POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering



Master's Degree Thesis

DataOps Pipelines and LLM-Driven Self-Healing: A Reference Architecture for Resilient Data Operations

Supervisor

Prof. PAOLO GARZA

Candidate

FEDERICO BALDI

Academic Year 2025/2026

Abstract

The exponential growth in enterprise data volume and complexity has highlighted the limits of monolithic data architectures, leading to the adoption of decentralized paradigms such as Data Mesh and DataOps engineering methodologies. Although these approaches aim to resolve issues in data team workflows, their implementation in production environments remains inconsistently handled, particularly regarding data quality assurance, pipeline observability and controlled error management across the ETL/ELT lifecycle, leading to extended Mean Time to Recovery (MTTR).

The first stage of this thesis is the design, implementation and validation of a DataOps-oriented data pipeline, built on the European Soccer Database as an experimental basis. Structured according to Domain-Driven Design principles within a Medallion Architecture, and fully implemented as Infrastructure-as-Code (IaC) via Databricks Asset Bundles (DABs) and Data Build Tool (`dbt`), the pipeline integrates automated data quality testing against core KPIs—completeness, validity, freshness, and consistency—alongside systematic observability of job performance, execution failures and anomaly detection. Clear separation of development, testing, and production environments, combined with structured pre-release validation stages, ensures data issues are intercepted before propagating downstream, establishing a resilient and scalable foundation for automated data operations.

Building upon this foundation, the thesis explores a self-healing extension natively embedded within the pipeline orchestration workflow. Triggered autonomously upon job failures, the agent diagnoses the root cause of the detected anomaly, derives candidate technical corrections through an LLM-assisted reasoning process, and validates each solution through controlled execution within an isolated Sandbox environment—ensuring that only verified patches are submitted as GitLab Merge Requests for human review, thereby preserving Human-in-the-Loop oversight before any correction is adopted in production.

The framework was empirically evaluated using a structured benchmark of different failure scenarios—spanning SQL errors, cascading schema drift and governance policy violations—comparing multiple LLMs across iterative refinements of the agent's architectural guardrails. Results demonstrate that embedding structured

data quality testing, observability and LLM-assisted self-healing directly into the DataOps lifecycle substantially reduces manual effort and improves pipeline resilience, offering a grounded reference architecture for adoption in modern enterprise contexts.

Acknowledgements

ACKNOWLEDGMENTS

A mamma e papà,
ai vostri sacrifici e ai valori che mi avete trasmesso.

Ad Alessia,
per il bene incondizionato che sempre mi dimostra.

A Sara,
l'amore della mia vita.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XI

1 Introduction 1

2 DataOps and Data Mesh 5
- 2.1 From Traditional Data Management to DataOps . . . 5
- 2.2 From Data Warehouse to Lakehouse . . . 8
- 2.3 The Data Mesh Paradigm . . . 10
  - 2.3.1 The Limitations of Centralized Platforms . . . 11
  - 2.3.2 The Four Principles of Data Mesh . . . 11

3 Enabling Technologies and Core Infrastructure 16
- 3.1 Cloud Storage and the Medallion Architecture . . . 16
  - 3.1.1 Mono-Zone Architecture . . . 16
  - 3.1.2 Zone-Based Architectures and Medallion Layers . . . 16
  - 3.1.3 Data Lakehouse Architecture . . . 18
  - 3.1.4 Transactional Storage with Delta Lake . . . 19
- 3.2 The Correct-by-Design Lakehouse . . . 21
- 3.3 The Transformation Framework: Data Build Tool (dbt) . . . 23
  - 3.3.1 Software Engineering Principles in Analytics . . . 24
  - 3.3.2 Layered Data Modeling with dbt . . . 25
- 3.4 Infrastructure as Code: Databricks Asset Bundles (DABs) . . . 25
  - 3.4.1 Environment Isolation and CI/CD Pipelines . . . 26
- 3.5 Agentic AI in Data Operations . . . 27
  - 3.5.1 Evolution from DataOps to AIOps . . . 28
  - 3.5.2 Failure Taxonomy and Multi-Phase Diagnosis Workflow . . . 28

3.5.3 Agentic Reasoning and the ReAct Framework 29
3.5.4 Cognitive Architecture of LLM-Based Agents 30
3.5.5 Bounded Autonomy and Governance Guardrails 31

4 Architectural Design: The Soccer Domain 32
4.1 The European Soccer Database 32
4.1.1 Database Structure 33
4.1.2 Dataset Coverage and Sample Records 34
4.1.3 Dataset Role in the Ingestion Pipeline 34
4.2 Domain Decomposition into Bounded Contexts 35
4.2.1 Methodology and Criteria 36
4.2.2 Source-Aligned: Competitions Domain 37
4.2.3 Source-Aligned: Roster Domain 38
4.2.4 Source-Aligned: Match Domain 38
4.2.5 Consumer-Aligned: BI Analytics Domain 38
4.3 DataOps Orchestration via Databricks Asset Bundles 39
4.3.1 Job Centralization and Topology 39
4.4 Self-Healing LLM Agent Architecture 41
4.4.1 In-Job vs. Out-Job Approach 42
4.4.2 Profiling and Memory Modules 42
4.4.3 Planning Module 43
4.4.4 Action Module 45

5 DataOps Implementation and Empirical Evaluation 46
5.1 DataOps Infrastructure Implementation 46
5.1.1 Physical Implementation of the Medallion Architecture . . . 47
5.1.2 Declarative Infrastructure Governance 48
5.1.3 Data Contracts and Observability Integration 51
5.2 Self-Healing Agent Implementation 52
5.2.1 In-Job Activation and Artifact Access 53
5.2.2 LangGraph Cognitive Engine 54
5.2.3 Architectural Evolution: v1.0 vs. v1.1 55
5.3 Experimental Setup and Benchmark Design 58
5.3.1 Benchmark Taxonomy and Failure Scenarios 58
5.3.2 Candidate Language Models 58
5.3.3 Evaluation Metrics 60
5.4 Quantitative and Qualitative Benchmark Results 61
5.4.1 Quantitative Performance Metrics 61
5.4.2 Discussion and Qualitative Analysis 63

