Agentic Self-Healing for Data & AI Pipelines: An Affordable Vendor-Agnostic Architecture using Open-Source Software

Solomon Eshun
ishango.ai
Accra, Ghana

Dennis Murage
ishango.ai
Nairobi, Kenya

Sharleen Muoki
ishango.ai
Nairobi, Kenya

Chih-Chun Chen
ishango.ai
Berlin, Germany

Stephen Adjignon
ishango.ai
Accra, Ghana

Matteo Staar
EnBW
Karlsruhe, Germany

Michael Rammensee
EnBW
Karlsruhe, Germany

Oliver Angélil
ishango.ai
Zurich, Switzerland

Abstract

Modern organizations rely on data, machine learning, and software delivery pipelines to move data, train models, deploy applications, refresh dashboards, and support business-critical decisions. However, these pipelines often fail because of data quality issues, schema changes, upstream source changes, infrastructure problems, orchestration failures, and model workflow issues. Existing ZeroOps, observability, and AI operations platforms can help teams detect incidents, investigate root causes, and in some cases recommend or execute fixes. However, many of these solutions are expensive, vendor-specific, or difficult for smaller teams to adapt across different tools and environments. This paper first compares existing off-the-shelf solutions for AI-assisted pipeline monitoring, root-cause analysis, and automated remediation, including their strengths, limitations, and practical trade-offs. Based on this comparison, we find that the main gap is architectural rather than technological: the required ingredients for self-healing pipelines already exist, but they are fragmented across vendor-specific platforms, observability tools, incident systems, and open-source components. We therefore propose an affordable, vendor-agnostic reference architecture for agentic self-healing pipelines using open-source and low-cost tools. The proposed architecture combines monitoring, pipeline metadata, incident history, deterministic policy checks, AI-assisted diagnosis, approval workflows, and controlled remediation actions to help teams detect, diagnose, repair, verify, and learn from pipeline issues with less manual effort. The goal is to provide a practical reference architecture that can be adapted across data engineering, machine learning operations, and software delivery environments.

Keywords self-healing pipelines · AI-assisted operations · data observability · root-cause analysis · automated remediation · agentic AI · machine learning operations · software delivery pipelines · vendor-agnostic architecture · open-source software · incident management · Agentic Recovery and Incident Response

1 Introduction

Data, machine learning, and software delivery pipelines have become the connective tissue of modern organizations. A single business dashboard may depend on ingestion jobs that pull from external APIs, transformation layers built with SQL frameworks, feature pipelines that feed model training, batch scoring jobs, and continuous delivery workflows that deploy the services consuming those predictions. Each link in this chain can fail, and in practice each does: schemas drift as upstream teams change source systems, late or malformed data breaks assumptions baked into transformations, infrastructure degrades or runs out of

resources, orchestration dependencies deadlock, and model quality decays silently as the world shifts away from the training distribution [1, 2, 3].

The operational cost of these failures is disproportionately borne by people. Interview studies of machine learning practitioners consistently find that a large share of engineering time is consumed not by building new capability but by diagnosing and repairing broken pipelines [4]. The typical failure workflow remains largely manual: an alert (or worse, a stakeholder) reports a stale dashboard; an engineer reconstructs the failure by traversing logs, orchestrator UIs, and upstream systems; a fix is applied by hand; and the fix is verified by watching the next run. Each step depends on tacit knowledge held by a small number of senior engineers, making the process slow, stressful, and difficult to scale.

The industry response has been a rapidly growing category of AI-assisted operations tooling. AIOps platforms apply machine learning to event correlation and anomaly detection [5, 6]; data observability products monitor freshness, volume, schema, and distributional health of data assets; and, most recently, large language model (LLM) agents have been embedded into operations platforms to summarize incidents, propose root causes, and in some cases execute remediations [7, 8]. Several vendors now market fully "ZeroOps" or autonomous operations experiences. These products are genuinely capable, but they present three recurring adoption barriers for small and medium-sized teams. First, *cost*: pricing is typically per-host, per-table, per-user, or by enterprise contract, and grows quickly with estate size. Second, *vendor coupling*: the most autonomous experiences are only available when the entire pipeline estate lives inside a single vendor's ecosystem. Third, *adaptability*: heterogeneous environments—an Airflow instance here, a dbt project there, a homegrown scoring service, a legacy CI server—rarely fit the happy path these platforms assume.

At the same time, the ingredients for building an equivalent capability have become commodities. Open-source monitoring, lineage, data quality, and workflow tooling are mature; LLM inference is inexpensive and available both as hosted APIs and as locally deployable open-weight models; and agent design patterns such as tool-augmented reasoning are well documented [9, 10]. What is missing is not technology but *architecture*: a reference design that shows how to assemble these commodity parts into a trustworthy self-healing loop with appropriate human oversight.

This paper makes two contributions:

1. **A structured comparison of eight off-the-shelf platforms** for AI-assisted pipeline monitoring, root-cause analysis (RCA), and automated remediation—Databricks Genie ZeroOps, Acceldata ADM, Dynatrace Davis AI with Workflows, Datadog Bits AI with Workflow Automation and Data Observability, Monte Carlo, IBM Databand, PagerDuty SRE Agent with Runbook Automation, and ServiceNow Predictive AIOps with Autonomous Workforce and AI Control Tower—evaluated on scope, autonomy, vendor coupling, and cost (Section 3).
2. **An affordable, vendor-agnostic reference architecture** for agentic self-healing pipelines built from open-source and low-cost components, combining telemetry, pipeline metadata, incident history, deterministic policy rules, LLM-based diagnosis, human approval workflows, and guarded remediation actions (Sections 4 and 5).

The architecture is deliberately prescriptive rather than empirical: it distills recurring patterns from the surveyed platforms and from the AIOps and LLM-agent literature into a design that a team of two to four engineers can stand up incrementally, without committing to any single vendor. We discuss limitations, risks, and buy-versus-build guidance in Section 6.

2 Background and Related Work

2.1 Why pipelines fail

Across data engineering, MLOps, and software delivery, pipeline failures cluster into six recurring classes:

- **Data quality issues.** Null spikes, duplicated records, out-of-range values, and distributional anomalies that violate assumptions embedded in downstream logic. Data cascades of this kind are pervasive and compound silently [3].
- **Schema changes.** Columns renamed, retyped, dropped, or added by upstream owners, frequently without notice; the absence of enforced data contracts makes these the canonical breaking change [11].
- **Upstream source changes.** API version bumps, altered export cadences, moved file locations, revoked credentials, or semantic changes in how a source system populates a field.

- **Infrastructure problems.** Out-of-memory kills, disk exhaustion, spot-instance preemption, network partitions, expiring certificates, and quota limits.
- **Orchestration failures.** Dependency deadlocks, misconfigured schedules, stuck sensors, backfill collisions, and retry storms in tools such as Airflow [12].
- **Model workflow issues.** Training divergence, feature skew between offline and online paths, prediction drift, and gradual performance decay [1, 13].

These classes differ in where they surface (data plane, control plane, or model quality plane) but share a diagnostic structure: the observed symptom is usually several hops downstream of the root cause, which is why lineage metadata is central to any effective diagnosis [14].

2.2 The self-healing loop

We use *self-healing* to mean a closed operational loop with eight stages: *detect* (a monitor or test raises a signal), *triage* (deduplicate, classify, and prioritize), *diagnose* (identify the root cause), *plan* (select a candidate remediation), *approve* (a policy gate decides whether a human must confirm), *remediate* (execute the action), *verify* (confirm the system has recovered), and *learn* (record the episode so future incidents resolve faster). Classical site reliability engineering executes this loop with humans at every stage, supported by runbooks and automation for the mechanical parts [15]. The question this paper addresses is how much of the loop can be delegated to software, and under what guardrails, without importing unacceptable risk. In practice this means routing known failures to deterministic rules, and reserving LLM agents for incidents where diagnosis requires contextual reasoning.

