International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group | Volume 7 Issue 2 PP 35-45, 2026
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V7I2P105

Original Article

Agentic Data Engineering: LLM-Augmented Pipeline Generation, Self-Healing ETL, and Autonomous Repair

Jeevan Krishna Paruchuri
Independent Researcher, USA.

Received On: 23/02/2026 ***Revised On: 25/03/2026*** ***Accepted On: 02/04/2026*** ***Published On: 09/04/2026***

Abstract - *Production data engineering organizations spend a significant fraction of their on-call time on a narrow class of recurring incidents schema drift, transient cluster failures, late-arriving data, configuration regressions that are individually simple but collectively expensive. This paper presents a position and prototype design for agentic data engineering: the use of large language model (LLM) agents, organized around the ReAct (Reason and Act) framework, as a first responder to pipeline failures, augmenting rather than replacing the on-call engineer. The design is grounded in operational experience from a production banking data platform comprising 35 production pipelines, an observed incident frequency of 15-20 failures per month, and 3-4 schema changes per month exactly the conditions under which a well-scoped agent can absorb a meaningful share of triage work. We describe a tool-use architecture in which the agent has access to a fixed, audited set of read-mostly diagnostic tools (Airflow DAG status, Spark job logs, Delta Lake history, Trino query plans, schema registry diff) and a smaller set of write-capable repair tools that are gated behind explicit human approval before any change reaches production. We propose an evaluation methodology over 6 months of retrospective real incidents plus a synthetic incident benchmark, with a target diagnostic accuracy of 83% and a target mean-time-to-resolution (MTTR) reduction of 36x relative to the human-only baseline. We address the regulatory realities that govern any automated change in financial services SOX change management, GDPR data handling, immutable audit trails and argue that the chain-of-thought reasoning produced by ReAct agents is not a limitation but an audit feature, making the agent's decisions more inspectable than equivalent script-based automation. We conclude with a frank discussion of the limitations: hallucinated diagnoses, the cost of tool calls under high incident volume, the risk of automation bias on the human reviewer, and the need for narrowly scoped agents rather than generalist ones. The contribution of the paper is a practitioner-oriented design framework and a reproducible evaluation protocol, intended to bridge the gap between the LLM agent literature and the operational realities of regulated data platforms.*

Keywords - *LLM Agents, React, Self-Healing Pipelines, AIOps, Data Engineering, Tool Use, Human-In-The-Loop, Pipeline Repair.*

1. Introduction

For most data engineering teams, the first thirty minutes of an incident are spent answering questions that the system already knows the answer to. Which pipeline failed. What was the upstream state. Which run was the last successful one. What changed between then and now. Whether the failure is in the input data, the transformation logic, or the destination. These questions are answerable from logs, metadata stores, and version control, and they are answered by the on-call engineer because nothing else is doing the work. The engineer eventually arrives at a diagnosis, takes a recovery action, files a postmortem, and the cycle repeats the next time an incident occurs.

This paper takes the position that a substantial fraction of this triage work is amenable to automation by an LLM agent operating under the ReAct framework, and that the right framing is augmentation rather than replacement. The agent observes the incident, reasons about possible causes, calls diagnostic tools to gather evidence, proposes a remediation, and waits for human approval before any change reaches production. The human reviewer retains full control; the agent shortens the path from "alert fired" to "informed decision in front of a human."

The grounding for this position is operational. The platform under discussion runs 35 production pipelines orchestrated through Airflow (with a legacy ADF surface still being migrated), built on Spark, Delta Lake, and Trino, deployed on Azure Kubernetes Service. Over the past year we observed 15 to 20 pipeline failures per month and 3 to 4 schema changes per month flowing in from upstream systems. The failures cluster into a small number of recurring categories: schema drift (a column added or renamed upstream), transient infrastructure (executor lost, broker down, throttled storage), late-arriving data, configuration regressions introduced by recent deployments, and resource exhaustion. Each category has a near-deterministic diagnostic procedure that an experienced engineer can execute in minutes and that an LLM agent with the right tools can also execute, with the bonus that the agent never sleeps and never has to context-switch from other work.

The platform is also a financial services platform, which means automated repair is not unconstrained. SOX change management requires that production changes follow a

documented review-and-approval path. GDPR places limits on what data the agent can access during reasoning. Internal audit policies require that every automated action be traceable to a human approver and logged for the seven-year retention window. These constraints rule out a fully autonomous agent and they make the human-in-the-loop pattern not a workaround but the architectural baseline.

We pursue three research questions. RQ1. Can an LLM agent organized around the ReAct framework produce diagnostically correct triage of production pipeline failures at a rate sufficient to be useful, measured against retrospective real incidents and synthetic benchmarks? RQ2. What is the right scope of tools read-only diagnostic versus write-capable repair to give such an agent, and how should the human approval gate be designed to capture the benefit without introducing unacceptable risk? RQ3. How do regulatory constraints in financial services shape the design space, and where does the chain-of-thought reasoning produced by ReAct provide advantages over conventional script-based automation?

The contributions of the paper are three. First, a concrete agent architecture and tool taxonomy grounded in production data engineering practice rather than in synthetic benchmarks. Second, a proposed evaluation methodology over a 6-month retrospective incident corpus plus a synthetic benchmark, with explicit success criteria. Third, a frank discussion of failure modes and limitations, including the 7% failure rate we anticipate from initial prototype evaluation, the risk of automation bias on human reviewers, and the regulatory boundaries that constrain what can ever be fully automated in this domain.

The remainder of the paper is organized as follows. Section 2 reviews the relevant literature on LLM agents, ReAct, AIOps, and self-healing systems, distinguishing the present contribution from each. Section 3 presents the agent architecture and tool taxonomy. Section 4 walks through a worked example incident trace. Section 5 describes the proposed evaluation methodology. Section 6 discusses regulatory and operational constraints. Section 7 enumerates limitations and risks. Section 8 outlines future work and concludes.

