WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

Experiment IQ: AI-Powered ML Experiment Diagnosis

Tharun P, Saravanakumar A, Vetriselavan V, Tamil Selvan K

Artificial Intelligence and Data Science (AI&DS), Sri Shakthi Institute of Engineering and Technology, Coimbatore, Tamilnadu, India
Email : tharunp0987@gmail.com.

ABSTRACT

Training a machine-learning model rarely fails with an error message. More often the program finishes and the model is simply poor: it overfits, stalls, becomes unstable or diverges. Engineers diagnose such silent failures by reading loss curves by hand, guessing a cause and retraining, which wastes compute and does not preserve what earlier runs taught. This paper presents the design of ExperimentIQ, an AI-powered system that diagnoses silent training failures using the user's own experiment history and then checks whether the suggested fix works. A Logger SDK captures the configuration and per-epoch metrics of a PyTorch training loop and sends them through a FastAPI backend to PostgreSQL with pgvector. Rule-based detectors identify overfitting, divergence, plateau and instability. A Retrieval-Augmented Generation (RAG) layer combines a configuration-distance search over past runs with semantic search over a troubleshooting knowledge base. A Large Language Model (LLM), used in a fixed pipeline or as a tool-calling agent limited to six steps, turns the compressed evidence into a structured JSON diagnosis that cites run IDs and recommends a configuration change, and a validation layer rejects diagnoses that cite runs which do not exist. The paper also defines an evaluation protocol on a self-generated labelled dataset of 300+ runs with injected failures, comparing a plain-LLM baseline, the fixed pipeline and the agent.

Keywords: ML Experiment Diagnosis, Training Failure Detection, Retrieval-Augmented Generation, LLM Agents, pgvector, FastAPI.

I. Introduction

Machine-learning engineers spend most of their time running experiments: training the same network repeatedly with different learning rates, batch sizes, optimisers and regularisation. Each run produces a configuration and training curves, which are evidence about what works, yet that evidence is usually lost in notebooks, logs and memory.

The costly problems are silent failures: the program finishes normally and the model is simply bad. Validation loss rises while training loss falls (overfitting); the loss explodes or becomes NaN (divergence); improvement stops far too early (plateau); or the loss swings erratically (instability). There is no error message to search for, so a beginner may not notice and an expert must read the curves, recall similar cases and guess a remedy before spending more GPU time.

ExperimentIQ is designed around one testable claim: given a weak training run, the system diagnoses why, retrieves similar past runs as evidence, recommends a fix, and then measures whether the fix works. Deterministic detectors supply explainable facts, a RAG layer supplies evidence from the user's own history, and an LLM assembles both into a structured diagnosis whose cited runs can be verified.

II. Problem Statement

Engineers and students run dozens or hundreds of experiments, and many fail silently. Diagnosing them depends on reading curves by hand, recalling similar cases from memory and trial-and-error retraining, which is slow, inconsistent and expensive. Experiment trackers record runs but do not explain failures; hyperparameter optimisers search settings but do not diagnose; general AI assistants give generic advice without access to the user's history, rarely cite checkable evidence and do not verify that their advice works. The problem is therefore the lack of an evidence-grounded, evaluated system that detects silent failures, explains them from the user's own history, recommends a cited fix and verifies that it helps.

III. Objectives

The objective is to build and evaluate an AI copilot that diagnoses weak ML training runs and recommends verified fixes grounded in past experiments: a Logger SDK and FastAPI/PostgreSQL backend; deterministic detectors for four failure types; a labelled dataset of 300+ runs with injected failures; retrieval over past runs and text; an LLM diagnosis engine with baseline, pipeline and agent modes; citation validation; evaluation including fix success rate; and a Streamlit dashboard with Docker Compose deployment.

IV. Literature Survey

MLflow [1] and Weights & Biases [2] record and plot experiments, and Optuna [3] searches for better settings, but neither explains why a run failed. The failure signatures are well documented: Prechelt [4] formalises early stopping from the generalisation gap, Zhang et al. [5] show that deep networks can fit random labels, and learning-rate sweeps and curve inspection are standard advice [6], [7]. Studies of ML engineering note that failures are often silent and costly to localise [8], [9].

