TraceAggregator: A Distributed Observability Platform for Multi-Agent LLM Pipelines

Aishwarya Madhave, Nihad Karathodath Parambil
Master's in Software Engineering
San Jose State University
San Jose, United States
aishwarya.madhave@sjsu.edu
nihad.karathodathparambil@sjsu.edu

Saransh Soni, Vandan Shah
Master's in Software Engineering
San Jose State University
San Jose, United States
saransh.soni@sjsu.edu
vandansanket.shah@sjsu.edu

***Abstract*—Multi-agent LLM pipelines fail in ways traditional tracing tools cannot surface. Jaeger will tell you a request took 4 seconds and the `reviewer_agent` errored. It w ill not tell you that the researcher caused the latency, that the coder's hallucination spread downstream, or that the reviewer picked one candidate over two others with 0.3 1 confidence after rejecting them. The causal order, token cost, and reasoning all stay hidden. This paper describes TraceAggregator, the observability platform we built for these systems. A lightweight SDK captures spans across agent boundaries. A reconstruction engine uses vector clocks to rebuild the causal DAG even when spans arrive out of order. The Blame View ranks agents by their share of latency, tokens, and errors, with bootstrap confidence intervals attached so that a stable ranking is distinguishable fr om a noisy one. Decision events are stored as structured records with chosen candidate, rejected candidates, and rationale, rather than as free-form log lines. Durability si ts on a write-ahead log. Per-tenant isolation is enforced at both ingest and query layers. Six SLOs are evaluated on a fixed cadence and feed into an incident lifecycle engi ne. The React dashboard shows traces, DAGs, blame rankings, SLO status, and incidents. On a four-agent LangGraph pipeline, trace reconstruction stayed under the 60-second p9 5 target and blame attribution held up under deliberate fault injection.**

***Index Terms*—Distributed Tracing, Multi-Agent Systems, Large Language Models, Observability, Vector Clocks, Causal DAG Reconstruction, Blame Analysis, Decision Observability, LangGraph, ClickHouse, gRPC, FastAPI, Service-Level Objectives, Incident Management, Metadata Governance, Write-Ahead Log**

I. Introduction

In a multi-agent LLM system, an orchestrator dispatches work to specialized sub-agents (researchers, coders, reviewers, and so on). A single user request can fan out across several agents running concurrently. Every agent calls an LLM. Every call can fail, time out, or burn unexpected tokens. Results merge back through fan-in nodes before a res ponse is returned. Debugging any of this means reasoning about causal order across asynchronous boundaries, attributing cost and latency to specific agents, and figuring out *why* an agent made a particular choice rather than just *what* it returned. Standard microservice observability does not give you any of that.

The widely used tracing tools (Jaeger, Zipkin, OpenTelemetry) were designed for synchronous request/response service meshes. None of them model vector-clock causality across concurrent agent branches. None treat AI decision events as records you can query. None offer blame primitives that understand token economics or LLM-specific failures like hallucination, timeout, or confidence collapse. The data model itself is wrong for the problem.

TraceAggregator addresses this gap with four core contributions:

1) A framework-agnostic SDK with a single decorator (`@instrument_node`) that injects trace context, propagates vector clocks through LangGraph state, and shi ps spans and decisions over a non-blocking gRPC channel.
2) A causal DAG reconstruction engine that handles out-of-order arrival. When explicit `parent_span_id` links are missing or do not resolve, the engine falls back to vector-clock precedence, and it handles fan-in cleanly when multiple legitimate parents exist.
3) A two-tier blame system. V1 produces a weighted point estimate across latency, tokens, and errors. V2 layers on bootstrap confidence intervals and an error-amplifi cation factor based on how far downstream an error's effects can be observed in the DAG.
4) An operational control plane with six SLOs, a four-rule alerting worker, a state-machine incident engine, and the React dashboard.

The remainder of this paper is organized as follows. Section II describes the system architecture. Section III details platform functionalities. Section IV discusses stakeholder use cases. Section V enumerates technologies used. Section VI covers the testing strategy and results. Section VII addresses security and governance. Section VIII concludes with a discussion of limitations and future work.