2.3 AIOps and LLM-based incident management

AIOps research has historically focused on the left half of the loop: anomaly detection, event correlation, and failure prediction over metrics and logs [5]. Notaro et al. survey more than a decade of such methods and note that automated *remediation* remains the least developed stage [6]. LLMs have recently shifted this frontier. Ahmed et al. show that fine-tuned language models can recommend plausible root causes and mitigation steps for cloud incidents at scale [7], and Chen et al. demonstrate an LLM-based RCA assistant deployed over real production incidents, using retrieval over incident history to ground its diagnoses [8]. In parallel, the agent literature has converged on a pattern—interleaved reasoning and tool use [9]—that maps naturally onto operations work: an agent that can query metrics, read logs, inspect lineage, and consult runbooks can perform the same investigation a human would, faster and with perfect recall of prior incidents [10]. Adjacent tool categories apply the same ideas to narrower domains: Sentry provides AI-assisted error monitoring and debugging for application code [16], and LangSmith Engine detects and diagnoses recurring failures in LLM application traces [17]; both are relevant context, though neither targets self-healing for data and ML pipelines. Commercial operations platforms have productized these advances aggressively, which motivates the comparison that follows.

3 Off-the-Shelf Solutions: A Comparative Analysis

This section surveys eight commercial offerings that span the data observability, application observability, incident response, and IT service management (ITSM) traditions. We selected these platforms because they represent the major approaches currently used for AI-assisted operations: platform-native autonomous operations, data observability, application and infrastructure observability, incident response automation, and enterprise IT service management. The goal is not to provide an exhaustive market survey, but to compare representative systems that expose the architectural trade-offs relevant to self-healing pipelines: detection, root-cause analysis, remediation autonomy, governance, vendor coupling, and cost. We deliberately exclude *implementation frameworks* such as LangGraph, Pydantic AI, Prefect Marvin, and LangSmith from this comparison: they are building blocks for constructing agentic systems rather than off-the-shelf operations platforms, and we return to them as implementation options in Section 4 [18, 19, 20, 17].

We evaluate each platform on five dimensions: *primary scope* (which failure classes from Section 2.1 it targets), *detection* and *RCA* capability, *remediation autonomy* (from alert-only to closed-loop execution), *vendor coupling* (how much of the estate must live in the vendor's ecosystem to realize the value), and *indicative cost tier*. Assessments are based on vendor documentation and public product descriptions as of mid-2026 [21, 22, 23, 24, 25, 26, 27, 28]; capabilities evolve quickly, so the comparison should be read as a snapshot of category structure rather than a procurement verdict.

3.1 Platform summaries

Databricks Genie ZeroOps. Databricks embeds agentic assistance directly into its Data Intelligence Platform: Genie provides conversational access to data and operational context, while Genie ZeroOps—announced in mid-2026 and initially available in preview—is a background agent that autonomously monitors, investigates, and proposes fixes for jobs, pipelines, tables, and ML workloads [21]. Because the platform owns the compute, the catalog (Unity Catalog), the lineage graph, and the orchestrator, its agents act with unusually rich context: they can trace failures through lineage, validate candidate fixes against cloned data in an isolated environment, and apply them once a user approves. The corresponding limitation is scope: the autonomy applies to workloads running *on Databricks*. Pipelines that traverse external orchestrators, warehouses, or delivery systems fall outside the healing boundary, and adopting the capability implies adopting the platform.

Acceldata ADM. Acceldata's Agentic Data Management positions a fleet of specialized agents over an enterprise data observability substrate: agents monitor data reliability, diagnose pipeline and infrastructure issues, propose corrective actions, and execute approved playbooks [22]. It is notable for covering both the data plane (quality, freshness, schema) and the compute plane (Spark, warehouse performance, cost), and for supporting multiple underlying stacks. It remains a proprietary enterprise product: the agent layer, the metadata store, and the automation runtime are all vendor-operated, and pricing follows enterprise data observability norms.

Dynatrace Davis AI + Workflows. Dynatrace pairs its causal AI engine (Davis) with an AutomationEngine that triggers workflows from AI-identified problems [23]. Davis's strength is deterministic causal correlation over a richly instrumented topology model (Smartscape), which yields high-precision root-cause identification for application and infrastructure failures; workflows can then execute remediations such as rollbacks, restarts, or ticket enrichment. The platform is strongest where its OneAgent instrumentation runs—application services, hosts, Kubernetes—and comparatively thin on data-pipeline-native concerns such as table-level quality, dbt models, or training workflows. Consumption-based pricing across hosts, logs, and events is a recurring cost-management theme for adopters.

Datadog Bits AI + Workflow Automation + Data Observability. Datadog combines an LLM assistant (Bits AI) that investigates incidents conversationally, a Workflow Automation product with hundreds of action integrations, and a growing Data Observability offering that adds table freshness, volume, and quality monitoring to its core telemetry platform [24]. The breadth is attractive for teams already standardized on Datadog, and the workflow builder makes human-approved remediation practical. Costs, however, accumulate per product module and per host/GB, and the AI features primarily reason over data already inside Datadog—estates with significant telemetry outside the platform see reduced value.

Monte Carlo. Monte Carlo largely defined the data observability category: ML-driven monitors for freshness, volume, schema, and field-level quality, automatic lineage, and incident management with impact analysis [25]. Its RCA support—correlating an anomaly with recent code, data, or infrastructure changes—is strong within the data warehouse/lakehouse perimeter. Remediation remains largely human-executed: the platform excels at detection, triage, and routing, but does not aim to repair pipelines autonomously. Pricing is enterprise SaaS, typically scoped by monitored tables/users, which smaller teams often find difficult to justify.

IBM Databand. Databand focuses on pipeline observability for data engineering stacks (notably Airflow and Spark): run-level monitoring, SLA tracking, data quality checks, and alerting with lineage context [26]. It provides solid detection and diagnostic context for orchestration and data quality failures but stops short of automated remediation, and as part of the IBM portfolio it is typically procured within larger enterprise agreements. Its pipeline-run-centric data model is a good conceptual fit for the incident store we propose in Section 4.

PagerDuty SRE Agent + Runbook Automation. PagerDuty approaches the problem from incident response: the SRE Agent performs agentic triage and investigation when an incident fires—gathering telemetry from connected monitoring tools, summarizing probable cause, and proposing next steps—while Runbook Automation (formerly Rundeck) provides audited, parameterized execution of operational procedures across arbitrary infrastructure [27]. This combination is comparatively vendor-neutral: PagerDuty orchestrates over whatever monitoring and infrastructure a team already has. Its blind spot is the data plane—it has no native notion of tables, schemas, or model quality—so data incidents must be surfaced by third-party monitors before PagerDuty adds value. Per-user and per-capability pricing is moderate relative to enterprise observability suites.

ServiceNow Predictive AIOps + Autonomous Workforce + AI Control Tower. ServiceNow offers the most complete closed loop of the surveyed platforms: Predictive AIOps correlates events and detects anomalies against a CMDB topology; AI agents (the "Autonomous Workforce") investigate and execute remediations through the platform's workflow engine; and AI Control Tower provides centralized governance—inventory, policy, and audit—over all AI agents operating in the enterprise [28]. The governance layer is genuinely differentiated and anticipates regulatory scrutiny of autonomous operations. The trade-offs are equally clear: the loop runs on ServiceNow's platform primitives (CMDB, ITSM workflows), realizing value requires deep platform adoption, and licensing sits firmly at the enterprise tier. Like PagerDuty, it is infrastructure- and service-centric rather than data-pipeline-native.

3.2 Synthesis

To make the comparison easier to scan, Figure 1 summarizes the eight platforms—plus the proposed architecture introduced in Section 4, included here as a reference point—across capability dimensions relevant to self-healing pipeline operations. The scores are qualitative and based on public documentation; they are not intended as a procurement benchmark, but highlight broad architectural patterns across the market. Figure 2 then positions the eight surveyed platforms on the two axes that matter most for the architecture question: remediation autonomy and vendor coupling.