2. Related Work

The contribution of this paper sits at the intersection of four research streams, and it is useful to position it against each. The LLM agent and tool-use literature has matured rapidly since 2023. The ReAct framework [1] demonstrated that interleaving reasoning steps with tool calls produces more reliable task completion than either pure chain-of-thought reasoning or pure tool-call sequencing. Subsequent work on tool-augmented LLMs (Toolformer, Gorilla, Voyager) explored richer tool ecosystems and longer-horizon agent loops. The present work adopts the ReAct pattern as its core control loop and contributes a domain-specific tool taxonomy for data engineering, an area underrepresented in the agent literature.

The AIOps literature has long pursued automated incident triage through anomaly detection, log clustering, and rule-based remediation. Systems such as Microsoft's incident triage research and various industry AIOps platforms have demonstrated that statistical methods can correlate alerts and surface likely root causes. These systems are powerful but brittle: they require carefully curated feature engineering, they struggle with novel incident types, and they do not produce inspectable reasoning. The agent approach proposed here is complementary, not competitive: AIOps surfaces the alert and enriches it with statistical signals, the agent reasons over the enriched alert and proposes a diagnosis. The MLOps literature on data quality, drift detection, and feature monitoring (Sculley et al., Polyzotis et al., Breck et al.) provides the conceptual scaffolding for what counts as a pipeline failure and how to detect it. The present work assumes this monitoring infrastructure is in place an agent without good monitoring has nothing to act on and focuses on the layer above: what to do once a failure is detected.

The code generation literature (HumanEval, Codex, GPT-4, Code Llama) has demonstrated that LLMs can produce useful code across a range of tasks, with reported pass rates that vary widely by benchmark. The relevance to pipeline repair is direct: a substantial fraction of pipeline fixes are small code changes (a column rename, a config update, a retry parameter adjustment). The present work draws on this literature for the repair-tool design but differs in two important ways. First, the agent does not produce code from a natural-language specification; it produces code in response to a structured diagnostic context. Second, no generated code reaches production without human review, which changes the risk profile materially. The novel contribution of this paper is the integration of these four streams into a single architecture grounded in regulated production data engineering, with explicit attention to the regulatory and human-factors constraints that shape the design space.

3. Agent Architecture

This section presents the agent architecture in concrete enough detail to implement. The architecture has four components: an alert intake layer, a ReAct reasoning core, a tool registry, and a human-in-the-loop approval gate.

3.1. The ReAct Loop

The core of the agent is a standard ReAct loop. The agent receives an incident context (the alert payload, recent pipeline state, pointers to relevant logs and metadata) as a system prompt. It then iterates through cycles of Thought (the agent reasons about what to do next), Action (the agent calls one of its registered tools with structured arguments), and Observation (the agent receives the tool's response). The loop terminates when the agent emits a structured Diagnosis and Proposal message containing its conclusion and, if applicable, a proposed remediation. The loop is bounded. We cap the agent at a maximum of 12 tool calls per incident, which is enough to handle the recurring failure categories and tight enough to prevent runaway costs and hallucinated

rabbit holes. If the agent reaches the cap without producing a diagnosis, the incident is escalated to human-only handling and the agent's partial trace is attached for context.

3.2. Tool Registry

The tool registry separates read-only diagnostic tools from write-capable repair tools, and the separation is enforced architecturally rather than by convention. The read-only diagnostic tools include: airflow_dag_status (returns the last N runs of a DAG with their states and timing), spark_job_logs (retrieves error logs and stack traces from a specified Spark job), delta_history (returns the version history and operation log of a Delta table), kyuubi_query_plan (returns the logical and physical plan for a recent Trino query), schema_registry_diff (compares the current schema of a source against its previous version), git_blame (returns the commit that last touched a specified pipeline file), and metric_query (executes a Prometheus query and returns the result). These tools cannot modify any production state; they can only read.

The write-capable repair tools are deliberately few. They include propose_dag_retry (constructs a retry of a failed Airflow task with optional parameter overrides), propose_schema_evolution (constructs a Delta Lake schema evolution operation), propose_config_rollback (constructs a revert of a recent configuration change), and propose_resource_increase (constructs a Kubernetes resource increase for a Spark job). Each propose_* tool produces a structured proposal but does not execute it. Execution requires the human approval gate. The asymmetry between the two categories is deliberate. The agent has broad latitude to investigate and narrow latitude to act. This is the same asymmetry that governs human on-call work in regulated environments: read access is widely granted, write access is tightly controlled.

3.3. The Human Approval Gate

Every proposed remediation enters the human approval gate before execution. The gate presents the on-call engineer with: the original alert, the agent's full reasoning trace, the diagnosis, the proposed remediation, and an explicit approve/reject/modify control. The on-call engineer can approve as-is, modify the proposal before approval, or reject and handle the incident manually. Three properties of the gate are essential. First, the agent's reasoning trace is the audit artifact the chain-of-thought is not optional internal scratch space, it is the document that goes into the incident record alongside the human approver's decision. Second, the gate is mandatory; there is no path by which the agent can bypass it, even for "simple" repairs. Third, the gate is fast: the user interface is designed to allow approval in seconds, not minutes, so that the agent's contribution to MTTR is not wiped out by gate latency.

3.4. Tool Call Budgeting and Cost

LLM tool calls cost money and time. At the volume of 15-20 incidents per month, the agent's operating cost is modest on the order of a few dollars per incident at current API pricing but at higher volumes the budgeting matters. We cap each incident at 12 tool calls and add an early-termination heuristic that ends the loop if the agent's confidence in its current diagnosis exceeds a threshold for two consecutive iterations. In aggregate, the expected operating cost is well under the cost of an equivalent amount of human on-call time.

4. Worked Example: A Schema Drift Incident

To make the architecture concrete, this section walks through a representative incident trace of the kind the agent is designed to handle.

The alert fires at 02:14 local time: an Airflow DAG that loads daily transactions from the core banking system has failed during a Spark transformation step with a Column 'merchant_subcategory_v2' not found error. The incident is forwarded to the agent.

Thought 1: The error message suggests a missing column. The most likely causes are an upstream schema change, a typo in a recent code deployment, or a transient issue with the source data. I should check the schema registry diff first.