II. System Architecture

7 TraceAggregator is structured as five layers: instrumentation, ingestion, storage, analysis, and presentation. Fig. 1 shows the data flow end-to-end.

```
LangGraph Agents (demo/pipeline.py)
    | @instrument_node -> sdk/core.py
    | emit_decision -> sdk/core.py
    v
Collector (gRPC :50051)
    | collector/server.py (async)
    | collector/writer.py (WAL + batch)
    v
ClickHouse
    raw_spans | raw_decisions
    v
Engine Worker (engine/worker.py)
    | dag.py -> causal DAG
    | blame.py / blame_v2.py
    v
ClickHouse
    reconstructed_traces | decision_edges
    v
FastAPI (:8000)
    v
React UI (:5173)
```

Fig. 1. End-to-end data flow in TraceAggregator.

A. Instrumentation SDK

The SDK (`sdk/core.py`, `sdk/instrument.py`) is framework-agnostic but ships with first-class LangGraph support. The `@instrument_node` decorator wraps any LangGraph node functi on: it reads four context keys from the incoming state (`_trace_id`, `_vector_clock`, `_parent_span_id`, `_tenant_id`), calls `begin_span()` to create a n ew `SpanContext` and increment the agent's vector clock component, executes the node, captures token counts and error classification, then calls `build_span()` to assemble the protobuf message and `emit_span()` to dispatch it on a background thread. The outgoing state is updated with the new context so downstream nodes inherit the causal lineage.

Agent latency is deliberately decoupled from collector availability: all gRPC emission runs on a `ThreadPoolExecutor` background thread. If the collector is unreachable, spans are silently dro pped after a configurable number of retries rather than blocking the agent.

B. gRPC Collector

The collector (`collector/server.py`) is an asyncio-based gRPC service exposing four RPCs: `RecordSpan`, `StreamSpans`, `RecordDecision`, and `StreamDecisions`. The streaming variants support high-throughput ingestion from pipeline replays or load generators. Tenant resolution from request metadata is enforced at the collector boundary; spans lacking a valid te nant header are rejected with `UNAUTHENTICATED`.

C. Durable Batch Writer

The batch writer (`collector/writer.py`) implements a two-phase durability guarantee. Every row is written to the write-ahead log (WAL), a per-row JSON file under `./wal/{span,decision }/`, before an ACK is returned to the client. Rows are concurrently enqueued into an in-memory buffer (capacity 50,000) and flushed to ClickHouse in configurable batches (default 500 rows, 1-second interval). On startup, the writer scans the WAL for unprocessed rows and replays them, ensuring exactly-once semantics via per-row idempotency keys.

D. ClickHouse Storage Layer

Raw spans and decisions are stored in two ClickHouse tables (`tracing.raw_spans`, `tracing.raw_decisions`) using `MergeTree` with configurable TTLs (default 30 days for raw spa ns). Reconstructed traces and decision edges occupy a second pair of tables using `ReplacingMergeTree`, which provides idempotent engine reruns by retaining only the newest reconstruction row per `trace_id`.

E. Engine Worker

The engine worker (`engine/worker.py`) polls `raw_spans` for traces updated within the last 300 seconds. It processes traces in parallel using a `ThreadPoolExecutor` (configurab le via `ENGINE_RECON_MAX_WORKERS`). For each trace, it reconstructs the causal DAG, computes both V1 and V2 blame scores, and writes results to `reconstructed_traces`. A separate pa ss links decision events to spans via descendant traversal, populating `decision_edges`.

F. FastAPI Query Service

The API layer (`api/main.py`) exposes a RESTful interface with cursor-based pagination (preferred over offset for deep ClickHouse scans), multi-dimensional filters (agent, event type, token bo unds, latency bounds, metadata key/value), and a Server-Sent Events (SSE) stream for the live-updating trace list UI. Tenant isolation is re-enforced at the API boundary by injecting a `WHERE tenant_id = ?` clause on all queries.

