loop Concurrent Agent Development Cycle is not static: the quality of each component (review tooling, session log structure, smoke test coverage) directly affects throughput.

9 Future Work

Proactive detection. AIRun is reactive; the next phase is an AI agent that continuously monitors metrics and generates draft RCA documents before engineers are paged. The required MCP tools are already deployed.

Automated remediation. The retry amplification case study shows the potential: the agent identified the exact config change (`maxAttempts: 3`→`1`) and the commit that introduced it. Future work closes this loop by letting the agent draft the fix for human approval.

Failure taxonomy at scale. The current taxonomy was derived from 89 failures at one site. A multi-organization corpus would enable quantified prevalence rates to guide tool developer prioritization.

10 Conclusion

We have presented a five-phase bootstrap methodology for developing AI-consumable tools through production agent failure pattern analysis. The methodology is grounded in a key insight: tool descriptions are error prevention surfaces, and they should be evolved based on observed agent failure patterns rather than written to spec and left static.

Applied to a production deployment of AIRun at eBay, the methodology improved agent query success rate from 60% to 95% and reduced investigation time from 20–30 minutes to under 3 minutes in documented case studies. The Closed-Loop Concurrent Agent Development Cycle—connecting a production usage agent to a development agent via structured failure logs—accelerated tool evolution compared to traditional development cycles.

As AI agents become standard consumers of platform APIs, the tool-developer-side methodology described here fills a gap in existing engineering practice. The failure taxonomy, anti-pattern description format, correlation ID schema, and async architecture are reusable patterns for teams building production AI tooling and LLM function-calling APIs.

AI Usage Disclosure. Claude (Anthropic) was utilized to generate sections of this Work, including text, tables, graphs, code, data, citations, etc.

Acknowledgments

The authors thank their management for providing the team space to experiment with this work before its value was apparent, and the observability platform team for building

the foundation on which AIRun is built. Thanks to the on-call engineers whose production investigations generated the failure corpus that drove this methodology. Claude was utilized to generate sections of this Work, including text, tables, graphs, code, data, citations, etc.

References

- [1] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [2] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024.
- [3] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, 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.
- [4] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report, Google, 2010.
- [5] Liming Dong, Qinghua Lu, and Liming Zhu. AgentOps: Enabling observability of LLM agents. *arXiv preprint arXiv:2411.05285*, 2024.
- [6] Dany Moshkovich, Hadar Mulian, Sergey Zeltyn, et al. Beyond black-box benchmarking: Observability, analytics, and optimization of agentic systems. *arXiv preprint arXiv:2503.06745*, 2025.
- [7] Donghao Huang, Gauri Malwe, and Zhaoxia Wang. When agents fail to act: A diagnostic framework for tool invocation reliability in multi-agent LLM systems. *arXiv preprint arXiv:2601.16280*, 2026.
- [8] Amrita Saha and Steven C.H. Hoi. Mining root cause knowledge from cloud service incident investigations for AIOps. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.
- [9] Yifan Xiong, Yuting Jiang, Ziyue Yang, et al. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2024.
- [10] Zijin Hong, Zheng Yuan, Qinggang Zhang, et al. Next-generation database interfaces: A survey of LLM-based text-to-SQL. *arXiv preprint arXiv:2406.08426*, 2024.
- [11] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.
- [12] Vasundra Srinivasan. Bridging protocol and production: Design patterns for deploying AI agents with model context protocol. *arXiv preprint arXiv:2603.13417*, 2026.
- [13] Anthropic. Model Context Protocol (MCP) Specification, version 2025-11-25. <https://modelcontextprotocol.io>, 2025.
- [14] David H. Auld, Kanio G. Dimitrov, Jafar M. Al-Kofahi, et al. Optimizing behavior and deployment of large language models. U.S. Patent 12,423,064 B2, September 2025. Assignee: Microsoft Technology Licensing, LLC.
- [15] Mohammed Mehedi Hasan, Hao Li, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. Model Context Protocol (MCP) Tool Descriptions Are Smelly! Towards Improving AI Agent Efficiency with Augmented MCP Tool Descriptions. *arXiv preprint arXiv:2602.14878*, 2026.

- [16] Kai Pan. Agent-first tool API: A semantic interface paradigm for enterprise AI agent systems. *arXiv preprint arXiv:2605.10555*, 2026.
- [17] Emerson Murphy-Hill, Caitlin Sadowski, Andrew Head, John Daughtry, Andrew Macvean, Ciera Jaspan, and Collin Winter. Discovering API usability problems at scale. In *Proceedings of the 2nd International Workshop on API Usage and Evolution (WAPI)*, pages 14–17, 2018.
- [18] Sven Peldszus, Jan Rutenkolk, Marcel Heide, Jan Sollmann, Benjamin Klatt, Frank Köhne, and Thorsten Berger. Developer perspectives on REST API usability: A study of REST API guidelines. *arXiv preprint arXiv:2601.16705*, 2026.