- Action 1: schema_registry_diff(source="core_banking.transactions", since="24h")
- Observation 1: The diff shows that the upstream transactions table renamed merchant_subcategory to merchant_subcategory_v2 approximately 6 hours ago.

Thought 2: This is a schema rename. I should confirm that no recent code change in our pipeline already accommodated this and that the failure is indeed caused by the upstream change rather than a local regression.

- Action 2: git_blame(file="dags/transactions/transform.py", lines="merchant_subcategory")
- Observation 2: The most recent change to that line was 47 days ago. No local code change is implicated.

Thought 3. The diagnosis is clear: an upstream schema rename from merchant_subcategory to merchant_subcategory_v2 is breaking the downstream transformation. The standard remediation is a schema evolution that maps the new name to the expected internal column, plus a one-line code change in the transformation. I should propose both.

- Action 3: propose_schema_evolution(table="raw.transactions", change="rename merchant_subcategory_v2-> merchant_subcategory_compat") followed by a proposed git diff for the transformation script.

Diagnosis and Proposal. The agent emits its final structured message: root cause is upstream schema rename, evidence is the schema registry diff and the git blame result, proposed remediation is the schema evolution and code change, estimated risk is low (the change is backward-

compatible with downstream consumers), and the proposal awaits human approval.

The on-call engineer is paged at 02:18. The reasoning trace is in front of them when they open the alert. They review the agent's evidence, confirm the diagnosis matches their independent check, approve the proposed remediation, and the change is applied at 02:21. Total MTTR: approximately 7 minutes from alert to remediation. The human-only baseline for a similar incident was approximately 4 hours, dominated by the time the engineer spent finding the schema diff and tracing whether a local code change was implicated.

5. Proposed Evaluation Methodology

This section describes the evaluation we propose to conduct against the prototype, with explicit success criteria.

5.1. Datasets

The evaluation uses two complementary datasets. The first is a 6-month retrospective real incident corpus drawn from the platform's incident log, spanning approximately 100 incidents. Each incident has a known root cause (established by the human postmortem) and a known remediation (the action that resolved it). The agent is run against each incident in a closed-book setting it sees the alert and has access to the tool registry as it would in production, but it does not see the postmortem.

The second is a synthetic incident benchmark of approximately 50 hand-constructed incidents that exercise specific failure categories: schema drift, executor failures, late-arriving data, config regressions, resource exhaustion, and a small number of multi-cause incidents. The synthetic benchmark exists to ensure category coverage that the retrospective corpus may not provide and to enable reproducibility by other researchers.

5.2. Metrics

The primary metrics are: diagnostic accuracy (the percentage of incidents for which the agent's identified root cause matches the human-established root cause), proposal correctness (the percentage of proposed remediations that, on independent review, are judged to be the right action), MTTR contribution (the difference in time-to-resolution between the agent-augmented path and the human-only baseline), and tool call efficiency (the average number of tool calls per incident).

The targets for the prototype evaluation are: 83% diagnostic accuracy, 80% proposal correctness, 36x MTTR reduction, and an average of 6 tool calls per incident. These targets are aggressive but defensible given the structured nature of the incident categories and the focused tool registry.

5.3. Failure Analysis

A central commitment of this evaluation is honest failure analysis. The 17% of incidents where the agent's diagnosis is wrong are not noise to be averaged away; they are the most informative data points in the evaluation. We will categorize each failure: did the agent hallucinate a cause, did it call the wrong tools, did it miss a relevant tool, did the tool return misleading information, or was the incident genuinely outside the scope of the agent's design? Each failure category points to a different mitigation, and the evaluation report will include all of them.

5.4. Reproducibility

The retrospective corpus is necessarily proprietary because it contains operational data from a banking platform. The synthetic benchmark, however, is intended to be released as a reproducible artifact, with each incident specified in enough detail that a third party can replay it against their own agent implementation. This addresses one of the persistent reproducibility problems in agent research, where benchmark tasks are often described only at a high level.

6. Regulatory and Operational Constraints

This section discusses the constraints that shape the design and the ways in which the architecture accommodates them. SOX change management. Every production change must follow a documented review-and-approval path with traceable accountability. The human approval gate satisfies this requirement directly: the gate is the review step, the human approver is the accountable party, and the agent's reasoning trace is the supporting documentation. No agent-proposed change can reach production without passing the gate.

GDPR data handling. The agent is given access to operational metadata and logs, but not to raw customer data. Tool implementations enforce this: spark_job_logs returns log lines but redacts any column values that contain PII; delta_history returns operation metadata but not the data itself. The agent reasons over schemas and structures, not over personal data. Audit trail and retention. Every agent invocation, every tool call, every proposal, and every human approval decision is logged to the platform's immutable audit zone with the standard 7-year retention. The audit log is the same one that captures human actions, which means agent activity is inspectable by the same processes that inspect engineer activity.

Risk tolerance and tool scope. The architectural separation between read-only diagnostic tools and write-capable repair tools is the principal risk control. Within the write-capable tools, the scope is deliberately narrow: schema evolution, retry, config rollback, resource adjustment. The agent cannot modify code repositories, cannot grant permissions, cannot access secrets, and cannot interact with any system outside its explicit tool registry. Expansion of the tool registry is itself a change-managed process.

Chain-of-thought as audit feature. A common concern with LLM-based automation is opacity: how do you explain what the agent did and why. ReAct's chain-of-thought is a direct answer to this concern. The agent's reasoning is captured verbatim, in natural language, and is more inspectable than the equivalent script-based automation,

where the "reasoning" is implicit in the code structure and visible only to engineers who can read the code. We treat the reasoning trace as a first-class audit artifact.

7. Limitations and Risks

This section enumerates the known limitations of the proposed architecture and the risks the team accepts in deploying it. Hallucination. The agent may produce confident but incorrect diagnoses. The mitigation is the human approval gate, which puts a domain expert in the loop before any change reaches production. The residual risk is that the human reviewer may exhibit automation bias and approve incorrect proposals because they look plausible. We address this through reviewer training (the on-call rotation training material includes examples of plausible-but-wrong agent traces) and through requiring the reviewer to articulate one independent check before approving any non-trivial change.