G. React Dashboard

The UI (`ui/src/`) is a Vite + React + Tailwind application with five pages: a live trace list backed by SSE, a trace detail page with timeline waterfall, interactive DAG tree, Decision Chain panel, and Blame panel; a cross-trace agent blame leaderboard with 1h/6h/24h/7d windows; an SLO dashboard; and an incidents page with acknowledge/resolve actions. Vite proxies `/api/*` to tttlocalhost:8000 during development, which eliminates CORS configuration.

III. Platform Functionalities

A. Causal DAG Reconstruction

The DAG reconstruction algorithm (`engine/dag.py`) uses the Lamport-style vector clock ordering relation: clock A causally precedes clock B iff $A[k] \leq B[k]$ for all agents k and $A\ neqB$. The algorithm runs in two phases:

Phase 1 – Explicit edges. If a span carries a `parent_span_id` that resolves to another span in the batch, that edge is trusted directly.

Phase 2 – Inferred edges. For spans without a resolvable explicit parent, the algorithm computes the *causal frontier*: the maximal set of spans that strictly precede the current spa n and are not themselves dominated by any other predecessor in that set. This frontier represents the immediate causal dependencies. If a single frontier candidate exists, it becomes the inferred par ent. If multiple exist (fan-in), all are recorded and the edge type is marked `explicit_plus_fanin` or `inferred`. Inferred edges are flagged in the UI to signal potential missing spa ns.

This approach handles the common pattern of an orchestrator dispatching to parallel research and coder agents, both of which are legitimate causal parents of a downstream reviewer node.

B. Blame Attribution

1) V1: Weighted Point Estimate: The V1 model (`engine/blame.py`) aggregates per-agent totals for latency, tokens, and errors, then computes share fractions across the trace. The composite blame score is:

$$B_i = 100 \cdot \left(0.5 \cdot s_i^{lat} + 0.3 \cdot s_i^{tok} + 0.2 \cdot s_i^{err}\right) \qquad (1)$$

where s_i^{lat}, s_i^{tok}, and s_i^{err} are the agent's fractional shares of total trace latency, tokens, and error count respectively. Agents are ranked by B_i descending.

2) V2: Bootstrap CI and Error Amplification: The V2 model (`engine/blame_v2.py`) adds two enhancements. First, it bootstraps the span population 1,000 times, computing B_i on each resample, then extracts the 2.5th and 97.5th percentil es as a 95% confidence interval. This shows whether a blame ranking is statistically robust or an artifact of small sample size.

Second, errors incurred by agents with many DAG descendants are amplified relative to leaf errors, using a multiplier:

$$\alpha_i = \min\left(1 + \frac{d_i}{D_{\max}}, 2.0\right) \qquad (2)$$

where d_i is the number of descendants of agent i's error spans, computed via BFS while avoiding double-counting in multi-parent DAGs, and $D_{\max}$ is the maximum descendant count in the trace. The amplifier is capped at $2.0\times$ to prevent a single root error from completely dominating the score.

C. Decision Observability

Decision events (`emit_decision_event` in `sdk/core.py`) are first-class records separate from spans. Each decision captures: the selected candidate ID, a confidence score $\in [0, 1]$, a structured rationale summary, and a list of all evaluated candidates with scores. The SDK enforces quality constraints on rationale text (LLM-05 rule): the rationale must begin with an action ver b from a curated list, be between 10 and 512 characters, and contain no filler phrases. Violations are stored in span metadata for operator review.

Decisions are linked to their associated spans by the engine worker via `decision_edges`, enabling the Decision Chain view in the UI and the `/traces/{id}/root-cause` endpoint that r anks decisions by their downstream impact.

D. SLO Evaluation

Six quantitative SLOs are defined in `slo/spec.py` and evaluated periodically by `slo/worker.py`.