6 Conclusions and Future Work 65
6.1 Main Contributions 65
6.2 Empirical Key Takeaways 66
6.3 Limitations and Future Work 67
6.4 Final Remarks 67

A Full Per-Scenario Benchmark Results 68
A.1 Claude Opus 4.8 Benchmark Results 69
A.2 Claude Haiku 4.5 Benchmark Results 70
A.3 Qwen3.5 122B A10B Benchmark Results 71

B Case Study: Multi-File Jinja Refactoring 72
B.1 Scenario Description and Pre-Remediation Baseline 72
B.2 Baseline Framework (v1.0) Execution 73
B.3 Guardrailed Framework (v1.1) Execution 74
B.4 Comparative Execution Summary 75

Bibliography 76

List of Tables

3.1 Data Pipeline Failure Taxonomy. 29

4.1 Main entities of the European Soccer Database. 34
4.2 Leagues and countries covered by the European Soccer Database. . 35
4.3 Sample records from the `Match` table (Serie A, 2010/2011). 36
4.4 Soccer Database entities mapped to Data Mesh bounded contexts. . 37
4.5 Failure taxonomy for DataOps pipelines (dbt-specific). 44

5.1 Environment Segregation Topology in Databricks Unity Catalog. . . 49
5.2 Action Toolset Scoping and Execution Contract. 56
5.3 Benchmark Test Suite Taxonomy across 15 Failure Scenarios. 59
5.4 Benchmark Performance Summary per Model (v1.0 vs. v1.1). 62

A.1 Claude Opus 4.8 — Baseline Version v1.0. 69
A.2 Claude Opus 4.8 — Guardrailed Version v1.1. 69
A.3 Claude Haiku 4.5 — Baseline Version v1.0. 70
A.4 Claude Haiku 4.5 — Guardrailed Version v1.1. 70
A.5 Qwen3.5 122B A10B — Baseline Version v1.0. 71
A.6 Qwen3.5 122B A10B — Guardrailed Version v1.1. 71

B.1 Comparative Execution Metrics for Scenario `L6-B` under Claude Haiku 4.5. 75

List of Figures

1.1 Evolution of data platform architectures. 2
1.2 The Data Engineering Lifecycle and its Undercurrents. 3

2.1 Comparison between the DataOps and DevOps processes. 6
2.2 The Value-Innovation Pipeline in the DataOps framework. 7
2.3 Comparison between traditional ETL and modern ELT architectures. 9
2.4 Differences between Data Lake and Data Mesh. 10
2.5 Four Principles of Data Mesh. 12

3.1 Mono-zone architecture for data lakes. 17
3.2 Multi-zone functional architecture for data lakes. 18
3.3 Medallion Architecture: Layered Data Refinement Model. 19
3.4 Data Lakehouse architecture. 19
3.5 Delta Lake: unified stream and table storage pipeline. 20
3.6 Git-for-data: point-in-time debugging and branching. 23
3.7 Transactional pipelines: atomicity and isolation guarantees. 23
3.8 The role of dbt in the modern data stack as a transformation layer. 24
3.9 CI/CD pipeline with Declarative Automation Bundles. 27
3.10 System architecture of the diagnosis and remediation framework for ETL pipelines. 30
3.11 The ReAct loop: Thought, Action, and Observation. 30

4.1 Databricks Job DAG with Self-Healing Agent triggering. 41
4.2 Cognitive architecture of the Self-Healing AI Agent. 45

Acronyms

ETL
: Extract, Transform, and Load

ELT
: Extract, Load, and Transform

CI/CD
: Continuous Integration and Continuous Deployment

ACID
: Atomicity, Consistency, Isolation, Durability

DML
: Data Manipulation Language

DAG
: Directed Acyclic Graph

SLO
: Service Level Objective

SLA
: Service Level Agreement

DDD
: Domain-Driven Design

SCD
: Slowly Changing Dimensions

OBT

One Big Table

OLTP

Online Transaction Processing

SoR

System of Record

KPI

Key Performance Indicator

IaC

Infrastructure as Code

DAB

Databricks Asset Bundles

AIOps

Artificial Intelligence for IT Operations

LLM

Large Language Model

ReAct

Reasoning and Acting

HITL

Human-in-the-Loop

MTTR

Mean Time to Recovery

ACI

Agent-Computer Interface

RCRR

Root Cause Resolution Rate

MR CR

Merge Request Creation Rate

DRY

Don't Repeat Yourself

Chapter 1

Introduction

Data pipelines have become fundamental components of modern enterprise data infrastructure. Defined as automated processes for collecting, transforming, and delivering data from heterogeneous source systems to a destination—such as a data warehouse or data lake—they serve as the mechanisms for the flow and access of information. These systems ingest raw data and yield high-quality, consistent information that supports downstream use cases ranging from business intelligence dashboards to machine learning applications. As organizations increasingly rely on data to guide strategic decisions, the reliability and correctness of these pipelines directly determine the quality of insights produced for further analysis [1].

Historically, the organization's data infrastructure has undergone a significant architectural evolution: traditional data warehouses, while effective for business intelligence, mandated rigid schema-on-write paradigms that constrained their ability to support advanced analytics and machine learning workloads [2]. The subsequent adoption of data lakes introduced flexibility through schema-on-read approaches, but these repositories often lacked the transactional integrity and governance capabilities of warehouses. The Lakehouse architecture emerged to reconcile these trade-offs, combining the flexibility of data lakes with the management features of data warehouses within a unified platform, as illustrated in Figure 1.1 [3]. The architectural details and technical foundations of this evolution are discussed in Chapter 2 and Chapter 3.