Lewis et al. [10] show that retrievable external memory improves grounding, and ReAct [11] shows that models can interleave reasoning with tool calls. Hallucination [12] and uneven use of long contexts [13] motivate compact context and output validation. Among the trackers, optimisers, AutoML surveys [14] and general-purpose LLM assistants reviewed here, we did not find one that combines automatic silent-failure detection, history-grounded retrieval, cited evidence and measured fix success in a single self-hostable workflow.

V. Proposed System

ExperimentIQ has a layered client–server design. Training scripts, an MLflow importer and a Streamlit dashboard talk over REST to a FastAPI backend [15], which validates requests with Pydantic and stores runs, curves, detections, diagnoses and text embeddings in one PostgreSQL database with pgvector [16]. The AI layer contains the detectors, retrieval engine, context builder, orchestrator (fixed pipeline or tool-calling agent), LLM and validation layer, and an evaluation harness runs all modes on a held-out test set. The design follows five principles: detect before using the LLM; take evidence from the database, not model memory; return validated, cited output; start with a fixed pipeline and add an agent only where it helps; and evaluate against labelled ground truth and a plain-LLM baseline.

VI. Methodology

A. Logging and Failure Detection

The SDK exposes start_run(), log() and finish(), records the seed, git commit, library versions and hardware, and never crashes training if the backend is unreachable. Four pure detector functions return the failure type, severity, start epoch and triggering numbers, with thresholds in a YAML file tuned only on the knowledge-base split. Overfitting is flagged when the smoothed validation loss rises for N epochs after its minimum while training loss keeps falling; divergence when a loss value is NaN or infinite, far above its starting value, or spikes sharply; plateau when validation improvement over the last W epochs is negligible at a poor level; and instability when the coefficient of variation of loss changes is high. Table I summarises them.

Failure	Signature in the curves	Typical fixes
Overfitting	Training loss falls while validation loss reaches a minimum, then rises	Weight decay, dropout, early stopping
Divergence	NaN/inf, loss far above its start, or a large spike	Lower learning rate, clipping, warm-up
Plateau	Validation loss barely improves over a long window	Raise learning rate, change optimiser
Instability	Large epoch-to-epoch swings in loss	Larger batch, lower learning rate, clean labels

TABLE I. FAILURE TYPES HANDLED BY EXPERIMENTIQ

B. Labelled Dataset

Real logs with ground-truth causes are scarce, so small CNNs are trained on a CIFAR-10 [17] subset with PyTorch [18], injecting failures and adding healthy controls. The injected cause, expected symptom and known fix are stored in a hidden ground-truth table, and the runs are split into a knowledge base and a held-out test set (Section X).

C. Retrieval and Context

Numeric hyperparameters embed poorly, so similar runs are found with a weighted distance over normalised configuration fields (log learning rate, log batch size, optimiser, weight decay, dropout, epochs) after filtering by dataset and model family. Troubleshooting passages and failure notes are embedded with a sentence-embedding model [19] and searched by cosine distance in pgvector. The context builder reduces each curve to a few statistics, adds detector outputs, the top-k similar runs and top passages, and drops the lowest-ranked items first to meet a token budget.

D. Diagnosis and Validation

Three modes share one JSON schema (problem, explanation, evidence_run_ids, recommended_changes, confidence). The baseline sees only the curve summary and configuration; the fixed pipeline runs detect → retrieve → context → LLM; the agent calls five typed tools (get_run, run_detectors, find_similar_runs, compare_runs, search_docs) for at most six logged steps. The output is parsed with Pydantic, every cited run ID is checked against the database, one retry is allowed, and a result that still fails is flagged as unverified.

VII. Workflow

The training script logs metrics each epoch, the backend stores them and the detectors record any failure. On a diagnose request the system retrieves similar runs and passages, builds a compact context, obtains a diagnosis from the LLM or agent, validates the citations and shows the result in the dashboard. The recommended change is then applied and the run repeated to measure improvement (Fig. 1).

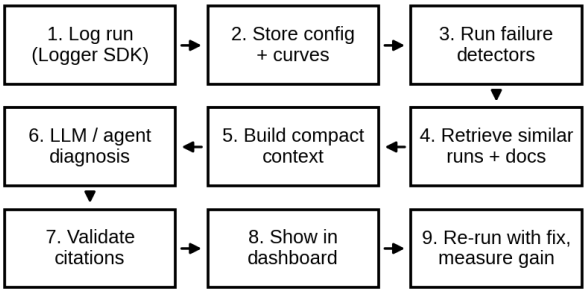


FIGURE 1. END-TO-END EXPERIMENTIQ WORKFLOW

VIII. Scope of This Paper

This paper presents the design of ExperimentIQ and the protocol used to evaluate it. It describes the complete workflow, from the Logger SDK to the Docker Compose deployment, and defines how each component will be measured. Measured results are not reported here; they will be published once the runs on the held-out test set are complete.