Cost under high incident volume. At 15-20 incidents per month the per-incident LLM cost is negligible. At 1,000 incidents per month it would not be, and the architecture would need batching, model selection (smaller models for simpler categories), or rate limiting. We do not currently have this problem but we acknowledge it as a scaling concern. Out-of-distribution incidents. The agent is good at the recurring failure categories that look like its training distribution. It is poor at incidents that look novel multi-system failures, security incidents, capacity events caused by external factors. The escalation path (12-tool-call budget, escalate to human-only) handles this case but the agent is not adding value on those incidents.

Automation displacing learning. One legitimate concern is that engineers who never have to investigate basic incidents lose the diagnostic skills that come from doing so. We accept this trade-off but note it as a long-term cultural risk to be monitored. The 7% to 17% failure rate is real. The target diagnostic accuracy of 83% means that roughly one in six incidents will get an incorrect diagnosis. The human approval gate is the safeguard, but the cumulative effect of many small wrong proposals still requires reviewer attention.

GPT-3 reached approximately 48% pass-at-1 on the HumanEval benchmark, and GPT-4 has reported significantly higher pass rates in the 80s and above on the same benchmark. Industrial deployments of code generation through tools like GitHub Copilot have made the technology a routine part of many software engineers' workflows. The natural question for a data engineering team is whether the same productivity gains transfer to pipeline development, and if so, under what conditions.

The honest answer is that they transfer only partially, and the conditions matter. This paper argues that the value of LLM code generation in data engineering is largely a function of how constrained the search space is at the point of generation. In an unconstrained environment give the model a free-form prompt and ask for a complete Python pipeline the output is often plausible-looking and silently wrong on details that matter for financial correctness. In a constrained environment give the model a config-driven framework with explicit Source/Transformer/Sink abstractions and ask it to fill in a YAML configuration plus a templated transformer body the output is far more reliable and far more useful.

We pursue three research questions:

- RQ1. In what domains can LLMs reliably generate data pipeline code? Specifically, how do structured, config-driven frameworks change the LLM's task and improve the reliability of its output?
- RQ2. What are the failure modes of LLM-generated code in production data systems, particularly in financial services where correctness is regulated and non-negotiable?
- RQ3. What hybrid architecture can capture the productivity gains of LLM code generation while maintaining the safety properties required for production deployment in regulated industries?

Our thesis is that config-driven pipeline frameworks unlock LLM value by providing structured templates that constrain what the model has to invent. For specific use cases boilerplate aggregation pipelines, schema change detection and repair, dead-letter error handling LLM-generated code can reduce engineering effort by 40 to 60 percent. For other use cases complex stateful transformations, regulatory-critical algorithms, novel business logic LLMs cannot be trusted to generate code that production deployment in financial services would accept. The hybrid architecture we propose places LLMs at the front of the workflow, framework validation and automated tests in the middle, and human review at the gate in front of production. We do not believe LLMs will replace data engineers in regulated environments. We do believe they will substantially change what data engineers spend their time on, and we believe the change is largely positive.

The paper proceeds as follows. Section 2 reviews background on code generation and prior work on LLMs in data systems. Section 3 describes config-driven pipeline frameworks generally and our specific framework. Section 4 details three concrete use cases in which we applied LLM augmentation. Section 5 documents failure modes and limitations. Section 6 presents the hybrid architecture and the governance model around it. Section 7 reports experimental results from an eight-week trial. Section 8 covers deployment lessons. Sections 9 and 10 discuss broader implications and practical recommendations. Section 11 concludes.

8. Config-Driven Pipeline Frameworks

The motivating problem for our framework was the cost of duplication. Across thirty-five production pipelines, the broad pattern was always the same: read from a source, apply a transformation, write to a sink. The specifics varied different sources, different schemas, different aggregation rules but the boilerplate around the specifics was repeated dozens of times in slightly different forms. Copy-paste

pipelines created two compounding problems. First, they multiplied the maintenance burden: a fix to one pipeline often had to be replicated across many. Second, they imposed an onboarding tax on new engineers, who had to reverse-engineer the implicit conventions of each pipeline before they could safely modify it.

The config-driven approach is the standard response to this kind of duplication. A single framework codifies the structure of a pipeline; configuration files specify the variations. The benefits in our case were a reduction in code duplication, faster onboarding (new engineers could configure a pipeline in hours rather than days), and easier auditing because every pipeline followed the same pattern.

8.1. Framework Architecture

Our framework is built around three abstractions: Source, Transformer, and Sink.

A Source specifies how to read data. Supported source types include Kafka, Oracle CDC (via Kafka Connect with JDBC source connectors), ADLS, and S3. A representative source configuration:

- Bootstrap_servers: kafka.prod:9092
- Schema_registry: schema.prod:8081

A Transformer specifies how to transform data. Most transformers are SQL queries, but the framework supports custom Scala logic for cases where SQL is insufficient (for example, the fraud scoring model, which requires programmatic feature construction). A representative transformer configuration:

- SUM(amount) as total,
- FROM source_transactions
- GROUP BY customer_id

A Sink specifies how to write the result. Supported sink types include Delta Lake, Iceberg, and S3. A representative sink configuration:

- path: /mnt/curated/customer_aggregates
- merge_key: customer_id

The framework provides extension points for custom Source implementations (for example, a fixed-width mainframe format reader), custom Transformer logic (for example, the fraud scoring model), and error handlers (dead-letter patterns). Each abstraction is implemented as a Scala trait, with concrete implementations registered as case classes; the framework uses reflection plus the configuration to instantiate the right combination at runtime. Configuration is validated before runtime: schema compatibility between source and transformer, writability of the sink location, and consistency of declared dependencies are all checked before any Spark job is launched.

8.2. Production Framework

The framework supports thirty-five production pipelines spread across customer data, financial transactions, and risk and compliance domains. The configuration footprint is approximately one thousand YAML files, with most pipelines specified in fifty to one hundred lines of configuration. The Scala framework code itself is approximately four thousand lines, plus an additional two thousand lines of custom Transformer implementations for cases where the SQL-only path is insufficient. Maintenance of the framework consumes approximately eight hours per week of engineering time on backward compatibility, new Source types, and bug fixes.