Despite the architectural advancements of the Lakehouse, modern data pipelines exhibit a high degree of operational fragility. This vulnerability is often manifested as **data downtime**, defined as periods of time when data is partial, erroneous, missing or otherwise inaccurate [5]. A main driver of this instability is **schema drift**: the unexpected alteration of structural metadata, such as renamed tables or altered column types, in the source systems [5]. These changes often result in undetected errors—where the pipeline continues to process data but produces incorrect or misleading results, in a "garbage in, garbage out" dynamic—leading

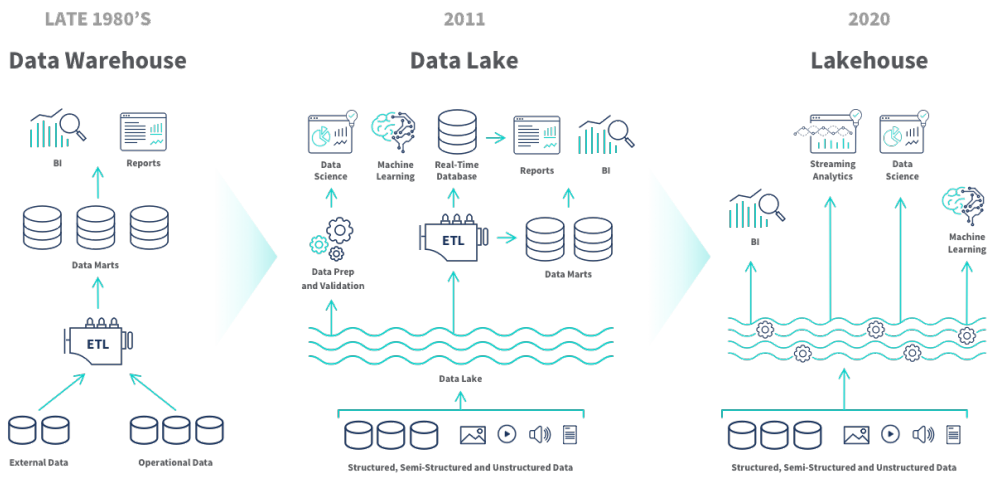


Figure 1.1: Evolution of data platform architectures: from data warehouses and data lakes to the Lakehouse paradigm [4].

to significant information loss as errors propagate through the data lifecycle [6]. Consequently, maintaining the reliability of the flow requires constant monitoring of key data quality dimensions. The academic literature identifies a broad taxonomy of such dimensions [7]; among the most relevant for the scope of this work are:

- **Accuracy**: the adherence to a given reality of interest, encompassing correctness, validity, and precision of the represented information.
- **Completeness**: the capability of representing all and only the relevant aspects of the reality of interest.
- **Consistency**: the capability of the information to comply without contradictions to all properties of the reality of interest, as specified in terms of integrity constraints, data edits, and business rules.
- **Trust**: including credibility, reliability, and reputation, catching how much information derives from an authoritative source. The trust cluster encompasses also issues related to security.

To mitigate the inherent fragility of modern data pipelines and guarantee the continuous delivery of high-quality insights, organizations must transition from ad-hoc, manual data management practices toward a rigorous, engineering-led methodology. This shift is embodied by **DataOps**, a process-oriented methodology derived from DevOps, Agile development, and continuous integration/continuous

delivery practices, aimed at the fast delivery of high-quality insights through improved collaboration, integration, and automation of data flows between data managers and consumers [8]. The theoretical foundations, core principles, and operational benefits of DataOps are discussed in detail in Chapter 2.

Consequently, data engineering has evolved to bridge data management, platform security, DataOps principles, architecture, orchestration, and core software engineering standards. This intersection implies a fundamental "Software Engineering for Data" approach, where pipelines are no longer viewed as static scripts but as living software products. This requires the implementation of an automated Data Engineering Lifecycle that manages data from generation and ingestion, through transformation, to serving—ensuring that the flow and access of information remain resilient even when source systems undergo unpredictable schema drift [1] (illustrated in Figure 1.2).

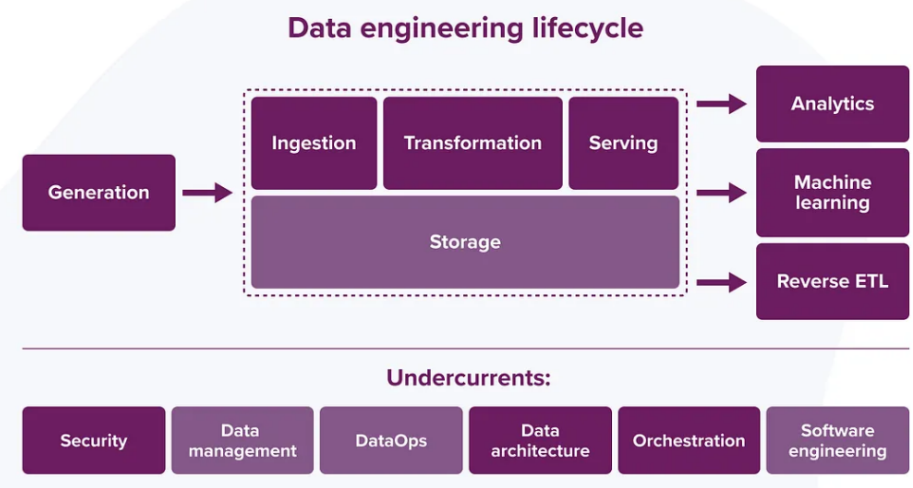


Figure 1.2: The Data Engineering Lifecycle and its Undercurrents [1].

Despite the increasing maturity of Lakehouse architectures and the growing adoption of DataOps practices, modern data pipelines remain inherently fragile. Schema drift continues to be a primary driver of silent pipeline failures, where incorrect data propagates undetected through transformation layers. Current approaches to mitigating these issues are predominantly reactive: errors are discovered only after they have corrupted downstream analytics, requiring costly manual remediation. This thesis addresses the following research question:

> *How can a DataOps-oriented data platform be designed and implemented to proactively detect, prevent, and automatically remediate pipeline failures caused by schema drift and data quality violations, while ensuring the continuous delivery of trusted, analytics-ready data products?*

This central question can be decomposed into the following sub-questions:

1. How can DataOps principles—specifically automation, CI/CD for data, and quality-by-design—be operationalized within a Lakehouse architecture to minimize data downtime?