	Monitoring / detection	Data-quality awareness	Root-cause analysis	Automated remediation	Cross-domain support	Governance / auditability	Open source	Low cost	Vendor-agnostic
Databricks Genie ZeroOps	**strong**	good	**strong**	**strong**	partial	good	none	none	none
Acceldata ADM	**strong**	**strong**	**strong**	good	good	good	none	none	partial
Dynatrace Davis AI + Workflows	**strong**	partial	**strong**	good	good	good	none	none	partial
Datadog Bits AI + Data Obs.	**strong**	good	good	good	good	good	none	partial	partial
Monte Carlo	**strong**	**strong**	good	partial	partial	partial	none	partial	partial
IBM Databand	good	good	good	none	partial	partial	none	partial	partial
PagerDuty SRE Agent + Runbook	partial	none	good	good	good	good	none	partial	good
ServiceNow Pred. AIOps + ACT	**strong**	partial	**strong**	**strong**	good	**strong**	none	none	none
Proposed architecture (Sec. 4)	good	good	good	good	**strong**	good	**strong**	**strong**	**strong**

Figure 1: Capability comparison of the eight surveyed solutions, plus the proposed architecture (bold border), across dimensions relevant to agentic self-healing. Cells are scored on an ordinal scale—none, partial, good, and strong—based on publicly available vendor documentation and product descriptions as of mid-2026 [21, 22, 23, 24, 25, 26, 27, 28]. The scoring is qualitative and intended to summarize category-level trade-offs rather than provide a procurement ranking.

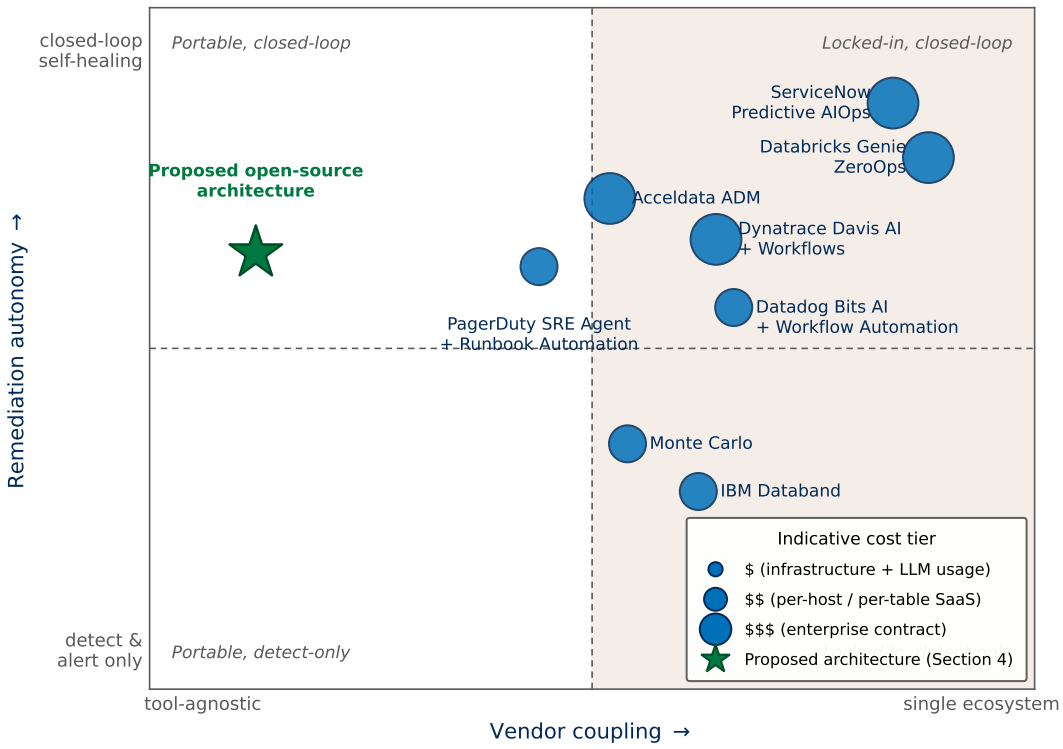

Figure 2: Positioning of the surveyed platforms by vendor coupling (horizontal) and remediation autonomy (vertical); marker size indicates indicative cost tier. Axis placements are the authors' qualitative assessments from vendor documentation, not measured quantities. The most autonomous experiences (upper right) require the deepest ecosystem commitment; the proposed architecture targets the underserved upper-left region.

Three structural observations emerge:

1. **Autonomy correlates with lock-in.** The platforms that close the loop most completely (Databricks, ServiceNow) do so precisely because they own the surrounding context—catalog, orchestrator, CMDB, workflow engine. Autonomy is easy when the healing boundary coincides with the platform boundary; it is the heterogeneous estate that is hard.
2. **No single product covers all three pipeline domains.** Data observability products (Monte Carlo, Databand) are weak on remediation and blind to application delivery; application observability products (Dynatrace, Datadog) are thin on table- and model-level semantics; ITSM/incident products (ServiceNow, PagerDuty) depend on others for detection. Teams routinely need two or three subscriptions to cover the failure classes of Section 2.1.
3. **The mechanisms are convergent and reproducible.** Underneath the branding, every platform combines the same five ingredients: telemetry, metadata/topology, an inference layer (statistical, causal, or LLM), a policy/approval gate, and an action runtime. Each ingredient now has a mature open-source counterpart. This is the observation the remainder of the paper builds on.

4 A Vendor-Agnostic Reference Architecture

Based on the findings from the platform comparison, we propose **Agentic Recovery and Incident Response**, a vendor-agnostic reference architecture for detecting, diagnosing, repairing, verifying, and learning from failures in data, machine learning, and software delivery pipelines. This architecture is not a fixed software product or a mandatory tool stack; it is a modular architecture that defines the responsibilities, interfaces, and safety controls required for agentic self-healing workflows. Throughout this section we name concrete tools, but they are illustrative rather than prescriptive: the architecture is defined by interfaces and

responsibilities, not by a fixed technology stack, and a practical implementation should select the smallest viable set of tools that fits the organization's existing environment.

4.1 Design principles

The architecture is shaped by six principles, chosen to invert the trade-offs identified in Section 3.2:

- **P1 — Vendor agnosticism through open interfaces.** Every integration point uses an open standard or open-source tool: OpenTelemetry for traces and logs, Prometheus exposition for metrics, OpenLineage for lineage events [14, 29]. The estate being healed (orchestrators, warehouses, CI systems) is treated as replaceable—and so is every component of the architecture itself.
- **P2 — The LLM is a swappable commodity.** All agent calls go through a model gateway so that hosted APIs, model routing providers, and locally served open-weight models—the large language models (LLMs) themselves, e.g., GPT, Claude, or Llama-family models—are interchangeable per task, controlling cost, latency, and data residency.
- **P3 — Deterministic before generative.** Not every incident requires LLM reasoning. Known low-risk failures first pass through a deterministic policy layer that applies rules, thresholds, playbooks, and risk classifications; LLM-based agents are reserved for incidents whose diagnosis requires reasoning across logs, lineage, incident history, or ambiguous evidence. This reduces cost, improves predictability, and limits hallucination risk.
- **P4 — Guarded autonomy, not full autonomy.** Agents may only execute actions drawn from an explicit, version-controlled allowlist, and every action class carries a risk tier that determines whether human approval is required. This follows the SRE principle that automation must be at least as auditable as the human procedure it replaces [15].
- **P5 — Memory is the moat.** The system's compounding value comes from its incident history: every episode—symptom, diagnosis, action, approval, outcome—is stored and retrieved to ground future diagnoses, mirroring the retrieval-grounded RCA shown effective in production settings [8].
- **P6 — Incremental adoption.** Each layer of the architecture is independently useful. A team can deploy monitoring alone, add diagnosis later, and enable actuation last, once trust is established.