IX. Dashboard, API and Reliability

The Streamlit dashboard offers a run list with filters, loss and accuracy curves, detected issues with severity, a Diagnose button with a choice of mode, clickable evidence runs and a side-by-side run comparison. The agent's tool-call trace (tool name, arguments and latency) can be inspected, so the reasoning path is auditable.

The FastAPI backend exposes REST endpoints to create runs, append metrics, list and compare runs, return detections and similar runs, search the knowledge base and request a diagnosis in baseline, pipeline or agent mode. Every request and response is described by a Pydantic model, which gives automatic validation and interactive documentation. Database access is parameterised, credentials are kept in environment variables, the SDK never interrupts training when the backend is unreachable, and a diagnosis that fails validation after one retry is marked unverified.

The service itself runs on an ordinary laptop (an Intel Core i3 or AMD Ryzen 3 class processor, 8 GB RAM and 10 GB of free storage). Building the labelled dataset and re-running fixes involve training several hundred small CNNs, which is practical on an NVIDIA GPU with 8 GB or more of memory or on a free cloud GPU such as Google Colab or Kaggle.

X. Evaluation Protocol

Data. Small CNNs are trained on a CIFAR-10 subset for 10 to 30 epochs. Failures are injected as follows: a learning rate far above the stable range (divergence); a small training subset with many epochs and no weight decay or dropout (overfitting); a learning rate several orders of magnitude too small (plateau); and a very small batch size or randomly flipped labels (instability). Healthy controls use configurations from a known good range.

Split and leakage control. The runs are divided about 70% / 30% into a knowledge base and a held-out test set using a fixed random seed. Runs that share a sampled configuration are kept in the same split so that near-duplicates do not appear on both sides. Held-out runs are excluded from the retrieval index and from threshold tuning, and ground-truth labels are hidden from all diagnosis modes.

Detectors. The initial thresholds are: overfitting, window 3, patience 4, gap scale 1.0; divergence, loss multiple 5.0, spike 4.0 standard deviations; plateau, window 5, minimum relative improvement 0.005; instability, coefficient of variation 0.6. They are tuned on the knowledge-base split only and then frozen before the test set is used.

Metrics. Detector precision, recall and F1 are computed per failure type against the injected labels. Retrieval recall@k is the fraction of truly similar runs (same injected failure and similar configuration) found in the top k results, for k = 3, 5 and 10. Diagnosis accuracy is the share of diagnoses whose problem matches the ground-truth cause. The invalid citation rate is the share of diagnoses citing a run ID that does not exist in the database. A fix succeeds if re-training with the recommended change gives a better best validation loss than the original run. Latency is wall-clock time from request to validated response, and cost is computed from token counts and the provider's price. LLM calls use low temperature and are repeated so that variance can be reported, and proportions are given with 95% confidence intervals.

XI. Evaluation Design and Discussion

The three diagnosis modes differ in how much grounding they receive, as summarised in Table II.

Property	Baseline	Pipeline	Agent
Curve summary + configuration	Yes	Yes	Yes
Detector output	No	Yes	Tool call
Similar past runs	No	Yes	Tool call
Knowledge-base passages	No	Yes	Tool call
Cited run IDs checked	No	Yes	Yes
Order of steps	None	Fixed	Model-chosen (max 6)

TABLE II. DESIGN COMPARISON OF THE DIAGNOSIS MODES

Metric	What it measures	Proposed target
Detector F1	Detection per failure type	≥ 0.85 overfitting, divergence; ≥ 0.80 plateau; ≥ 0.75 instability
Recall@k	Retrieval of similar runs	Reported for k = 3, 5, 10
Invalid citations	Cited runs that do not exist	Reported per mode
Fix success	Validation improves after re-run	Reported per mode

TABLE III. EVALUATION METRICS AND PROPOSED TARGETS

Ablations remove similar-run retrieval, detector output or context compression and replace the fixed pipeline with the agent. Results will be reported whatever the outcome, including cases where the baseline matches the grounded modes on easy problems such as clear overfitting, together with wrong diagnoses, citations that fail validation and fixes that did not help.