TABLE I
Defined Service-Level Objectives

SLO Name	Target	Comparison	Window
ingest acceptance	0.999	$\geq$	60 min
flush success	0.990	$\geq$	60 min
recon lag p95	60 s	$\leq$	60 min
recon lag p99	120 s	$\leq$	60 min
trace completion	0.990	$\geq$	60 min
api latency p95	500 ms	$\leq$	60 min

SLO status is exposed via the dashboard and feeds into the alerting worker's SLO breach rule.

E. Automated Alerting and Incident Lifecycle

The alerting worker (`alerting/worker.py`) polls ClickHouse every 15 seconds and evaluates four detection rules:

- **Runaway tokens:** any trace exceeding a configurable token ceiling, defaulting to 4,000.
- **Error burst:** an agent emitting at least three errors within a 15-minute window.
- **Stuck traces:** spans ingested but no reconstruction row written after five minutes.
- **SLO breach:** an SLO failing in at least three of the last five evaluation cycles, using a K-of-N gate to suppress transient noise.

Each alert is routed through the incident state machine (`alerting/incidents.py`): new incidents transition to `open` and fire a webhook; re-fires against an open incident increment xtttoccurrence_count without re-paging; resolving and re-firing creates a fresh incident. Stale incidents (no re-fire in 600 seconds) are auto-resolved. This model eliminates alert storms while pr eserving page-on-recurrence semantics.

IV. Stakeholder Personas

A. ML Engineer / Pipeline Developer

ML engineers are the primary instrumentation consumers. They use `@instrument_node` to add tracing to new agent nodes with a single decorator, inspect the Timeline waterfall to understand per -agent latency, and read the Decision Chain to audit agent reasoning paths. The blame leaderboard helps them identify which agents to optimize first when token budgets are exceeded.

B. Platform / SRE Team

Platform engineers operate the collector, engine, and storage layer. They monitor the six SLOs on the SLO dashboard, receive webhook pages for SLO breaches and error bursts, acknowledge and resolve i ncidents through the Incidents page, and consult the runbook (`docs/runbook.md`) for remediation procedures. The WAL provides durability assurance during maintenance windows.

C. Product / Research Team

Product teams use the cross-trace blame leaderboard over 24h and 7d windows to understand systemic cost and reliability patterns across the pipeline fleet. The `/agents/blame` endpoint feeds c ustom analytics dashboards and capacity-planning exercises. Decision event data informs product decisions about confidence thresholds and candidate selection strategies.

V. Technologies Used

A. Protocol Buffers and gRPC

The wire protocol between SDK clients and the collector is defined in `proto/tracing.proto` using Protocol Buffers 3. gRPC provides strongly-typed, bidirectional-streaming RPC with HTTP/2 mult iplexing, enabling high-throughput streaming ingestion (`StreamSpans`, `StreamDecisions`) alongside low-latency unary calls. The Python gRPC runtime uses asyncio for the server and a êxtttThreadPoolExecutor for the non-blocking client, isolating agent execution from collector latency.

B. ClickHouse

ClickHouse is used as the analytical store for all time-series span and trace data. Its columnar storage and vectorized query engine provide sub-second aggregations over millions of spans for the bla me leaderboard and SLO evaluator. The `ReplacingMergeTree` engine handles idempotent engine reruns without application-level deduplication logic. Per-column TTL expressions enforce the governa nce-defined retention policy.

C. LangGraph and LangChain

The reference demo pipeline is built on LangGraph (v0.2.34), a graph-based orchestration framework for stateful LLM agents. LangGraph's state-passing model is used directly for context propagation: texttt@instrument_node reads and writes four reserved keys (`_trace_id`, `_vector_clock`, `_parent_span_id`, `_tenant_id`) in the shared state dict, requiring zero changes to agent business logic.

D. FastAPI and Uvicorn

The query API is built on FastAPI (v0.115.0), which provides automatic OpenAPI documentation, Pydantic request/response validation, and native async support via Uvicorn. The SSE live-stream endpoint uses `sse-starlette` for push-based trace delivery to the UI, avoiding the polling overhead of REST polling.