4.2 Architecture overview

Figure 3 shows the seven logical layers of the architecture. Layers 2 and 3 are explicitly labelled *Context* because they provide the live evidence and accumulated operational knowledge that the reasoning layer consults; they do not themselves decide or execute remediations. Each layer is defined by a responsibility and an interface; the example tools noted in the figure and below are replaceable implementations of those responsibilities, not requirements.

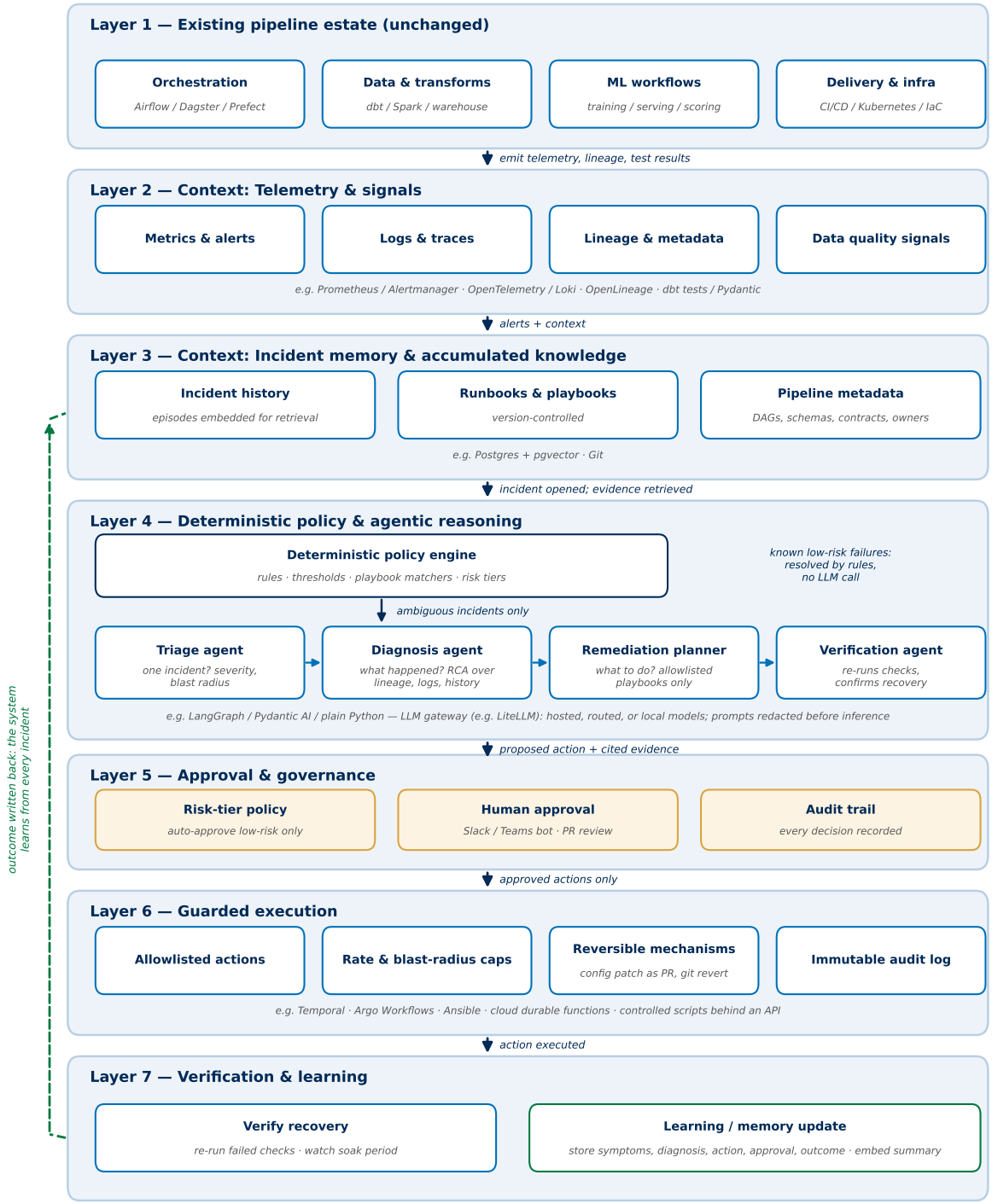

Figure 3: The proposed reference architecture as seven logical layers. The existing pipeline estate (top) is instrumented through open standards; alerts flow through the incident memory layer into a reasoning layer where a deterministic policy engine handles known failures and four LLM agents handle ambiguous ones; proposed actions pass an approval and governance layer (shown in amber, marking where a human is in the loop) before reaching a guarded execution layer; and verification outcomes are written back into incident memory (dashed green arrow), closing the learning loop. Tools named in small print are illustrative examples, not required components.

Layer 1: Existing pipeline estate. The systems being healed—orchestrators (e.g., Airflow, Dagster, Prefect), transformation and compute layers (e.g., dbt, Spark, warehouses), ML workflows (experiment tracking, model serving, batch scoring), and delivery infrastructure (CI/CD, Kubernetes, infrastructure-as-code). The architecture requires no replacement of any of these; it only requires that they emit telemetry.

Layer 2: Context: Telemetry & signals. Four signal families, each an interface with mature open implementations: metrics and alerting (e.g., an existing monitoring system, or Prometheus with Alertmanager [29]), logs and traces (e.g., OpenTelemetry collectors with a store such as Loki), lineage and catalog metadata (e.g., OpenLineage events into Marquez or OpenMetadata [14]), and data quality signals—lightweight data quality checks, dbt tests, or optional tools such as Pydantic [30]. Model-quality signals—drift statistics, evaluation metrics—are exported as ordinary metrics so that no separate ML monitoring stack is needed at the start.

Layer 3: Context: Incident memory and accumulated knowledge. A deliberately boring core: a relational database (e.g., Postgres) holding incidents, their timelines, and their resolutions, with text fields embedded into a vector index for similarity retrieval; runbooks and remediation playbooks maintained as Markdown in Git; and a pipeline metadata registry (DAG definitions, schemas, data contracts, ownership) assembled from the lineage layer. Over time, resolved incidents accumulate evidence about recurring failure signatures, effective actions, and appropriate approvals. Retrieval makes that accumulated knowledge available during diagnosis while still requiring each proposed action to be checked against current evidence and policy. This layer is what converts one-off firefighting into organizational memory (P5).

Layer 4: Deterministic policy and agentic reasoning. The reasoning layer has two tiers. Incidents first pass through a *deterministic policy engine*: version-controlled rules, thresholds, playbook matchers, and risk classifications that recognize known failure signatures and route them directly to a proven remediation without any LLM call (P3). Only incidents the rules cannot resolve—ambiguous evidence, novel symptoms, cross-system failures—are escalated to four cooperating LLM agents, each an LLM invocation with task-specific tools and prompts following the reason-and-act pattern [9]. The four agents have deliberately separate responsibilities:

- The *triage agent* decides whether related alerts belong to a single incident, classifies the failure against the taxonomy of Section 2.1, estimates blast radius from lineage (which downstream assets and consumers are affected), and assigns severity.
- The *diagnosis agent* investigates *what happened*: it retrieves similar past incidents, walks the lineage graph upstream from the failing asset, queries logs and metrics around the failure window, and produces a root-cause hypothesis with cited evidence. Grounding in retrieved history and live telemetry—rather than free generation—is the key defense against confabulated diagnoses [7, 8].
- The *remediation planner* decides *what should be done*: it maps the diagnosis to a candidate action *from the allowlisted playbook library only*. Diagnosis and planning are intentionally distinct responsibilities—one answers "what happened?", the other "what should we do?"—so each can be audited, evaluated, and improved separately. If no playbook applies, the planner's only permitted output is an escalation with a well-structured summary—an outcome that is itself valuable, since it hands the on-call engineer a completed investigation.
- The *verification agent* re-runs the checks that detected the failure, watches recovery metrics for a configurable soak period, and either closes the incident or escalates with full context.