The pain points the framework does not yet solve, and which motivated our exploration of LLM augmentation, are concentrated in three areas. Schema changes from upstream systems occur three to four times per month, and each one requires manual updates to one or more YAML configurations and sometimes to transformer code. Error recovery for new failure modes particularly malformed records that violate validation assumptions requires custom logic that has to be written, tested, and deployed for each new case. Testing new transformers, especially edge cases, is largely manual.

9. LLM-Augmented Pipeline Generation

We applied LLM augmentation to three concrete use cases. In each case, we measured the engineering time required with and without the LLM in the loop.

9.1. Use Case 1: Boilerplate Pipeline from Description

The scenario is the most common: an analyst requests a new pipeline that aggregates customer transactions by month. The manual baseline for building this kind of pipeline in our framework is one to two hours, encompassing the time to understand the desired aggregation, write the YAML configuration, and validate it against sample data.

The LLM-augmented workflow runs as follows. The analyst provides a natural language description: "Create a pipeline that reads the Oracle CUSTOMER_TRANSACTIONS table, aggregates by customer_id and month (sum of amount, count of records), and writes to Delta at /mnt/curated/monthly_customer_txn. The source is CDC via Kafka." This description is wrapped in a prompt that includes the framework's Source/Transformer/Sink schema and a worked example, then submitted to GPT-4. The LLM produces a YAML configuration approximately like the following:

- Bootstrap_servers: kafka.prod:9092
- Schema_registry: schema.prod:8081
- DATE_TRUNC('month', event_time) as year_month,
- SUM(amount) as total_amount,
- COUNT(*) as txn_count
- GROUP BY customer_id, DATE_TRUNC('month', event_time)
- path: /mnt/curated/monthly_customer_txn
- merge_key: customer_id

The analyst reviews the output, makes any required adjustments (for example, additional aggregations or a different merge key), and the framework auto-generates

basic test cases a mock source, a schema validation, and a row count check. The end-to-end time falls from one to two hours to fifteen to thirty minutes, a reduction of approximately 70% on this category of work. Confidence in the output is high, because aggregation SQL is one of the things LLMs do well, and any errors that do occur are caught by the framework's pre-runtime validation.

9.2. Use Case 2: Schema Change Detection and Auto-Repair

The scenario is operationally common and operationally annoying: an upstream Oracle table adds a column or changes a column type, and an existing pipeline fails. The manual baseline for resolving this kind of failure is two to four hours, including the time to detect the failure, debug it, identify the schema change as the root cause, update the pipeline configuration, test the fix, and deploy.

The LLM-augmented workflow starts with the framework detecting the schema change automatically. When a Spark read fails because the source schema differs from what the pipeline expects, the framework captures both the old and new schemas (using the Avro schema registry's evolution tracking) and emits a structured failure message: "Expected customer_id (BIGINT), got customer_id (STRING). Also found new column customer_segment (STRING)."

This structured message becomes the input to an LLM prompt that explains the change and asks for an updated transformer query. The prompt includes the current transformer query, the old and new schemas, and the constraint that the response should preserve the existing aggregation semantics while accommodating the new schema. GPT-4 typically produces output similar to the following:

- SELECT CAST(customer_id as BIGINT) as customer_id,
- SUM(amount) as total
- GROUP BY CAST(customer_id as BIGINT), customer_segment

The framework applies the proposed fix in a sandbox, runs the existing test cases against it, and surfaces the result to a human reviewer. If the tests pass and the reviewer approves, the fix is committed and deployed. The end-to-end time falls from two to four hours to roughly fifteen minutes, the bulk of which is the human review.

The confidence level here is medium rather than high. Type casting is straightforward, but there are subtle cases the LLM can get wrong. If the new customer_id is a STRING because the source switched from a numeric identifier to an encoded reference that requires a lookup table, a naive CAST will compile and run but will produce semantically incorrect joins. The framework catches the structural error but not the semantic one; the human reviewer remains the safety net.

9.3. Use Case 3: Error Recovery and Dead-Letter Handling

The scenario arises when a pipeline encounters malformed records for example, transactions with customer_id IS NULL or with negative amount values that should not exist. The manual baseline for adding a proper dead-letter pattern is four to eight hours, encompassing the design of the validation rules, the implementation of a separate stream for bad records, and the testing of both the happy path and the failure path.

The LLM-augmented workflow starts with the framework detecting the validation failures and reporting them in structured form: "10,000 records failed validation: customer_id IS NULL (8,000 records), amount < 0 (2,000 records)." This report becomes the input to an LLM prompt asking for an updated transformer that filters valid records, a dead-letter sink configuration that captures invalid records with an error reason, and logging for each error type. GPT-4 produces something similar to the following:

- Valid records transformer
- SELECT customer_id, amount, event_time
- WHERE customer_id IS NOT NULL
- Dead-letter transformer
- SELECT customer_id, amount, event_time,
- CASE WHEN customer_id IS NULL THEN 'MISSING_CUSTOMER_ID'
- WHEN amount < 0 THEN 'NEGATIVE_AMOUNT'
- END as error_reason,
- CURRENT_TIMESTAMP() as error_ts
- WHERE customer_id IS NULL OR amount < 0

The accompanying sink configuration adds a second Delta sink at a _dead_letter path. The framework applies the changes, runs them through the test harness, and notifies a human reviewer. The end-to-end time falls from four to eight hours to approximately thirty minutes. Confidence is medium-to-high because the dead-letter pattern is well-known and standard, the LLM has seen many examples of it during training, and the human review focuses on whether the validation rules accurately reflect the business intent.

10. Failure Modes and Limitations

The successes of the previous section should be read alongside the failure modes that motivated our hybrid architecture. We organize the limitations into five categories.

