A.I.T.I.A

Autonomous Investigation & Treatment of Infrastructure Anomalies

Technical Architecture Whitepaper

Version 3.0 (Cloud Native) | January 30, 2026

1. Introduction & Problem Statement

Modern cloud infrastructures are complex, distributed systems where failures are rarely isolated. A simple "High Latency" alert could be caused by thousands of underlying factors: a code regression, a database deadlock, a noisy neighbor container, or even a physical hardware failure in the data center (e.g., cooling leak).

Traditional monitoring tools (Datadog, Prometheus, Grafana) are excellent at answering **"What is happening?"** (e.g., "CPU is at 99%"). However, they fail to answer **"Why is it happening?"** and **"What should I do?"**. This gap forces Site Reliability Engineers (SREs) to manually correlate dashboards, read logs, and guess the root cause—a process that is slow, error-prone, and unscalable during 3 AM outages.

A.I.T.I.A (Autonomous Investigation & Treatment of Infrastructure Anomalies) is an **Agentic AI System** designed to solve this. It moves beyond passive observation to active reasoning. By combining **Statistical Causal Inference** (to prove root cause) with **Generative AI** (to interpret context), A.I.T.I.A acts as an autonomous Level-1 SRE that can detect anomalies, diagnose complex multi-variate failures, and prescribe precise remediation commands without human intervention.

2. System Architecture (Deep Dive)

The architecture of A.I.T.I.A has evolved from a local microservices cluster to a sophisticated **Cloud-Native Unified Control Plane** optimized for serverless deployment tiers (like Hugging Face Spaces).

2.1 The Unified Control Plane Pattern

WHY "UNIFIED APP" INSTEAD OF MICROSERVICES?

In the initial v1.0 design, we used separate Docker containers for the API (FastAPI), the Worker (Celery), and the UI (Streamlit). While scalable, this introduced significant overhead for deployment on PaaS platforms that charge per container or require complex orchestration (Kubernetes).

For v3.0, we adopted the **Unified Control Plane** pattern. A single Python process (managed by Streamlit's Tornado server) hosts both the reactive UI and the asynchronous backend logic. This reduces the cold-start time to under 10 seconds and allows the entire "Brain" to run effectively on 2GB RAM / 2 vCPU instances.

2.2 Asynchronous Core & Event Loop

Despite being a single process, the system remains highly concurrent. We utilize **Python's AsyncIO** capabilities heavily. The ingestion pipeline uses `uvloop` (a drop-in replacement for the asyncio event loop) to handle thousands of log lines per second with near-C performance.

- **Non-Blocking Ingestion:** The log receiver endpoint accepts JSON payloads and immediately offloads embedding tasks to a background thread, preventing UI freezes.
- **State Management:** We utilize `st.session_state` as an in-memory "Redux store," persisting the diagnosis history, vector store pointers, and forecast models across user interactions.

3. The Intelligence Pipeline

A.I.T.I.A processes data through a rigorous 4-stage pipeline. This is not a simple "Chat with Logs" wrapper; it is a structured data processing engine.

Stage 1: Semantic Ingestion (RAG Prep)

Raw text logs ("DB Connection Failed" etc.) are first normalized. We use the **Sentence-Transformers** library (specifically `all-MiniLM-L6-v2`) to convert these log lines into **384-dimensional dense vectors**.

WHY `ALL-MINILM-L6-V2`?

We chose this specific model because it is a "Distilled BERT" model. It offers the best trade-off between speed and accuracy. It maps sentences to a 384-dimensional vector space, which is small enough to perform **Cosine Similarity** searches in milliseconds on a standard CPU, yet rich enough to capture semantic nuances (e.g., understanding that "Connection Refused" and "Socket Timeout" are semantically related network errors).

Stage 2: Metric Aggregation

Parallel to log ingestion, the system tracks four Gold Signal metrics in a sliding time window (implemented via Pandas DataFrames for high-performance windowing):

- **Latency (ms):** Response time of the edge service.

- **Throughput (rpm):** Request volume.
- **DB Lock Time (ms):** Time spent waiting for database row locks.
- **Pool Wait (ms):** Time spent waiting for an available worker thread.

4. Causal Inference Engine

This is the "Brain's Left Hemisphere"—the logical, statistical validator. When an anomaly is detected (e.g., Latency > 800ms), the Causal Engine is triggered.

4.1 Graph Construction (PC Algorithm)

We use the **DoWhy** library (built on Microsoft's causal research) to model the system. We don't just assume correlation; we build a **Directed Acyclic Graph (DAG)**.

- **Nodes:** Represent variables (Latency, Errors, CPU).
- **Edges:** Represent directed causal links (CPU -> Latency).

The engine runs the **PC Algorithm (Peter-Clark)**. This involves conditional independence testing. It asks: *"If we hold CPU constant, are Latency and Errors still correlated?"* If the answer is No, it infers that CPU is the "Common Cause" (Confounder). This allows the system to filter out noisy symptoms and identify the true root driver.

4.2 Average Treatment Effect (ATE)

CALCULATING THE "IMPACT SCORE"

Once the graph is built, we calculate the ATE using Linear Regression or Propensity Score Matching.

```
ATE = E[Y | do(X=1)] - E[Y | do(X=0)]
```

In plain English: "If we artificially increased DB Lock Time by 1 unit (Step A), how much would Latency increase (Step B)?"
If the ATE is high (e.g., > 0.8), we have statistical proof that X causes Y.
This is what enables the system to say "Root Cause: Database" with 95% confidence.