The control plane connecting these agents is a small state graph, and any agent orchestration approach can implement it: LangGraph, Pydantic AI, Agno, Semantic Kernel, AutoGen, plain Python services, or a custom state machine [18, 19]. Routing between agents and their tools is driven by the triage agent's classification: a data-quality incident is routed to lineage and schema-diff evidence, an infrastructure incident to logs and metrics, and so on. This routing is implemented as conditional edges in the state graph rather than as a separate router agent, since the classification needed to route already exists as the triage agent's output. All agents call models through an *LLM gateway* (e.g., LiteLLM or a thin internal proxy [31]), behind which hosted APIs, model routing providers, and locally deployed open-weight models are interchangeable depending on cost, privacy, latency, and data-residency requirements (P2). The gateway also enables policy-based routing: high-volume triage tasks may use cheaper hosted models, sensitive incidents may be restricted to locally hosted models, and complex diagnosis tasks may use stronger frontier models when allowed by data-governance policy. Before incident context is passed to any model, the gateway applies data minimization and redaction: secrets, tokens, personal data, customer identifiers, and unnecessary raw logs are removed or

masked, and agents receive only the evidence required for diagnosis rather than unrestricted access to the full production environment.

Layer 5: Approval and governance. Between planning and execution sits a policy gate implemented where engineers already work: a Slack/Teams bot for operational approvals and pull-request review for code-level changes. Listing 1 shows the shape of a risk-tier policy. Low-risk, reversible actions (retry a task, refresh a materialized view) may be auto-approved; medium-risk actions require one approval; high-risk actions (schema migrations, production deployments, anything touching data deletion) always require a human and are often better left as recommendation-only. Every decision—automatic or human—is recorded in the audit trail.

Listing 1: Example risk-tier policy for remediation actions (YAML).

```yaml
actions:
  retry_task:
    risk: low
    auto_approve: true
    max_per_day: 5          # per pipeline
  backfill_partition:
    risk: medium
    auto_approve: false     # one approver in Slack
    requires: [diagnosis_confidence >= 0.8]
  rollback_deployment:
    risk: medium
    auto_approve: false
    mechanism: git_revert_pr
  alter_schema:
    risk: high
    auto_approve: never     # recommendation only;
                            # human executes via PR
```

Layer 6: Guarded execution. Approved actions execute through a guarded workflow execution layer rather than by giving agents direct shell access. Examples include Temporal, Argo Workflows, Ansible, cloud-native durable functions, or controlled internal scripts behind an API [32, 33]. The execution layer enforces the allowlist, applies rate limits and blast-radius caps (e.g., at most N automated actions per pipeline per day), executes changes preferentially through reviewable mechanisms (a config patch becomes a pull request; a rollback becomes a git revert plus redeploy), and writes an immutable audit record of every action, its initiator, and its approval trail.

Layer 7: Verification and learning. The loop does not end when an action executes. The verification agent confirms recovery, and a *learning/memory update* step then writes the complete episode back into the incident memory layer: symptoms, diagnosis, selected action, approval decision, remediation outcome, verification result, and final resolution. Incident summaries are embedded for similarity search so that future diagnoses retrieve and reuse them. This step is what makes the memory of Layer 3 compound over time (P5): every resolved incident makes the next diagnosis faster and better grounded.

4.3 Example implementation options and indicative cost

Table 1 lists example implementation options for each architectural concern. The tools listed are illustrative rather than prescriptive: the architecture is defined by interfaces and responsibilities, not by a fixed technology stack, and a practical implementation should select the smallest viable set of tools that fits the organization's existing environment. In most rows the minimal implementation is a tool the team already runs. Appendix B compiles a directory of all tools and platforms mentioned in this paper, with official links.

Table 1: Example implementation options for the reference architecture. Tools are illustrative, not prescriptive; start from the minimal column and substitute or extend only where the environment requires it.

Architectural concern	**Minimal implementation**	**Possible substitutes or extensions**	**Notes**
Metrics & alerts	Existing monitoring system, or Prometheus + Alertmanager	Grafana, Datadog, CloudWatch, Azure Monitor	Reuse whatever already pages the team
Logs & traces	Existing log store, or OpenTelemetry + Loki	ELK, Grafana Tempo, Datadog Logs	Structured logs around failure windows are what agents consume
Lineage & metadata	Orchestrator metadata and pipeline definitions	OpenLineage, OpenMetadata, Marquez	A dedicated lineage platform is optional at the start
Data quality checks	dbt tests or lightweight custom checks	Pydantic	Results exported as metrics; keep validation lightweight
Incident & memory store	Postgres + pgvector	SQLite, Qdrant, OpenSearch	Boring by design; embedded summaries enable retrieval
Deterministic policy layer	Version-controlled rules and risk tiers (YAML + a small rules service)	Open Policy Agent, custom rules engine	Handles known failures before any LLM call
LLM gateway	LiteLLM or a thin internal proxy	Direct provider SDKs, internal routing service	Hosted APIs, routing providers, and local open-weight models interchangeable
Agent orchestration	LangGraph or plain Python services	Pydantic AI, Agno, Semantic Kernel, AutoGen, custom state machine	The control plane is a small state graph; any framework—or none—can express it
Approval workflow	Slack or Teams bot	GitHub/GitLab pull-request review	Approvals live where engineers already work
Guarded actuation	Existing scripts behind a controlled API	Temporal, Argo Workflows, Ansible	Durable, audited, rate-limited execution

The steady-state cost has three components: (i) infrastructure for the monitoring and agent services—a few small VMs or a modest Kubernetes namespace; (ii) LLM inference, which is dominated by diagnosis calls and is bounded by incident volume rather than estate size (a team handling tens of incidents per week should expect LLM spend in the tens-to-hundreds of dollars per month range, depending on model choice)—and which the deterministic policy layer further reduces by resolving known failures without any model call; and (iii) engineering time, the honest headline cost, discussed in Section 6. Crucially, cost scales with *incidents*, not with hosts, tables, or seats—the inverse of the commercial pricing models surveyed in Section 3.

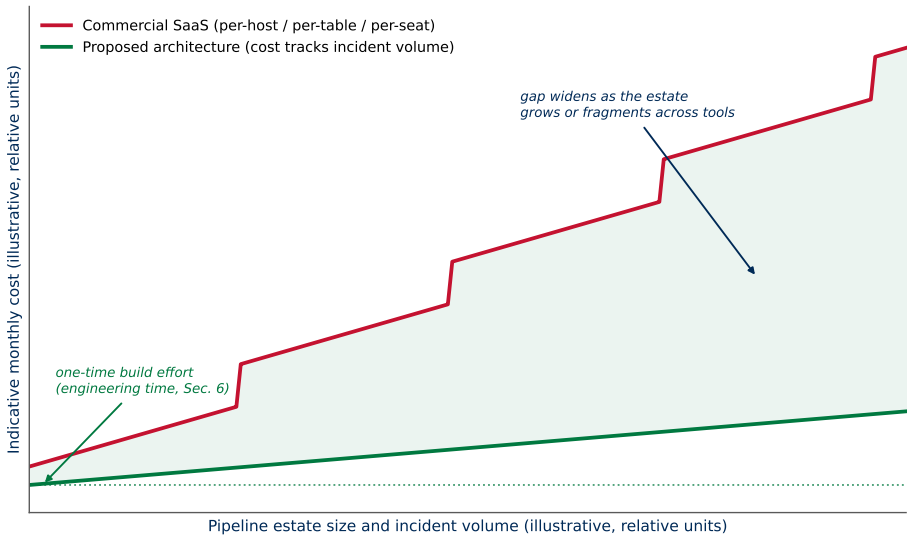

Figure 4: Illustrative cost scaling: commercial per-host/per-table/per-seat pricing grows with estate size, while the proposed architecture's cost grows with incident volume instead, after a one-time build effort. Curves are qualitative, intended to illustrate the pricing-model inversion argued for in this section, not measured data.