10.1. Hallucinations in Complex Business Logic

The most dangerous failure mode is silent semantic incorrectness in business logic. Consider a fraud scoring rule that should be implemented as: if customer_credit_score < 500 AND txn_amount > 10000 AND time_since_account_open < 30 days, then fraud_probability = 0.95. We have observed LLMs (and not just GPT-4) drop conditions from this kind of rule, producing variants such as: if customer_credit_score < 500 AND txn_amount > 10000, then fraud_probability = 0.95. The dropped time-based condition fundamentally changes the rule. Worse, we have seen LLMs invent formulas that look plausible but bear no

relationship to the documented business logic, such as: if customer_credit_score < 500, then fraud_probability = customer_credit_score / 1000. This is not a fraud rule; it is the LLM filling in the shape of a rule with a plausible-looking calculation.

The impact in financial services is direct and serious. Fraudulent transactions slip through, regulators (OCC, CFTC) expect documented and accurate fraud detection logic, and the audit trail of an LLM-generated rule is not the kind of provenance regulators want to see. The mitigations are non-negotiable for any production deployment: human review of all business logic, explicit unit tests for fraud rules with positive and negative cases, and A/B testing in shadow mode before any production cutover. We do not allow LLM-generated business logic into our production fraud or credit risk pipelines without these gates.

10.2. Stateful Transformations and Window Functions

Stateful transformations particularly window functions, lag and lead operations, session windows, and any transformation that depends on event ordering are a second category where LLM output is unreliable. The challenge is that the syntax of a window function is easy and the semantics are hard. An LLM can generate syntactically correct SQL that uses ROW_NUMBER() or LAG() but applies them with the wrong partitioning, the wrong ordering, or the wrong frame specification. The result is code that compiles, runs, and produces wrong answers.

Our experience includes two specific incidents in which window function logic was subtle enough that an LLM would, on inspection, almost certainly have generated incorrect code. In one case, the correct semantics required that late-arriving events not update past windows; an LLM-generated solution would likely have allowed the updates and quietly corrupted historical aggregates. We have adopted a strict rule: complex window function logic is hand-authored. LLMs are used only for simple aggregations where the windowing semantics are unambiguous.

10.3. Cross-Pipeline Dependencies and Lineage

The third failure mode is cross-pipeline. An LLM generating a new pipeline does not naturally know about the dependency graph among existing pipelines. If pipeline A produces customer features and pipeline B produces a downstream aggregate that depends on pipeline A, an LLM that generates a third pipeline C reading from A may not respect the timing constraints for instance, scheduling C to run before A has completed. The result is data freshness issues that are hard to detect because the pipeline runs successfully but produces stale outputs.

Our mitigation is structural: the framework maintains an explicit dependency DAG and validates new pipeline configurations against it. Configurations that introduce circular dependencies are rejected at validation time. The LLM cannot violate the graph because the framework does not let it.

10.4. Testing and Validation Challenges

Even when LLM-generated code is correct, validating that it is correct is hard. Auto-generating test cases for LLM-generated code is itself a code generation problem, and the quality of those tests determines how much confidence the team can place in the resulting pipeline. Our framework auto-generates basic tests schema validation, row count sanity checks and we ask LLMs to generate additional edge case tests for the specific pipeline. The results are mixed: LLMs can generate basic test cases reliably (for example, "verify that null customer_id is filtered") but miss the more sophisticated edge cases that an experienced engineer would think to test.

10.5. Regulatory and Compliance Risk

The final category is regulatory. CFTC and OCC requirements for documented and auditable algorithms are not optional. LLM-generated code is non-deterministic by default the same prompt, asked twice, may produce subtly different code which complicates audit. Locking the prompt and seeding the model can produce reproducibility, but this requires explicit infrastructure and discipline. The deeper concern is that regulators may not, as of 2023, accept "generated by an LLM" as sufficient documentation of an algorithm's provenance. Our position is that LLMs are acceptable for boilerplate and infrastructure code where the correctness of the output can be independently verified by tests and code review. They are not acceptable for the regulatory-critical algorithms themselves PD/LGD calculations, AML rules, fraud scoring which we continue to author by hand and document manually.

11. Hybrid Architecture and Governance

The architecture we have settled on is explicitly hybrid. An analyst or engineer describes the intent of a new pipeline (or the nature of a failure that needs repair) in natural language. The LLM produces a draft configuration and, where applicable, a transformer body. The framework validates the draft against its schema, dependency, and syntactic rules. A human reviewer reviews the code and the logic. The framework runs the auto-generated and any LLM-generated tests against the pipeline. The pipeline runs in a staging environment against representative data. For pipelines that touch regulatory-critical workloads, the risk and compliance team approves explicitly. Finally, GitOps triggers production deployment.

The architecture has three distinguishing properties. First, the LLM is at the front of the workflow, where its productivity benefits are largest and where its mistakes are cheapest to correct. Second, framework validation and automated testing form a deterministic safety net behind the LLM, catching the structural and behavioral errors that human review might miss. Third, human review remains the gate in front of production. We do not automate around the human review for any pipeline that touches sensitive or regulated data.

11.1. Prompt Engineering

Prompt quality directly impacts code quality, often by more than people expect. Our most effective prompts include four components. First, the framework context: a summary of the Source/Transformer/Sink abstractions and the supported types, so the LLM knows what shape of output is expected. Second, example configurations from the existing pipeline corpus, used as few-shot examples. Third, any business logic that constrains the output (fraud rules, customer segmentation rules, validation thresholds). Fourth, explicit constraints around data quality, regulatory requirements, and performance.

A representative prompt:

- You are a data engineer. Generate a config-driven ETL pipeline.
- Framework: Source (type: kafka/oracle_cdc/etc.),
- Transformers (type: sql/custom),
- Sink (type: delta/iceberg).
- Example pipeline (similar logic):
- topic: customer_events
- SELECT customer_id, SUM(amount) as total
- GROUP BY customer_id
- path: /mnt/curated/customer_totals
- NEW REQUEST: Read Oracle CDC for transactions, aggregate
- By customer_id and day (SUM, COUNT, MAX, MIN of amount),
- Write to Delta at /mnt/curated/daily_transactions.
- Constraints: customer_id must be non-null;
- Merge on customer_id + date.

We spent approximately two weeks of focused effort iterating on prompt templates for the most common pipeline patterns. The investment paid off: few-shot examples in particular reduced hallucination rates by a noticeable margin, and the prompts that included three to five examples performed substantially better than prompts with one or none.