E. React, Vite, and Tailwind CSS

The frontend is a single-page application built with React 18, bundled with Vite for fast hot-module replacement during development. Tailwind CSS provides utility-first styling. TypeScript type defin itions in `ui/src/types.ts` mirror the FastAPI response schemas exactly, catching schema drift at compile time.

F. Python Ecosystem

The backend is implemented in Python 3.11+ using: `clickhouse-connect` for ClickHouse connectivity, `pydantic` for decision payload validation and API schema enforcement, `python -dotenv` for environment configuration, and `certifi` for SSL certificate verification when invoking real LLM endpoints.

VI. Testing

The test suite comprises 18 test files totaling over 10,600 lines of test code, covering unit, integration, and end-to-end scenarios.

A. Unit and Integration Tests

All backend modules are tested with `pytest`. Key test files include:

- `test_engine.py`: DAG reconstruction correctness under out-of-order arrival, concurrent fanout/fan-in, missing spans, and degenerate single-node traces. V1 and V2 blame corr ectness under controlled latency/token/error distributions.
- `test_collector.py`: gRPC server acceptance/rejection, WAL write-before-ACK invariant, batch flush correctness, and WAL replay on startup.
- `test_api.py`: All REST endpoints including cursor pagination correctness, filter combinations, SSE stream delivery, and error response shapes.
- `test_alerting.py`: All four alert rules, incident state machine transitions (new, existing, reopened), webhook firing, and auto-resolve logic.
- `test_governance.py`: Metadata allowlist enforcement, regex-based PII redaction (email, credit card, bearer token), and sensitive-key redaction.
- `test_e2e_tenant_isolation.py`: End-to-end verification that spans ingested under tenant A are invisible to queries under tenant B.

B. Engine Smoke Test

A standalone offline smoke test, run as `python -m engine.tests`, validates the DAG reconstruction and blame pipeline without requiring a running ClickHouse instance, enabling fast CI pre-flight checks.

C. Demo Pipeline as Integration Test

Running `DEMO_MODE=true python -m demo.pipeline` exercises the full stack—SDK instrumentation, collector ingestion, WAL durability, engine reconstruction, and API delivery—against a real (containerized) ClickHouse instance. The `DEMO_CODER_FAILURE_RATE` parameter (default 0.5) injects controlled hallucination errors to exercise blame attribution and alerting rules under fau lt conditions.

D. Coverage Summary

TABLE II
TEST COVERAGE BY MODULE GROUP

Module Group	Test Lines	Key Scenarios
Engine (DAG + Blame)	199	Out-of-order, fan-in, CI bounds
Collector + WAL	224	Durability, replay, metrics
API + Pagination	503	Filters, cursors, SSE, CORS
Alerting + Incidents	486	Rules, state machine
SLO Evaluation	290	All 6 SLOs, breach detection
Governance	77	Redaction, allowlist, TTL
Tenant Isolation	83	Cross-tenant isolation E2E
Total	1,862	–

VII. SECURITY AND GOVERNANCE

A. Multi-Tenant Isolation

Tenant identity is resolved from the `X-Tenant-ID` request header by the shared helper `shared/trace_auth.py`. The default tenant, `default`, is used in single-tenant deployments. Every ClickHouse read and write includes a `WHERE tenant_id = ?` predicate, enforced at both the collector and API layers. The end-to-end isolation test, `test_e2e_tenant_isolation.py`, verifies that cross-tenant span leakage does not occur under concurrent ingestion.

B. Metadata Governance

The governance module, implemented in `shared/governance.py`, applies a three-stage pipeline to every span's metadata before persistence:

1) **Allowlisting:** Only keys present in `DEFAULT_METADATA_ALLOWLIST`, such as `component`, `error_message`, `latency_ms`, `output`, and `semantic_failure_type`, are retained. All other keys, including any beginning with underscore, are silently dropped.
2) **Key-based redaction:** Keys matching a configurable sensitive list, including password, API key, token, secret, authorization, and cookie, are replaced with `[REDACTED]`, applied recursively to nested dictionaries and lists.
3) **Pattern-based redaction:** Regex patterns detect and replace email addresses with `[EMAIL]`, credit card numbers with `[CARD]`, and bearer tokens with `[BEARER]` within string values.