5 The Self-Healing Lifecycle in Practice

Figure 5 shows the eight-stage self-healing lifecycle, with the human on-call engineer at the center: consulted at the approval gate, and handed a completed investigation whenever verification fails or no playbook applies.

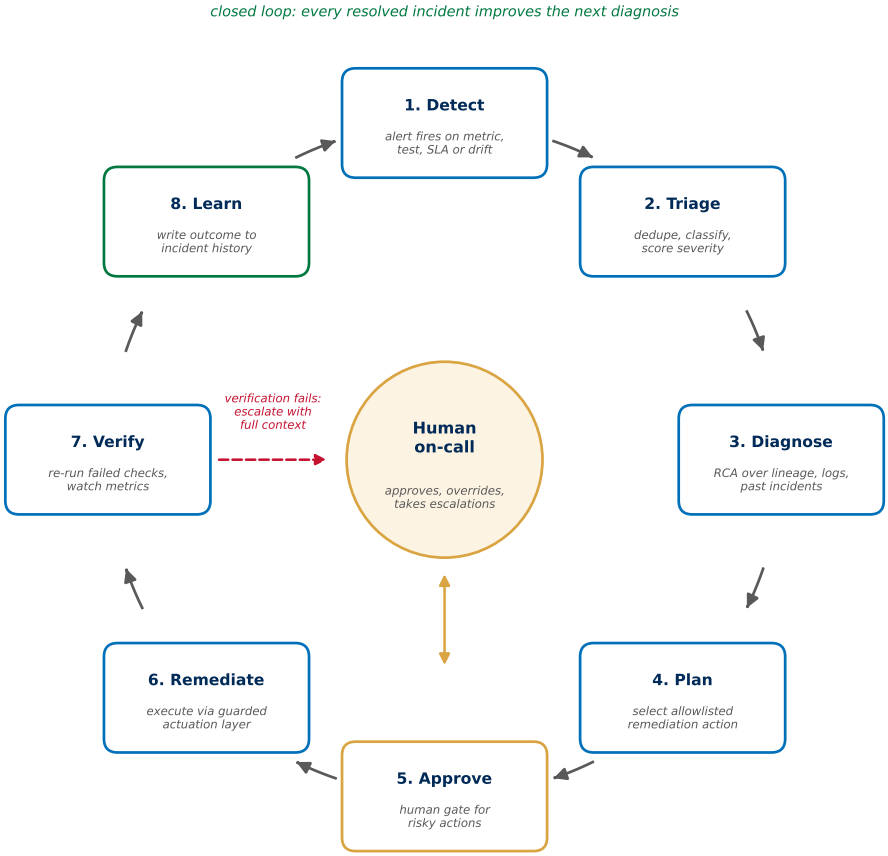

Figure 5: The self-healing incident lifecycle. Stages 1–4 and 6–8 are supported by a combination of deterministic checks, workflow logic, and AI agents; stage 5 remains a policy-controlled approval gate, shown in amber to mark the human-in-the-loop checkpoint. Low-risk reversible actions may be auto-approved, while high-risk or business-sensitive actions require human review. The human on-call engineer approves risky actions and receives escalations with full investigative context when verification fails.

Three worked examples illustrate the loop across the pipeline domains. To keep them technology-flexible we use general terms—an orchestrator task, a schema validation check—naming specific tools only where the detail matters.

Schema drift (data engineering). An upstream team renames `customer_id` to `cust_id` in a source export. A lightweight schema validation check fails at ingestion (*detect*); the triage agent classifies the incident as a schema change and, from lineage, flags fourteen downstream models and two dashboards as at risk (*triage*). The diagnosis agent compares the failing schema against the last successful run's schema, identifies the rename, and retrieves a similar incident from four months earlier (*diagnose*). The planner selects the `column_rename_patch` playbook, which generates a pull request updating the staging model's column mapping plus a data contract violation notice to the upstream owner (*plan*). Because the action is a reviewable PR, the policy tier is medium: one engineer approves (*approve*); CI runs, the patch merges, and the actuation layer triggers a backfill of the affected partition (*remediate*). The verification agent re-runs the failed validation suite and confirms downstream freshness (*verify*), then writes the episode—including the upstream notification—to the incident store (*learn*).

Stuck orchestration (infrastructure). An orchestrator sensor task (here, an Airflow sensor) deadlocks after a spot-instance preemption kills its worker mid-poll. The alerting layer fires on task-duration and worker-loss metrics; the triage agent merges both alerts into one incident. Diagnosis correlates the worker termination event in the logs with the stuck sensor and finds three prior identical incidents, all resolved by

clearing and re-running the task. The planner selects `retry_task`—low-risk, auto-approved, under its daily rate cap—and the actuation layer clears the task via the orchestrator API. Verification watches the DAG complete. Total human involvement: a Slack notification read after the fact. Notably, a failure signature this well-established is exactly what the deterministic policy layer is for: after enough recurrences, a rule can route it straight to the playbook with no LLM reasoning at all.

Model performance decay (MLOps). A weekly churn model's exported performance metric (here, AUC) drifts below its alert threshold. Triage classifies a model workflow issue with low urgency but high business impact. Diagnosis inspects feature drift statistics, finds that a promotional campaign shifted the distribution of two behavioral features, and notes—from the incident store—that retraining resolved a similar episode previously. The planner proposes the `trigger_retraining` playbook, but the policy requires human approval for model releases: the on-call ML engineer reviews the drift evidence, approves retraining into a staging slot, and the verification agent compares champion/challenger metrics before the human makes the final promotion decision. The loop here is deliberately only half-automated: the contribution is that the engineer received a completed investigation and a one-click decision rather than a bare alert [13].

Across all three examples the pattern is consistent. The goal here is not maximal autonomy, but *controlled autonomy*: agents compress investigation and execution mechanics, while irreversible, high-risk, or business-sensitive decisions remain under human control. In our view this division—rather than maximal autonomy—is the correct target for the current generation of LLM agents.

These examples also show that self-healing is not a single action, but a controlled lifecycle. The architecture does not simply detect an error and execute a fix. It gathers evidence, identifies the likely cause, selects from approved remediation options, applies policy-based approval, verifies recovery, and records the outcome. This lifecycle is what makes the system safer, auditable, and reusable across different pipeline environments—and it is the core of the paper's contribution.

6 Discussion

When to buy instead. The proposed architecture is not a categorical argument against commercial platforms. A team already standardized on Databricks obtains much of this capability by turning it on [21]; an enterprise with a mature ServiceNow practice may rationally extend it [28]; and a team with no operations bandwidth at all may be better served by a managed product than by any reference architecture. The build path is most attractive when the estate is heterogeneous, when per-host/per-table pricing is prohibitive relative to team size, when data residency rules out SaaS telemetry, or when vendor exit costs are a strategic concern. A useful heuristic is that the build path becomes attractive when the annual cost of commercial coverage exceeds the engineering effort required to maintain a minimal implementation. This estimate is context-dependent and should be validated against the organization's incident volume, existing tooling, and available platform engineering capacity.

Engineering cost is the real price. The components in Table 1 are free to license, not free to run. Standing up the telemetry layer, writing the first dozen playbooks, and tuning alert thresholds is genuine work, and skipping it does not make the work disappear—it relocates it into every future incident. Principle P6 mitigates the risk: each layer pays for itself independently, and the highest-value early deliverable is usually not remediation at all but the diagnosis agent, which converts every alert into a completed investigation. The practical starting point should be the smallest useful loop: ingest alerts from existing tools, create an incident record, generate a diagnosis summary, route it to the responsible engineer, and store the outcome. Automated remediation, lineage integration, and advanced policy enforcement can be added only after the diagnosis loop proves useful.