11.2. Version Control and Audit

Every LLM-generated artifact in our environment lives in git. Commits include the configuration (YAML), any transformer code (SQL or Scala), the prompt that produced the code, and the LLM model identifier (GPT-4 with a specific date or version). This provides two important properties: replayability (re-running the same prompt against the same model produces approximately the same output, allowing audit reconstruction), and diff tracking (a human reviewer can see exactly what the LLM changed from a prior template). The risk we explicitly track is that if a particular prompt or model version is later discovered to have a systematic flaw, every pipeline generated by that prompt-model combination is at risk and may need to be re-audited.

12. Experimental Evaluation

We ran an eight-week trial of GPT-4-augmented pipeline generation across the existing pipeline portfolio. The baseline was historical data on engineering time per pipeline, drawn from internal time tracking and PR review records over the prior six months. We measured three metrics: engineering time per pipeline (from initial request to deployment), error rate (bugs found in testing or in production after deployment), and code quality as judged by human reviewers during PR review.

The headline result was an average engineering time reduction of approximately 35%, with significant variation by pipeline category. Boilerplate aggregation pipelines saw the largest reductions, around 60%. Schema evolution and auto-repair work saw similar reductions, around 70%. Pipelines with substantial custom business logic saw much smaller reductions, around 10%, because the LLM's contribution was largely limited to scaffolding, with the human still doing the substantive work.

Error rates were higher for LLM-generated pipelines than for hand-authored ones in initial testing approximately twice as high but the errors were almost entirely simple mistakes: typos in column names, wrong filter conditions, missing predicates. These errors were caught by the framework's tests and by code review before reaching production. After the initial fixes, the LLM-generated pipelines were not measurably worse than hand-authored ones in production.

Code quality, as judged by human reviewers, was generally acceptable. Reviewers identified hallucinations in approximately 15% of LLM-generated code. The hallucinations were concentrated in complex logic exactly the area where we had predicted LLM weakness. Where the LLM was producing simple aggregations, hallucinations were rare; where the LLM was producing anything resembling business rules, hallucinations were the modal failure mode.

Adoption among the engineering team divided into three groups. Approximately 70% of engineers were enthusiastic about the LLM tooling and adopted it actively. Approximately 20% were cautious they wanted more review gates and slower rollout and used the tooling on lower-risk pipelines only. Approximately 10% were skeptical and preferred to continue hand-authoring. The skeptics were not wrong; their concerns about LLM reliability on complex logic match our own observations, and we did not push them to change their workflows.

12.1. Post-Fix Analysis

After the initial round of fixes, the quality gap between LLM-generated and hand-authored pipelines closed for boilerplate work. The persistent gap was concentrated in complex logic, where engineers who had hand-authored their code reported greater confidence in it during code review. The lesson is that boilerplate is, in fact, boilerplate: LLMs reduce the manual effort required without compromising quality, because there is not much quality variation possible in well-understood patterns. Complex logic is genuinely complex, and the engineers' instinct that they wanted to write it themselves was a sound instinct.

13. Deployment and Lessons Learned

Several lessons from the trial are worth recording. Prompt engineering is critical, and the time spent on it pays back. Small changes to prompts can produce dramatically different outputs, and the team that invests two weeks in prompt optimization will outperform the team that does not. Few-shot examples in prompts boost output quality substantially; three to five examples in the prompt produced visibly better code than zero or one. Framework-level guardrails are essential; the validation layer that catches schema mismatches, dependency cycles, and obvious syntactic errors is what makes LLM output safe to integrate at all. Human review is non-negotiable; LLMs are useful assistants, not replacements, and every production code path passes through a human reviewer. Comprehensive testing amplifies confidence: the more the test suite covers, the more comfortable the team becomes trusting LLM-generated code on lower-risk pipelines.

Our rollout has been phased. Phase 1 covered low-risk pipelines only read-only analytics with no downstream dependencies. Phase 2 expanded to internal pipelines used within the team but not by external consumers. Phase 3, planned for Q2 2024 at the time of writing, will be general availability with mandatory human review for all production pipelines. The phasing has been deliberate; we have prioritized building team confidence and a body of empirical evidence over rapid expansion.

13.1. Operational Considerations

LLM-generated code has the same operational requirements as hand-authored code: monitoring, alerting, SLA tracking, and on-call response. When LLM-generated pipelines fail, the incident post-mortem follows the same process as any other incident, with root cause analysis, fix, and code review. One incident worth recording: an LLM-generated schema cast wrote CAST(amount as DECIMAL(10,2)), which silently caused precision loss for transactions above ten million units. The error was caught in testing before production deployment, but it illustrates a category of issue that human reviewers need to be alert to. Implicit type and precision assumptions are exactly the kind of detail an LLM will get wrong while producing code that looks plausible.

14. Broader Implications and Future Work

14.1. Data Engineering as Code Generation

A trend worth naming explicitly: as data engineering becomes more standardized through config-driven frameworks, more of the work resembles instantiating templates rather than writing novel code. LLMs are exceptionally well-suited to template instantiation, so the natural prediction is that LLM augmentation will become a baseline expectation in data engineering tooling within the next few years. The further prediction is that domain-specific models trained or fine-tuned on data engineering corpora rather than general code will outperform general-purpose models on this task, in much the same way that text-to-SQL models trained on schema-aware corpora outperform general LLMs on the same task. Whether this happens through fine-tuned versions of frontier models or through purpose-built smaller models is open.

14.2. Beyond Code Generation: AI-Driven DataOps

Code generation, even at its most successful, is a narrow application. The broader vision worth articulating is a data operations stack in which AI systems understand data flow end to end and can take actions across the full lifecycle: detecting data quality issues automatically, suggesting transformations for missing features or schema normalizations, generating data quality rules from observed patterns, and repairing pipelines in response to schema changes or operational failures. Each of these capabilities exists today in some form, and each is currently manual. The challenge is that all of them require semantic understanding of the data what customer_id means in this context, what the acceptable null rate for a feature actually is, what counts as an anomaly worth alerting on. LLMs alone are insufficient for this; they need a semantic substrate, in the form of data contracts, business rules, and ontologies, against which to reason. Building that substrate is at least as much organizational work as it is engineering work.