Metadata payloads are truncated at 4,000 characters, configurable via `TRACE_METADATA_CHAR_LIMIT`.

C. Data Retention

Per-table TTL values are enforced at the ClickHouse schema level:

- `raw_spans`: 30 days
- `reconstructed_traces`: 90 days
- `slo_status`: 180 days
- `incidents`: 365 days

TTL values are configurable via environment variables and applied idempotently by `db/init_db.py`.

D. Decision Rationale Quality

The SDK validates decision rationale strings against a set of quality constraints before accepting them: the rationale must begin with an action verb from a curated list (route, delegate, select, dispatch, reject, approve, escalate, retry, skip, generate, assign, forward, invoke, trigger, abort), must be between 10 and 512 characters, and must not contain filler phrases (e.g., "as an AI", "I will", "please note"). Violations are recorded in span metadata under `rationale_violations` for operational review.

VIII. CONCLUSION

TraceAggregator addresses a class of observability problems that existing tools leave unsolved: causal ordering across asynchronous agent boundaries, blame attribution with statistical uncertainty, and first-class recording of AI decision events. The four contributions — the instrumentation SDK, DAG reconstruction engine, two-tier blame system, and operational control plane — each target a specific gap rather than extending a general-purpose tracing architecture.

Unlike conventional distributed tracing tools, TraceAggregator records AI decision events as first-class records, applies metadata governance before persistence, and provides confidence intervals in blame attribution rather than raw point estimates. The test suite (1,862 lines across 18 files) and end-to-end tenant isolation tests verify correctness across all major failure modes.

Future work includes: (1) a distributed collector tier with horizontal scaling and load-balanced ingestion for high-throughput pipelines; (2) anomaly detection on vector clock patterns to identify coordination anti-patterns (e.g., excessive synchronization, deadlock candidates); (3) integration with OpenTelemetry collector as an alternative ingestion path; and (4) cost attribution beyond token counts to include API pricing tiers and batched vs. streaming inference costs.

ACKNOWLEDGMENT

The authors would like to thank Prof. Rakesh Ranjan for his guidance and feedback throughout the development of this project. His instruction in CMPE 273 (Enterprise Distributed Systems) at San Jose State University provided the theoretical foundations and engineering discipline reflected in this work. We also thank our peer reviewers for their constructive suggestions, and the open-source communities behind LangGraph, ClickHouse, gRPC, FastAPI, and React, whose platforms made this system possible.

REFERENCES

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[2] B. H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google Technical Report, 2010.

[3] OpenTelemetry Authors, "OpenTelemetry Specification," CNCF, 2023. [Online]. Available: https://opentelemetry.io/docs/specs/otel/

[4] LangChain, Inc., "LangGraph: Build Stateful, Multi-Actor Applications with LLMs," 2024. [Online]. Available: https://github.com/langchain-ai/langgraph

[5] Y. Zarubin and A. Milovidov, "ClickHouse: Lightning-fast analytics database," *Proceedings of the VLDB Endowment*, 2022.

[6] B. Efron, "Bootstrap Methods: Another Look at the Jackknife," *Annals of Statistics*, vol. 7, no. 1, pp. 1–26, 1979.

[7] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Proceedings of the Workshop on Parallel and Distributed Algorithms*, 1988, pp. 215–226.

[8] T. Vaillancourt, "Zipkin: A distributed tracing system," OpenZipkin, 2012. [Online]. Available: https://zipkin.io

[9] Uber Technologies, "Jaeger: End-to-end distributed tracing," CNCF, 2017. [Online]. Available: https://www.jaegertracing.io

[10] Google, "Protocol Buffers Language Guide (proto3)," Google Developers, 2023. [Online]. Available: https://protobuf.dev/