Risks of LLM-driven operations. Three failure modes deserve explicit treatment. *Confabulated diagnoses*: an LLM will produce a fluent root-cause narrative whether or not the evidence supports one; grounding in retrieved incidents and live telemetry, requiring cited evidence, and reporting calibrated confidence are necessary mitigations [7, 8]. *Automation bias*: approvals degrade into rubber stamps when the agent is usually right; rotating human review of auto-approved actions, and periodically sampling agent diagnoses for audit, keep the human check meaningful. *Compounding actions*: a wrong remediation can trigger further alerts and further remediations; the rate caps, blast-radius limits, and preference for reversible mechanisms in the execution layer exist precisely to bound this failure mode. Where possible, known failure patterns should be handled by deterministic rules and playbooks before invoking LLM reasoning at all; this reduces cost, improves predictability, and limits the role of LLMs to ambiguous incidents that require contextual rea-

soning across logs, lineage, runbooks, and incident history. More broadly, an agent with actuation credentials is itself new attack surface, and the audit trail must be treated as a security control, not an afterthought.

Data privacy and provider governance. A further concern arises when LLMs are used for operations. Incident context may include logs, traces, customer identifiers, infrastructure names, source code snippets, database schemas, or commercially sensitive information. Sending this context to an external model provider may be unacceptable in regulated or security-sensitive environments—not only as a technical risk but as a governance and trust concern, since providers differ in whether prompts are retained, whether submitted data is used for model training, and what enterprise controls, data-residency options, and audit logs are available. The architecture therefore treats model access as a policy-controlled gateway rather than a direct dependency on a single provider. Sensitive incidents can be routed to locally hosted or enterprise-approved models; prompts are redacted and minimized before inference (Section 4); and provider settings on data retention, model training, and logging must be verified against current official documentation before deployment.

Limitations. This paper is prescriptive, not empirical: the architecture distills patterns from the surveyed platforms and published RCA research rather than reporting a longitudinal deployment, and the cost claims of Section 4.3 are indicative rather than measured. The platform comparison reflects vendor capabilities as publicly described in mid-2026 and will age quickly. The worked examples assume failure classes with recognizable precedents; genuinely novel incidents will always escalate to humans, and a system evaluated only on routine incidents will overstate its coverage. Finally, regulated environments may require stricter approval, explainability, and change-control regimes than the policy model presented here; the governance-first posture of platforms like ServiceNow's AI Control Tower [28] indicates the direction such extensions must take. Future work should instrument real deployments of this architecture and report incident-level outcomes—time to diagnosis, time to recovery, escalation rates, and remediation precision—against a pre-adoption baseline, following the outcome-oriented measurement tradition of the DevOps literature [34].

7 Conclusion

Pipeline operations sit at an awkward point in the market: the platforms that heal most autonomously demand the deepest ecosystem commitment and the largest budgets, while the teams that suffer most from pipeline failures—small, heterogeneous, cost-constrained—are precisely those least able to adopt them. This concern is sharpest for small data teams, organizations in emerging markets, and any environment where tooling is heterogeneous and budgets are tight; for them, affordability and vendor independence are not preferences but preconditions. This paper argued that the gap is architectural rather than technological. By architectural gap, we mean that the missing piece is not a single new algorithm or a completely new class of infrastructure tool; rather, it is a coherent design that connects telemetry, metadata, incident memory, deterministic policy, agentic diagnosis, approval workflows, guarded execution, verification, and learning into one controlled recovery loop. The mechanisms underneath commercial ZeroOps offerings—telemetry, metadata, inference, approval, actuation—are convergent and individually available as mature open-source components, and LLM agents can provide a flexible reasoning layer over telemetry, lineage, runbooks, and incident history, complementing rather than replacing deterministic rules and classical correlation methods. The reference architecture presented here assembles these components into a guarded self-healing loop: deterministic checks and agents triage incidents, diagnose against lineage and incident history, plan only allowlisted actions, verify recovery, and write outcomes back into organizational memory; humans approve what is risky and receive completed investigations for what is not. Its cost scales with incidents rather than estate size, its components are individually replaceable, and its value compounds through the incident memory it accumulates. For data engineering, MLOps, and software delivery teams alike, this architecture offers a guarded, memory-centered, vendor-agnostic pattern for building pipelines that increasingly diagnose, recover, and learn from failures with less manual effort.

References

[1] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, volume 28, 2015. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

[2] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. Challenges in deploying machine learning: A survey of case studies. *ACM Computing Surveys*, 55(6):1–29, 2022. https://doi.org/10.1145/3533378.

[3] Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M. Aroyo. "everyone wants to do the model work, not the data work": Data cascades in high-stakes AI. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2021. https://doi.org/10.1145/3411764.3445518.

[4] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. Operationalizing machine learning: An interview study. arXiv:2209.09125 [cs.SE], 2022. https://arxiv.org/abs/2209.09125.

[5] Yingnong Dang, Qingwei Lin, and Peng Huang. AIOps: Real-world challenges and research innovations. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019. https://doi.org/10.1109/ICSE-Companion.2019.00023.

[6] Paolo Notaro, Jorge Cardoso, and Michael Gerndt. A survey of AIOps methods for failure management. *ACM Transactions on Intelligent Systems and Technology*, 12(6):1–45, 2021. https://doi.org/10.1145/3483424.

[7] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023. https://arxiv.org/abs/2301.03797.

[8] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, 2024. https://doi.org/10.1145/3627703.3629553.

[9] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023. https://arxiv.org/abs/2210.03629.

[10] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. arXiv:2309.07864 [cs.AI], 2023. https://arxiv.org/abs/2309.07864.

[11] Andrew Jones. *Driving Data Quality with Data Contracts*. Packt Publishing, 2023. https://www.oreilly.com/library/view/driving-data-quality/9781837635009/.

[12] Apache Software Foundation. Apache Airflow: Platform for programmatically authoring, scheduling and monitoring workflows. Open-source software, 2025. https://airflow.apache.org/.

[13] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. The ML test score: A rubric for ML production readiness and technical debt reduction. In *Proceedings of the IEEE International Conference on Big Data*, 2017. https://doi.org/10.1109/BigData.2017.8258038.

[14] OpenLineage Project. OpenLineage: An open standard for data lineage. LF AI & Data Foundation, 2025. https://openlineage.io/.

[15] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. https://sre.google/sre-book/table-of-contents/.

[16] Sentry. Sentry: Application performance monitoring and error tracking. Sentry product page, 2026. https://sentry.io/.

[17] LangChain. Find and fix your agent's failures with LangSmith Engine. LangChain documentation, 2026. https://docs.langchain.com/langsmith/engine.

[18] LangChain. LangGraph: Build stateful, multi-actor agent applications. Open-source software, 2026. https://www.langchain.com/langgraph.

[19] Pydantic. Pydantic AI: Agent framework for production LLM applications. Open-source software, 2026. https://ai.pydantic.dev/.

[20] Prefect Technologies. Marvin: A Python framework for agentic AI workflows. Open-source software, 2026. https://github.com/PrefectHQ/marvin.

[21] Databricks. Introducing Genie ZeroOps: Put your data and AI operations on autopilot. Databricks Blog, June 2026, 2026. https://www.databricks.com/blog/introducing-genie-zeroops.

[22] Acceldata. Agentic Data Management Platform (ADM). Acceldata product page, 2026. https://www.acceldata.io/adm.

[23] Dynatrace. Davis AI. Dynatrace documentation. See also AutomationEngine: https://docs.dynatrace.com/docs/platform/automationengine, 2026. https://docs.dynatrace.com/docs/discover-dynatrace/platform/davis-ai.