2. How can data contracts and automated testing frameworks be integrated into the transformation layer to enforce data quality at the point of creation, following a *shift-left* approach?

3. What architectural patterns enable the construction of self-healing data pipelines capable of detecting schema drift, analyzing its root cause, and proposing corrective actions in an assisted manner?

The primary objective of this work is to design and prototype a complete, DataOps-oriented data pipeline that embodies the principles of automation, observability, and resilience across the entire data engineering lifecycle.

Chapter 2

DataOps and Data Mesh

2.1 From Traditional Data Management to DataOps

In the contemporary software development landscape, organizations increasingly recognize data as a strategic asset rather than merely an operational byproduct [9]. This paradigm shift is driven by unprecedented digital growth: global data generation, consumption, and storage continue to expand rapidly. While worldwide volume reached an estimated 173.4 zettabytes in 2025, projections indicate it will surge to 527.5 zettabytes by 2029 [10].

This massive influx has exposed the critical vulnerabilities of legacy infrastructure: traditional data management systems are primarily limited by predefined schemas and inflexible data models, often restricting the organization's ability to process heterogeneous data types [11]. Historically, data management relied heavily on centralized **data warehouse** architectures and monolithic **Extract, Transform, and Load (ETL)** pipelines, which introduced substantial delays between ingestion and utilization. To overcome these bottlenecks and properly manage the data lifecycle, **data engineering pipelines** emerged as structured workflows designed to collect, refine, and serve data across operational and analytical workloads. By automating stages of transformation, these pipelines convert raw ingestion data into standardized models, supplying downstream applications such as predictive modeling, executive dashboards, and business intelligence suites [12]. To bridge the historical divide between data management workflows and core software engineering, the **DataOps** methodology has emerged [9]. DataOps establishes an operational paradigm that combines agile software practices with data pipeline automation, treating data assets with the same rigor, CI/CD discipline, and lifecycle governance applied to application code. By extending the collaborative principles of **DevOps** to data-centric operations, this methodology establishes a cohesive ecosystem where

data workflows are managed with the identical engineering standards and automated testing used for software releases. The adoption of DataOps has transitioned into a business necessity: recent industry research indicates that organizations implementing these methodologies report up to 70% faster delivery cycles alongside a 30% drop in data-related defects [9]. Precisely because of the massive volume, the inherent flexibility, and the *schema-on-read* nature of modern data architectures, managing these environments manually is highly prone to errors and bottlenecks. The conceptual framework of DataOps is therefore essential to operationalize and govern this infrastructure. While sharing a significant philosophical alignment with DevOps—inheriting its emphasis on collaboration, automation, and continuous improvement—DataOps is uniquely tailored to manage data's inherent variability, volume, and downstream dependencies.

Historically, DataOps emerged from the realization that traditional data management approaches were insufficient to handle the unique volatility of modern data landscapes. While DevOps revolutionized software development by aligning development and operations through Continuous Integration and Continuous Deployment (CI/CD) [13], DataOps extends these principles by merging process-driven data governance with agile standards and end-to-end automation, aiming to accelerate delivery throughput, elevate asset quality, and promote continuous organizational refinement [14]. A key distinguishing element of DataOps is the application of Statistical Process Control, borrowed from Total Quality Management, which verifies that results at each intermediate step in the production of analytics match business requirements, as compared to traditional DevOps in Figure 2.1 [15].

To identify waste and manage the flow of data, lean thinking enables the delivery of analytical solutions as quickly as possible, much like an automated assembly line [16].

The DataOps framework operationalizes this approach through two interconnected pipelines:

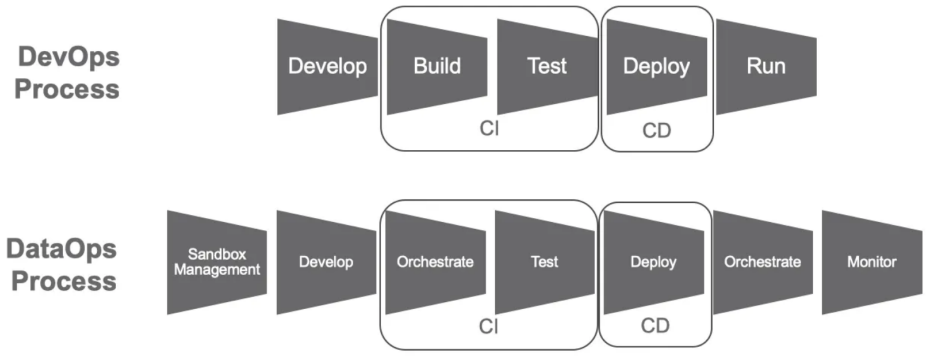


Figure 2.1: Comparison between the DataOps and DevOps processes [15].

- **Value Pipeline**: governs the continuous flow of data through production.
- **Innovation Pipeline**: manages the deployment of new analytical features.

These two work in concert as the **Value-Innovation Pipeline** (Figure 2.2) to break down the barrier between development and production. Automated testing mechanisms, inspired by Statistical Process Control, ensure quality is built into both pipelines, enabling the data engineering team to detect anomalies and deploy changes with confidence [15].

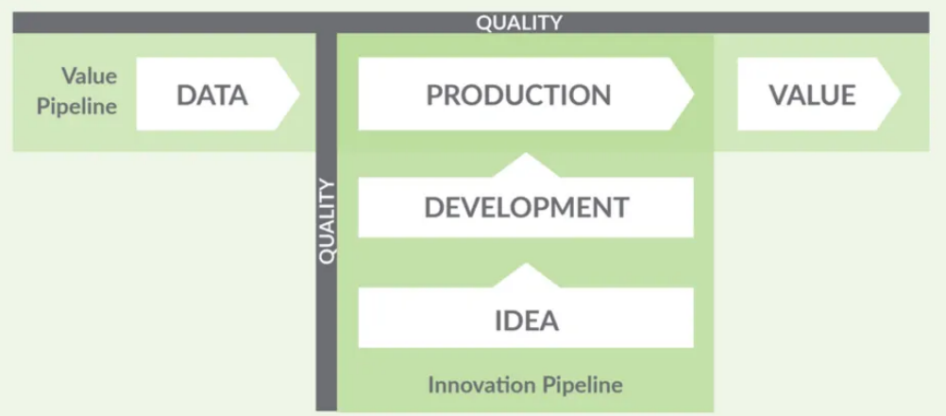


Figure 2.2: The Value-Innovation Pipeline: integration of the Value and Innovation pipelines in the DataOps framework [15].