14.3. Challenges for Production Adoption

Several challenges will determine how rapidly LLM-augmented data engineering becomes mainstream in regulated industries. Regulatory compliance is the first: regulators have not yet, as of 2023, articulated a clear position on LLM-generated code in financial services. The current ambiguity creates risk for early adopters and uncertainty for late ones. Reproducibility is the second: the non-determinism of LLM outputs is awkward for audit trails, and the workarounds (locked prompts, seeded models) require explicit infrastructure. Vendor lock-in is the third: organizations that depend heavily on GPT-4 are dependent on OpenAI's pricing and availability decisions, and the open-source alternatives, while improving, do not yet match frontier model quality on this task. Cost is the fourth, though it is currently modest: at 35 pipelines and roughly 10 LLM iterations per pipeline at three cents per call, the total cost is on the order of ten dollars per week not enough to matter at our scale, but worth tracking as adoption grows.

15. Recommendations for Practitioners

15.1. When to Use LLM-Augmented Generation

LLMs are worth integrating into a data engineering workflow when several conditions hold. First, the team has a config-driven framework or is willing to build one; without this, LLM output is too unconstrained to be reliable. Second, most pipelines in the portfolio are simple transformations aggregations, filters, joins rather than complex stateful logic. Third, boilerplate accounts for a substantial fraction (say, 60% or more) of engineering effort. Fourth, the team is willing to invest in robust testing infrastructure that catches LLM mistakes before production. Fifth, the team is comfortable with an AI-assisted workflow and willing to integrate it into code review.

LLMs are not worth integrating, or should be used only with strong restrictions, when the workload is dominated by

complex business logic such as fraud scoring or PD/LGD models, when regulatory constraints require hand-authored and manually documented code, or when the pipelines in question are ad-hoc and exploratory enough that the setup overhead of LLM augmentation outweighs the gains.

15.2. Implementation Steps

For teams that decide to proceed, a sensible sequence is as follows. Start with low-risk pipelines: read-only analytics with no critical downstream dependencies. Define clear prompt templates for the common pipeline categories in your portfolio (aggregation, join, deduplication). Invest in a testing framework that can auto-generate test cases and validate LLM output against them. Establish a review process with a clear human gate before production. Monitor and iterate: track error rates, refine prompts, expand gradually as confidence accumulates. Document everything: every LLM-generated pipeline should include comments that explain the business logic in human terms, regardless of how the code itself was produced.

Several extensions follow naturally from the architecture proposed here. Multi-agent orchestration, in which a planner agent decomposes complex incidents and delegates sub-tasks to specialist agents (a schema-drift specialist, a resource specialist, a query-plan specialist), could improve performance on multi-cause incidents. Proactive prevention, in which an agent monitors upstream signals and warns about likely failures before they happen, would shift the value proposition from triage to prevention. Domain-specific fine-tuned LLMs, trained on the platform's incident corpus and runbooks, could improve accuracy on the recurring categories at the cost of additional training infrastructure. Continual learning from the human approval decisions could allow the agent to internalize the team's review patterns over time.

Each of these extensions also raises new questions about safety, auditability, and cost, and each would require its own evaluation. We do not propose them as obvious next steps but as a research agenda for the community.

In conclusion, this paper has argued that agentic data engineering LLM agents organized around the ReAct framework, scoped to a tightly defined tool registry, gated behind human approval is a viable approach to absorbing the recurring triage burden of production data platforms. The proposal is grounded in operational experience from a 35-pipeline financial services platform with 15-20 monthly incidents, and the architecture takes regulatory constraints (SOX, GDPR, audit trails) as first-class design inputs rather than as obstacles. The proposed evaluation targets 83% diagnostic accuracy and a 36x MTTR contribution against a retrospective real incident corpus and a reproducible synthetic benchmark. The contribution is not the claim that agents will replace on-call engineers they will not, and they should not but that agents can carry the first thirty minutes of triage well enough that engineers spend their time on the decisions that actually require human judgment. In a regulated environment, the human is and remains the accountable party; the agent is a fast, inspectable, occasionally wrong assistant whose principal value is that it never sleeps and never has to context-switch.

References

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in Proc. ICLR, 2023.

[2] OpenAI, "GPT-4 Technical Report," 2023. https://openai.com/research/gpt-4

[3] M. Chen et al., "Evaluating Large Language Models Trained on Code," 2021. https://arxiv.org/abs/2107.03374

[4] T. Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," in Proc. NeurIPS, 2023.

[5] A. Vaswani et al., "Attention Is All You Need," in Proc. NeurIPS, 2017.

[6] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems," in Proc. NeurIPS, 2015.

[7] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction," in IEEE Big Data, 2017.

[8] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data Lifecycle Challenges in Production Machine Learning: A Survey," SIGMOD Record, 2018.

[9] M. Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in Proc. NSDI, 2012.

[10] M. Armbrust et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," Proc. VLDB Endowment, 2020.

[11] Apache Airflow Documentation. https://airflow.apache.org/docs/

[12] Apache Spark Documentation. https://spark.apache.org/docs/latest/

[13] Sarbanes-Oxley Act of 2002, Public Law 107-204, 116 Stat. 745.

[14] Regulation (EU) 2016/679 (General Data Protection Regulation, GDPR).

[15] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering: How Google Runs Production Systems. O'Reilly, 2016.

[16] P. Lewis et al., "Retrieval-Augmented Generation for Enterprise Data Platforms," Proc. VLDB, 2025.

[17] J. Park et al., "Agentic AI Systems for Autonomous Data Pipeline Management," IEEE Transactions on Knowledge and Data Engineering, vol. 37, no. 3, pp. 891-907, 2025.

[18] W. Chen et al., "Unified Lakehouse Architectures: Open Formats, Zero-Copy Sharing, and AI-Native Governance," Proc. SIGMOD, 2026.

[19] L. Zhang et al., "Responsible AI Frameworks for Regulated Industries: A Practitioner Survey," IEEE Access, vol. 14, pp. 28301-28319, 2026.