[24] Datadog. Bits AI SRE. Datadog product page. See also Workflow Automation: https://www.datadoghq.com/product/workflow-automation/, 2026. https://www.datadoghq.com/product/platform/bits-ai/.

[25] Monte Carlo. Data + AI Observability Platform. Monte Carlo product page, 2026. https://www.montecarlodata.com/product/data-observability-platform/.

[26] IBM. IBM Databand: Data pipeline observability. IBM product page, 2026. https://www.ibm.com/products/databand.

[27] PagerDuty. SRE Agent. PagerDuty product page. See also Runbook Automation: https://www.pagerduty.com/platform/automation/runbook/, 2026. https://www.pagerduty.com/platform/ai-agents/sre/.

[28] ServiceNow. Predictive AIOps. ServiceNow product page. See also AI Control Tower: https://www.servicenow.com/products/ai-control-tower.html, 2026. https://www.servicenow.com/products/predictive-aiops.html.

[29] Prometheus Authors. Prometheus: Monitoring system and time series database. Cloud Native Computing Foundation, 2025. https://prometheus.io/.

[30] dbt Labs. Data tests — dbt Documentation. dbt documentation, 2026. https://docs.getdbt.com/docs/build/data-tests.

[31] BerriAI. LiteLLM: Unified gateway for LLM providers. Open-source software, 2026. https://github.com/BerriAI/litellm.

[32] Temporal Technologies. Temporal: Durable execution platform. Open-source software, 2025. https://temporal.io/.

[33] Argo Project. Argo Workflows: Kubernetes-native workflow engine. Cloud Native Computing Foundation, 2026. https://argoproj.github.io/workflows/.

[34] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps*. IT Revolution, 2018. https://itrevolution.com/product/accelerate/.

A Lumis SDK: A Proof-of-Concept Implementation

Lumis SDK (v0.0.1) is an open-source proof-of-concept implementation accompanying the reference architecture described in this paper. It provides a lightweight, deterministic-first framework for investigating failures across data, machine learning, and software delivery pipelines.

- **Repository:** https://github.com/soloshun/lumis-sdk
- **Cookbooks and runnable examples:** https://github.com/soloshun/lumis-sdk/tree/main/cookbook

The cookbook directory is the recommended starting point. It contains synthetic, executable examples spanning data, ML, and software delivery scenarios, showing how project-owned rules and bounded evidence can support incident investigation, with optional model-assisted reasoning when deterministic diagnosis is inconclusive. The demonstrations do not execute remediation actions.

B Tools and Platforms Mentioned in the Paper

This appendix lists the tools and platforms mentioned in the paper. The list is intended as a practical directory, not an endorsement or a required implementation stack. The proposed architecture is tool-agnostic; these tools are examples of components that could satisfy different architectural responsibilities. Links point to official product pages or documentation.

Table B1: Off-the-shelf operations platforms surveyed in Section 3, and adjacent AI-assisted tools discussed in the text. All are commercial products (some with free tiers).

Tool / platform	**Role in the paper**	**Official link**
Databricks Genie ZeroOps	Surveyed platform (Sec. 3)	https://www.databricks.com/blog/introducing-genie-zeroops
Acceldata ADM	Surveyed platform	https://www.acceldata.io/adm
Dynatrace Davis AI	Surveyed platform	https://docs.dynatrace.com/docs/discover-dynatrace/platform/davis-ai
Datadog Bits AI	Surveyed platform	https://www.datadoghq.com/product/platform/bits-ai/
Monte Carlo	Surveyed platform	https://www.montecarlodata.com/product/data-observability-platform/
IBM Databand	Surveyed platform	https://www.ibm.com/products/databand
PagerDuty SRE Agent	Surveyed platform	https://www.pagerduty.com/platform/ai-agents/sre/
ServiceNow Predictive AIOps	Surveyed platform	https://www.servicenow.com/products/predictive-aiops.html
Sentry	Adjacent: AI-assisted application error monitoring (Sec. 2)	https://sentry.io/
Prefect Marvin	Adjacent: agentic AI framework (Sec. 3)	https://github.com/PrefectHQ/marvin
LangSmith Engine	Adjacent: LLM application trace observability and failure diagnosis	https://docs.langchain.com/langsmith/engine

Table B2: Open-source and low-cost components usable in the reference architecture. All are open source unless noted.

Tool	**Architectural concern**	**Official link**
Prometheus	Metrics & alerting	https://prometheus.io/
Alertmanager	Metrics & alerting	https://prometheus.io/docs/alerting/latest/alertmanager/
Grafana	Dashboards & visualization	https://grafana.com/
OpenTelemetry	Logs & traces	https://opentelemetry.io/
Grafana Loki	Log store	https://grafana.com/oss/loki/
Grafana Tempo	Trace store	https://grafana.com/oss/tempo/
OpenLineage	Lineage standard	https://openlineage.io/
Marquez	Lineage store	https://marquezproject.ai/
OpenMetadata	Metadata & catalog	https://open-metadata.org/
dbt tests	Data quality checks	https://docs.getdbt.com/docs/build/data-tests
Pydantic	Schema validation	https://docs.pydantic.dev/
PostgreSQL	Incident & memory store	https://www.postgresql.org/
pgvector	Similarity retrieval	https://github.com/pgvector/pgvector
Git	Runbooks, playbooks, audit	https://git-scm.com/
Open Policy Agent	Deterministic policy layer	https://www.openpolicyagent.org/
LangGraph	Agent orchestration	https://www.langchain.com/langgraph
LangChain	Agent framework	https://www.langchain.com/
Pydantic AI	Agent framework	https://ai.pydantic.dev/
Agno	Agent framework	https://github.com/agno-agi/agno
Semantic Kernel	Agent framework	https://learn.microsoft.com/en-us/semantic-kernel/
AutoGen	Agent framework	https://microsoft.github.io/autogen/
Temporal	Guarded execution	https://temporal.io/
Argo Workflows	Guarded execution	https://argoproj.github.io/workflows/
Ansible	Guarded execution	https://www.ansible.com/
Slack	Approval workflow (commercial)	https://slack.com/
Microsoft Teams	Approval workflow (commercial)	https://www.microsoft.com/en-us/microsoft-teams/group-chat-software
GitHub	PR review, code hosting (commercial)	https://github.com/
GitLab	PR review, code hosting (commercial)	https://gitlab.com/

Table B3: LLM providers, routing, and model serving. Model-provider choice is implementation-specific: teams should select providers based on cost, latency, model quality, data privacy, data residency, enterprise controls, and organizational compliance requirements. This list is not an endorsement and does not imply that all providers are suitable for regulated operations.

Tool / provider	**Category**	**Official link**
OpenAI	Hosted model provider	https://openai.com/
Anthropic	Hosted model provider	https://www.anthropic.com/
Google Gemini	Hosted model provider	https://ai.google.dev/
Groq	Hosted inference provider	https://groq.com/
Together AI	Hosted inference provider	https://www.together.ai/
Fireworks AI	Hosted inference provider	https://fireworks.ai/
OpenRouter	Model routing provider	https://openrouter.ai/
LiteLLM	LLM gateway (open source)	https://github.com/BerriAI/litellm
vLLM	Self-hosted model serving (open source)	https://docs.vllm.ai/
Ollama	Local model serving (open source)	https://ollama.com/
llama.cpp	Local inference (open source)	https://github.com/ggml-org/llama.cpp

Declaration of Generative AI Use

The authors used Claude (Anthropic) in the preparation of this work. For code development, the tool was used to accelerate syntax generation, debugging, and boilerplate code structuring, including the scripts that render the figures; the foundational logic and designs were conceptualized by the authors. For writing, the tool was used to improve language, grammar, and readability. After using this tool, the authors thoroughly reviewed, edited, and verified all resulting text and code, and assume full responsibility for the accuracy and integrity of the final contents of this paper.