The core principles driving this methodology include [9]:

- **Automation and Orchestration:** The methodology emphasizes automating recurrent tasks throughout the data lifecycle, such as ingestion, transformation, validation, and deployment.
- **Continuous Integration/Continuous Deployment (CI/CD) for Data:** Adapting software CI/CD practices, this principle involves frequently integrating data changes, automatically testing their validity and impact, and deploying them to production environments. To handle data safely, specific CI/CD pipelines often incorporate the *expand-contract* approach, enabling non-breaking schema changes.
- **Version Control and Reproducibility:** DataOps extends traditional version control mechanisms to guarantee reproducibility. Beyond application code, this approach enforces versioning across data schemas, transformation logic, and analytical models, allowing engineering teams to reconstruct any prior state of the data ecosystem.

- **Quality by Design:** Shifting away from reactive troubleshooting, DataOps embeds quality validation across data pipelines, implementing automated testing and monitoring to flag anomalies before impacting downstream consumers.

The successful implementation of DataOps yields measurable economic and operational benefits. Organizations evaluate these initiatives using multidimensional indicators, balancing technical metrics with business outcomes, which include pipeline performance metrics, data utilization metrics, and business impact metrics. From an operational efficiency standpoint, DataOps drives improvement through process automation—reducing manual effort by 40–60% for routine tasks—and error reduction by following the *shift-left* principle to minimize costly remediations. Ultimately, the framework facilitates strategic resource allocation by improving skill utilization, reducing coordination overhead, and shifting focus toward higher-value analytical activities [9].

2.2 From Data Warehouse to Lakehouse

The successful implementation of data-driven methodologies requires an underlying architecture capable of handling modern analytical demands. To overcome the limitations of legacy systems, the concept of the **data lake** has been introduced [11]. The term originated with James Dixon (former CTO of Pentaho) as a remedy for the constraints of **data marts**—specialized subsets of data warehouses inherently restricted to predefined analytical queries [17]. Architecturally, data lakes aggregate multi-source records into a centralized storage repository, effectively mitigating the isolation and inconsistencies typical of data silos while facilitating standardized governance across the enterprise [18].

A fundamental difference between these new architectures and traditional systems lies in how data is ingested and structured. Traditional data warehouses are based on a *schema-on-write* strategy, where a schema-first approach aims to optimize queries and ensure data consistency prior to storage [19]. In contrast, data lakes function as centralized repositories retaining heterogeneous data formats without enforcing rigid structural constraints at ingestion time. This operational model relies on two distinct properties of data lakes: handling broad *data variety* and adopting a *schema-on-read* paradigm (known as "late binding"), where data interpretation and schema modeling are deferred until analytical queries are executed [20]. Because of this flexibility, they implement an **Extract, Load, and Transform (ELT)** pipeline, illustrated in Figure 2.3, landing data in its native form before executing transformations on demand. This approach grants enterprises the freedom to deploy different processing techniques and analytics frameworks across the stored data without enforcing global data transformations [21].

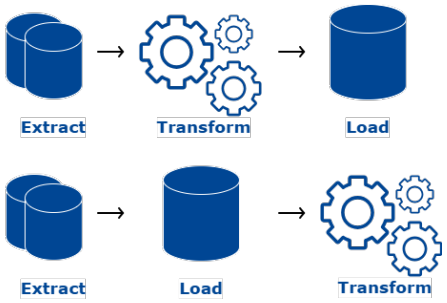


Figure 2.3: Comparison between traditional ETL and modern ELT architectures.

However, storing raw data without predefined structures presents significant risks. To prevent these vast repositories from degrading into unmanageable data swamps, academic literature emphasizes that a modern data lake is not merely a passive storage drive, but a complex, governed system. The most elaborate definition describes it as a distributed platform for storing and analyzing heterogeneous data, preserved in its native state and accessed primarily by data specialists. The data-lake properties include [11]:

- A metadata catalogue that imposes data quality.
- Data governance policies and tools.
- Availability to different kinds of users.
- Integration of any kind of data.
- A logical and physical organization.
- Scalability in terms of storage and processing.

To systematically enforce these properties and definitively resolve the trade-offs between data lakes and data warehouses, the **Lakehouse** architecture was introduced. According to Armbrust et al. [3], a Lakehouse is a data management system that combines low-overhead, directly accessible storage with core analytical DBMS features, including ACID guarantees, versioning mechanisms, auditing, indexing, caching, and query optimization. Thus, Lakehouses merge the key strengths of data lakes and data warehouses: low-cost storage in an open format accessible by a variety of systems from the former, and robust management and optimization capabilities from the latter.

Furthermore, according to Sheng et al. [22], the data lakehouse serves as the *de facto* cloud paradigm for analytics and AI workloads, combining object storage with open table formats and decoupled, multi-language engines. Nevertheless, lakehouses

represent the default substrate for analytics and AI, yet they remain susceptible to concurrent, untrusted modifications: schema mismatches often surface only at execution time, development and production easily drift apart, and multi-table pipelines can leak incomplete records following a failure.

2.3 The Data Mesh Paradigm

The proliferation of data expectations and use cases, and the availability of data from ubiquitous sources combined with business complexity have pushed organizations to an inflection point [23]. **Data Mesh** emerged to counteract centralized, monolithic data management paradigms that have long served as industry standards, such as data warehouses and data lakes [24]. Under these conventional setups, domain teams rely on a centralized unit to collect, process, and refine domain data; however, this group increasingly acts as a bottleneck for unlocking the value from their analytical assets [24]. This team is often composed of specialized data engineers isolated from functional business units where data is generated and consumed, creating a group devoid of business and domain knowledge [23]. Consequently, Data mesh represents an emerging, domain-oriented distributed data architecture designed to mitigate the operational bottlenecks inherent to centralized, monolithic frameworks [24]. It constitutes a socio-technical approach aimed at transferring domain data ownership away from the central team to domain-specific units [23, 24], as illustrated in Figure 2.4.