5. Machine Learning & Forecasting

While Causal Inference explains the past, Machine Learning predicts the future.

5.1 Time-Series Forecasting (Latency Projection)

We implemented a custom `LatencyForecaster` class located in `src/models/forecasting.py`.

- **Algorithm:** We utilize **Linear Regression** (via Scikit-Learn) fitted on the sliding window of the last 50 data points.

- **Projection:** The model calculates the slope (trend) and intercept of the latency curve. It then extrapolates this line `t+30 minutes` into the future.
- **Thresholding:** The system checks if the projected value at `t+30` exceeds the predefined SLO (Service Level Objective) of 1000ms. If yes, it triggers a "Predictive Alert," warning engineers of an impending breach before it happens.

5.2 Active Learning Feedback Loop (RLHF)

To ensure the AI improves over time, we built a **Reinforcement Learning from Human Feedback (RLHF)** mechanism. The UI includes "Positive/Negative" reinforcement buttons.

- **Data Collection:** User feedback is serialized into a JSONL (JSON Lines) format: `{"prompt": "...", "diagnosis": "...", "human_rating": 1}`
- **Dataset Creation:** This creates a proprietary dataset stored in `data/feedback_dataset.jsonl`. This dataset is "Gold Standard" training data, ready to be used to Fine-Tune future versions of the Llama model (e.g., using LoRA adapters) to align it specifically with the organization's unique infrastructure quirks.

6. LLM Integration & Bridging Logic

This is the "Brain's Right Hemisphere"—the creative, contextual reasoning engine.

6.1 The "Bridging Logic" Innovation

Standard AI models struggle to connect physical world events to digital software metrics. We solved this with a specialized Prompt Engineering strategy called **Bridging Logic**.

PROMPT ENGINEERING STRATEGY

We inject a specific System Prompt into the Groq Llama-3-70B model:
"You are a Senior Site Reliability Engineer. Your goal is to map discrete physical symptoms to their downstream software effects."

We then provide a "RAG Context" block containing similar past incidents. This allows the model to perform **One-Shot Reasoning**:
"I see a report of 'Coolant Leak'. Based on the context, coolant leaks cause thermal throttling. Thermal throttling reduces CPU clock speed. Reduced clock speed increases instruction execution time. This explains the 2600ms DB Lock Time."

This chain-of-thought is what creates the "Magic" where the AI bridges the gap between hardware and software.

7. Comprehensive Tech Stack

A full inventory of the technologies utilized in A.I.T.I.A:

CORE PLATFORM

- **Python 3.11:** Chosen for its robust async support and ML ecosystem dominance.
- **FastAPI Components:** Used for request validation (Pydantic models) and async data processing.
- **Streamlit:** The frontend framework, heavily customized with raw CSS injections for a "Cyberpunk/SRE" aesthetic.

DATA & STORAGE

- **ChromaDB:** Typical embedding storage. We use the persistent flat-file mode for simplicity in the Cloud version.
- **Pandas & Polars:** High-performance data manipulation for time-series windowing.
- **JSONL:** Line-delimited JSON for the feedback dataset (industry standard for LLM fine-tuning data).

MODEL LOGIC

- **Sentence-Transformers:** `all-MiniLM-L6-v2` for RAG.
- **DoWhy + NetworkX:** For Causal Graph generation and ATE stats.
- **Scikit-Learn:** For the Linear Regression forecasting models.

CLOUD & DEVOPS

- **Docker:** Multi-stage build process to keep image size small (~400MB) by stripping build dependencies.
- **Groq Cloud:** Inference-as-a-Service provider. We utilize their LPU (Language Processing Units) which offer 10x lower latency than standard GPUs for token generation.

- **Hugging Face Spaces:** The hosting platform, utilizing their "Docker SDK" runtime.

8. Future Enterprise Scalability

A.I.T.I.A v3.0 is a robust "Team Scale" tool. However, for a Fortune 500 deployment, here is the verified upgrade path:

Phase 1: Database Sharding

- **Current:** Local ChromaDB (SQLite-based).
- **Upgrade:** Migrate to a distributed Vector Search engine like **Milvus** or **Qdrant** running on Kubernetes. This allows searching billions of log lines in milliseconds.

Phase 2: Event-Driven Architecture

- **Current:** Poll-based ingestion (HTTP POST).
- **Upgrade:** Integrate with **Apache Kafka** or **AWS Kinesis**. The ingestion service would become a Kafka Consumer Group, allowing it to handle "Firehose" scale data ingestion without dropping packets.

Phase 3: Model Fine-Tuning

- **Current:** In-Context Learning (RAG) with Llama-3 Base.
- **Upgrade:** Use the `feedback_dataset.jsonl` collected by the tool to train a **LoRA (Low-Rank Adaptation)** adapter. This adapter would be

"baked in" to the model, making it explicitly knowledgeable about the specific company's acronyms, runbooks, and legacy systems without needing massive context windows.

A.I.T.I.A System Architecture Document

Generated for internal review and recruitment portfolio.