XII. Advantages

The design targets silent failures that error-fixing assistants cannot see. Detection is deterministic and explainable and needs no training data. Advice is grounded in the user's own run history with verifiable citations, and the recommended fix is re-run and measured. Run data stays in a self-hosted database, and the modular design lets the detectors, embedding model, LLM provider or vector store be replaced without changing the workflow.

XIII. Limitations

Failures are injected on small CNNs and CIFAR-10, so real-world failures such as other architectures, data bugs or distributed training may differ. Only four failure types are detected, and the detectors depend on thresholds tuned on the knowledge-base split. LLM outputs vary between calls and providers, and each diagnosis has latency and cost. Fix success is tested with a single re-run, so noisy training can mislead. Retrieval depends on the available history, and the compact context is sent to the LLM provider unless a local model is used.

XIV. Future Enhancement

Future versions can add early-failure prediction, a Model Context Protocol interface to the tools, graph-based orchestration, more frameworks and failure types, closed-loop re-runs with the recommended change, authentication and multi-user access, and a local open-source LLM option.

XV. Conclusion

ExperimentIQ addresses training runs that finish without errors but produce poor models. By combining deterministic detectors, retrieval over the user's own history and an LLM that returns structured, cited diagnoses, it aims to turn scattered experiment records into evidence-based advice, with a validation layer and a fix-success experiment to check that the advice is sound. The next step is to run the evaluation protocol and report how much grounding improves diagnosis accuracy and fix success over a plain LLM.

XVI. References

[1] M. Zaharia et al., “Accelerating the Machine Learning Lifecycle with MLflow,” IEEE Data Engineering Bulletin, vol. 41, no. 4, pp. 39–45, 2018.

[2] L. Biewald, “Experiment Tracking with Weights and Biases,” software available from wandb.com, 2020.

[3] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A Next-generation Hyperparameter Optimization Framework,” in Proc. 25th ACM SIGKDD Int. Conf. Knowledge Discovery & Data Mining, 2019, pp. 2623–2631.

[4] L. Prechelt, “Early Stopping — But When?” in Neural Networks: Tricks of the Trade, Springer, 1998, pp. 55–69.

[5] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, “Understanding deep learning requires rethinking generalization,” in Proc. Int. Conf. Learning Representations (ICLR), 2017.

[6] A. Karpathy, “A Recipe for Training Neural Networks,” blog post, 2019. [Online]. Available: https://karpathy.github.io/2019/04/25/recipe/

[7] V. Godbole, G. E. Dahl, J. Gilmer, C. J. Shallue, and Z. Nado, “Deep Learning Tuning Playbook,” GitHub repository, Google Research, 2023.

[8] D. Sculley et al., “Hidden Technical Debt in Machine Learning Systems,” in Advances in Neural Information Processing Systems 28 (NeurIPS), 2015.

[9] S. Amershi et al., “Software Engineering for Machine Learning: A Case Study,” in Proc. IEEE/ACM 41st Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 291–300.

[10] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Advances in Neural Information Processing Systems 33 (NeurIPS), 2020.

[11] S. Yao et al., “ReAct: Synergizing Reasoning and Acting in Language Models,” in Proc. Int. Conf. Learning Representations (ICLR), 2023.

[12] Z. Ji et al., “Survey of Hallucination in Natural Language Generation,” ACM Computing Surveys, vol. 55, no. 12, Art. 248, 2023.

[13] N. F. Liu et al., “Lost in the Middle: How Language Models Use Long Contexts,” Transactions of the Association for Computational Linguistics, vol. 12, pp. 157–173, 2024.

[14] X. He, K. Zhao, and X. Chu, “AutoML: A survey of the state-of-the-art,” Knowledge-Based Systems, vol. 212, Art. no. 106622, 2021.

[15] S. Ramírez, “FastAPI documentation.” [Online]. Available: https://fastapi.tiangolo.com

[16] A. Kane, “pgvector: Open-source vector similarity search for Postgres.” [Online]. Available: https://github.com/pgvector/pgvector

[17] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” Technical Report, University of Toronto, 2009.

[18] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in Advances in Neural Information Processing Systems 32 (NeurIPS), 2019.

[19] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” in Proc. EMNLP-IJCNLP, 2019, pp. 3982–3992.