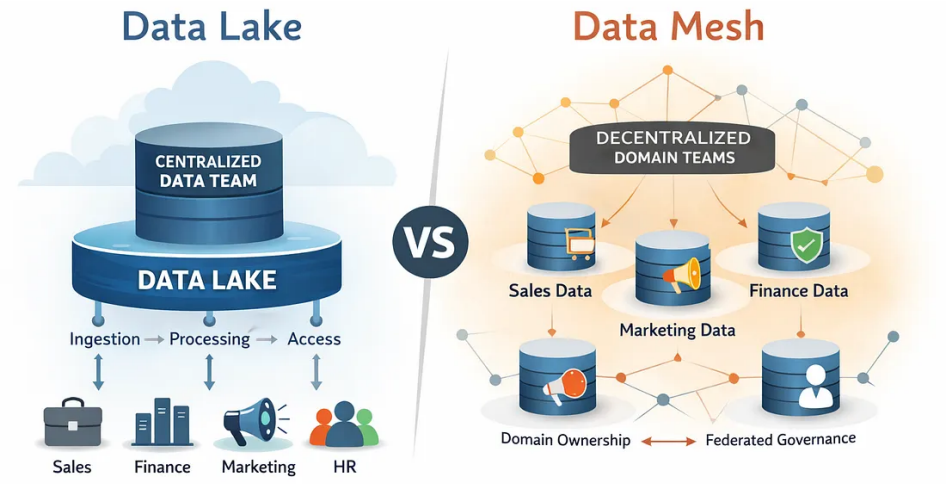


Figure 2.4: Differences between Data Lake and Data Mesh [25].

2.3.1 The Limitations of Centralized Platforms

According to the literature, there are three primary challenges that highlight the limitations of centralized architectures [24]:

- **Scaling with Data Sources:** As the number of data sources expands within an organization, it becomes increasingly challenging to maintain visibility across disparate feeds and guarantee their high-quality delivery. Despite the widespread adoption of data lakes, enterprises continue to face obstacles in discovering and exposing internal data assets across the entire organization, making various data sources interoperable, and establishing data quality and consistency at the source. In fact, centralized data lakes can quickly become data swamps without adequate data ownership and governance models.
- **Central Teams as a Bottleneck:** The counterpart to the continuous inflow of data sources is the parallel rise in data-driven use cases across enterprises. The data for these applications is usually supplied by a centralized team of specialized data engineers who also maintain the monolithic data platforms in which the data is aggregated. Consequently, relying on centralized units compromises delivery speed and architectural flexibility, which are essential to satisfy the diverse needs of downstream consumers.
- **Separation of Provider and Consumer Expertise:** In most organizations, data provider knowledge remains detached from data consumer expertise. Relying on centralized teams to understand both the context of the source domain and the consuming domain puts additional stress on the operational bottleneck, whereas the data provider and data consumer naturally possess deep domain-specific knowledge.

2.3.2 The Four Principles of Data Mesh

The Data Mesh paradigm is formalized through four core architectural principles, as illustrated in Figure 2.5:

Domain Ownership of Data

The first principle of the data mesh paradigm focuses on the decentralization of data ownership by transferring data accountability to the teams or specialists closest to the source who understand the data intimately [24]. This principle advocates applying the Domain-Driven Design approach to partition enterprise analytical data into data domains that mirror specific business units [24]. Consequently, organizations must shift the ownership of sharing analytical data to business domains situated nearest to it, serving either as its originating point or its primary

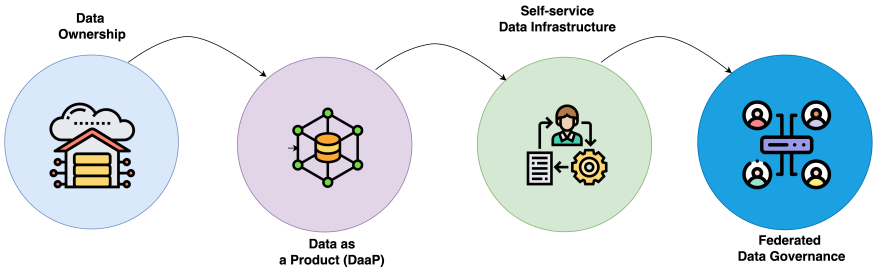


Figure 2.5: Four Principles of Data Mesh [26].

consumers [23]. Autonomous domain teams bear the responsibility of creating and sustaining data products from domain data [24]. This is essential because these units possess deep knowledge about their assets, including potential use cases and target data consumers, as well as the processes and methodologies for maintaining data quality [24]. Finally, this approach allows organizations to decompose the data artifacts, such as data, code, metadata, and policies, according to the business domain they serve, while supervising their entire lifecycle independently, essentially aligning business, technology and analytical data [23].

Data As a Product

The second principle extends product thinking to analytical data, treating it as a valuable asset for potential data consumers [24]. Enterprise data assets are packaged into products that enable users to explore and leverage domain data with full autonomy [24]. Dehghani [23] defines a data product as an *architectural quantum*, conceptualized as an independently deployable, highly cohesive component encompassing all the structural elements required for its function [24]. Accordingly, a data product encapsulates three fundamental components: data and metadata; pipelines for ingesting, processing, serving, and governing data; and infrastructure for building, deploying, and executing code while storing data and metadata [24]. This shifts the focus from simply exposing data to ensuring that data consumers can easily discover, understand, securely access, and trust the data [23]. To achieve this, Dehghani [23] introduced eight properties for a data product:

- **Discoverable:** Consumers must be able to easily find, explore, and gain confidence in the available data sets. Instead of relying on a traditional centralized catalog curated after data generation, Data Mesh embraces a source-oriented solution where the data product itself intentionally provides its discoverability information (e.g., origin, owners, quality metrics, and sample datasets) in a standard way across its lifecycle.

- **Understandable:** Once discovered, the data product must be easily understood in a self-serve manner without requiring user hand-holding. This requires providing a formal representation of the data's semantics and syntax, accompanied by sample datasets and computational notebooks that visually demonstrate how to query and use the data.
- **Trustworthy and Truthful:** A data product must confidently represent the facts of the business, closing the gap between knowns and unknowns for the user. Owners must guarantee an acceptable level of quality by communicating and monitoring Service Level Objectives (SLOs), such as timeliness, completeness, interval of change, statistical shape, and lineage. This requires shifting automated data integrity testing to the point of data creation.
- **Addressable:** It must offer a permanent, unique global address that allows users to access it programmatically or manually. This addressing system must accommodate continuous changes—such as schema evolution, new data time slices, or updated SLOs—while ensuring continuity of usage through an addressable aggregate root.
- **Interoperable and Composable:** To allow users to correlate and stitch data across different domains, data products must adhere to global standards and harmonization rules. This includes standardizing field types, global addresses, schema linking, and the management of polysemy identifiers (entities that cross multiple domain boundaries).
- **Secure:** Access control must be validated dynamically by the data product itself, directly in the flow of data. Security policies regarding access, encryption, confidentiality levels, and data retention must be implemented as code (policy-as-code), allowing them to be versioned, computationally tested, and enforced at runtime.
- **Natively Accessible:** The product must cater to a wide spectrum of data users (e.g., data analysts, data scientists, and developers) by allowing them to fetch data using their native tools. For instance, it should natively support access via SQL queries for analysts as well as real-time event streams for application developers.
- **Valuable on Its Own:** A data product must carry inherent business value independently, without strictly requiring correlation with other data products. Unlike traditional data warehouse modeling, which often uses "glue tables" to mechanically facilitate joins, a Data Mesh does not utilize data products that lack independent semantic value.

Self-Serve Data Platform

Data mesh promotes the development of domain-agnostic infrastructure and platform services, delivering them in a self-service manner to support the various stakeholders involved in designing, deploying, and maintaining data products [24]. This principle requires an advanced self-serve data platform designed to enable domain-oriented teams to manage the end-to-end lifecycle of their data products, sustain a reliable mesh of interconnected nodes, and simplify the consumer journey for discovering, accessing, and leveraging these assets [23]. According to Dehghani [23], the self-serve platform focuses on enhancing the productivity of data mesh stakeholders across their primary activities, namely building, utilizing, and supervising data products [24]. This centralized platform reduces the cognitive load on domain developers as they do not need deep knowledge of managing infrastructure and platform services [24]. A dedicated team of systems and cloud engineers develops and operates the platform, collaborating directly with business domains to gather and prioritize functional needs [24]. Ultimately, Dehghani [23] proposes an architectural division of the platform into three distinct layers: the data infrastructure utility plane, the data product experience plane, and the mesh experience plane, designed to encapsulate the capabilities to manage infrastructure, build data products, and govern the data mesh [24].

Federated Computational Governance

To guarantee that the data products remain interoperable while adhering to internal standards and external regulatory mandates, a federated governance team is tasked with establishing and enforcing product interoperability standards and compliance policies throughout the ecosystem [24]. Data mesh relies on a decentralized governance model where autonomous units oversee their data products, while overarching governance on a global level primarily aims to guarantee compatibility across data products and to uphold organization-wide rules [24]. According to Dehghani [23], this consists of an operating framework for data governance grounded in federated decision-making and accountability, driven by a collective comprising domain representatives, data platform specialists, and subject matter experts. This model establishes a system of incentives and responsibilities that balances the autonomy and agility of domains with the need for mesh-wide conformance, interoperability and security. Automated execution serves as a vital component within the federated computational governance model, guaranteeing its viable scalability across the entire organization [24]. The governance model fundamentally depends on embedding policies into code and enforcing them automatically at a fine-grained level for each data product [23]. This automation is crucial for managing the data governance challenges arising from frequent regulatory changes and the growth of product consumers and data sources [24].

In conclusion, while DataOps provides the rigorous engineering practices required for reliable data pipelines, Data Mesh introduces the decentralized sociotechnical architecture necessary to scale them. Together, these two paradigms form the theoretical foundation for the architectural implementation that will be discussed in the next chapter, where a concrete, scalable, and governed data platform will be proposed.

Chapter 3

Enabling Technologies and Core Infrastructure

3.1 Cloud Storage and the Medallion Architecture

To operationalize these principles, DataOps relies on robust technological infrastructure, with pipeline automation as a fundamental component that transforms manual processes into streamlined workflows. Modern implementations heavily leverage containerization technologies for consistent environments and orchestration frameworks alongside specialized tools for data transformation and stream processing [9].

The underlying data architecture supporting these pipelines has evolved significantly to accommodate growing analytical demands and governance requirements.

3.1.1 Mono-Zone Architecture

The earliest data lake architecture adopted a simple, flat design based on a single zone, in which incoming data are stored in their original representation [11], as shown in Figure 3.1. While cost-effective, this foundational model provides no native mechanisms for in-place data transformation, and it fails to track or log any user actions [27].

3.1.2 Zone-Based Architectures and Medallion Layers

To address the chaos of flat data lakes, architectural patterns evolved into zone architectures, which organize data into zones based on its level of refinement. A

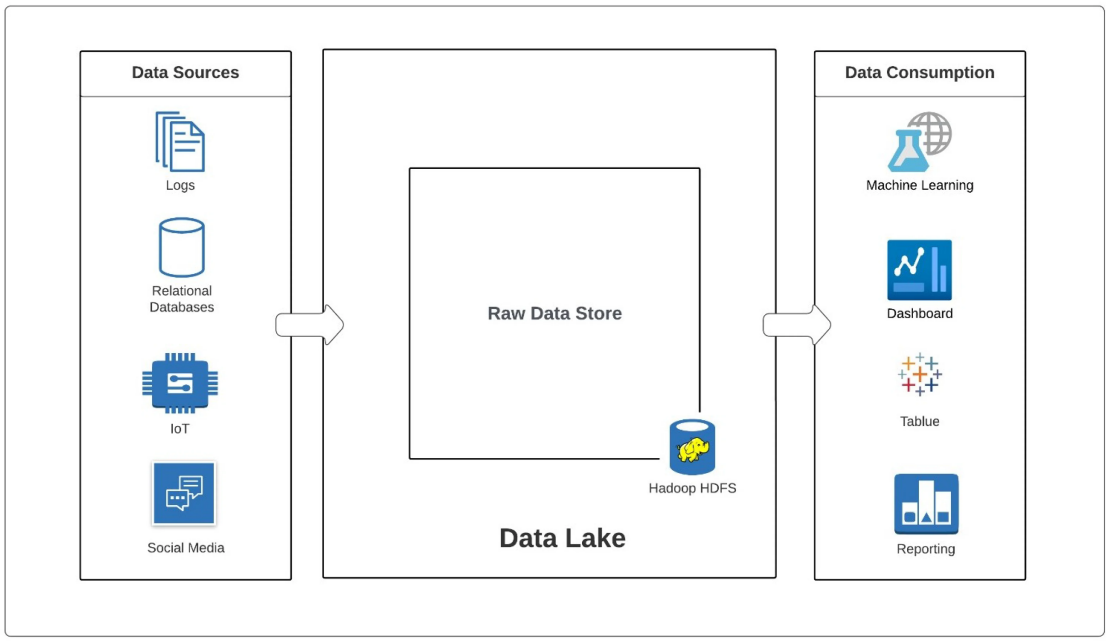


Figure 3.1: Mono-zone architecture for data lakes [11].

highly adopted variation is the multi-zone functional architecture, which incorporates multiple zones, namely the raw ingestion layer, the process stage, the access zone, and the governance zone [27], as illustrated in Figure 3.2.

In the context of modern Lakehouse deployments, this multi-zone approach is universally implemented through the **Medallion Architecture**. Defined as a structured framework for data refinement, it enhances data quality and accessibility through bronze, silver, and gold tiers. Specifically, this model relies on sequential stages that incrementally cleanse and prepare data for analytical and AI-driven applications [12]:

- **Bronze Layer (Raw Data Zone):** The bronze layer captures incoming raw data, typically preserved in append-only logs. It is designed to store data in its original format, supporting *schema-on-read* and enabling traceability.
- **Silver Layer (Process Zone):** The silver layer cleanses, integrates, and converts semi-structured data into reliable, queryable formats through complex transformations, including schema enforcement, missing values handling, deduplication, and semantic normalization.
- **Gold Layer (Access Zone):** The gold layer delivers refined, aggregated, and analytics-ready tables tailored for enterprise or AI applications. It presents

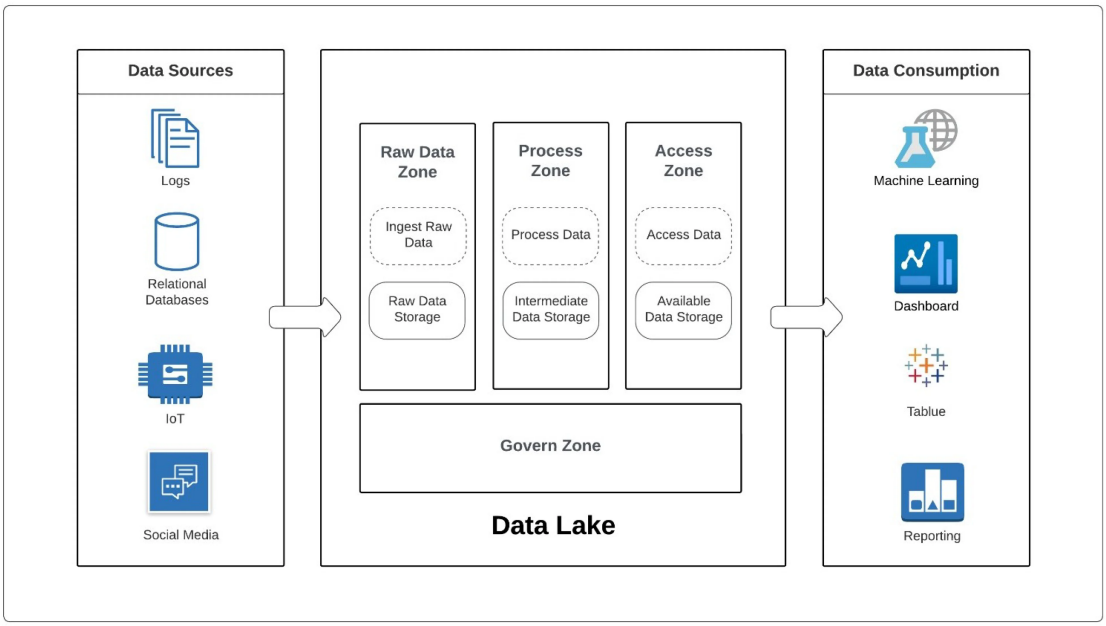


Figure 3.2: Multi-zone functional architecture for data lakes [11].

highly curated, often consolidated data suitable for consumption by business intelligence tools and machine learning models.

Such logical separation of concerns is essential for DataOps. By organizing data into maturity zones, the Medallion Architecture streamlines testing, compliance, and AI model governance, and allows for interoperability across platforms. Furthermore, to ensure the integrity of these pipelines, metadata is produced and enriched at each layer, enabling downstream AI models to validate input quality and detect drift [12], as illustrated in Figure 3.3.

3.1.3 Data Lakehouse Architecture

The current state-of-the-art that serves as the foundation for advanced DataOps and automated pipelines is the Lakehouse. This paradigm represents a novel architectural design that emerges as a potential replacement for traditional data warehouses. It effectively merges the flexibility of data lakes alongside the transactional guarantees of data warehouses across its layered structure [3], as shown in Figure 3.4.

At its foundation, it leverages cloud object storage to store data in open file formats like Parquet, ORC, and Avro, while a transactional metadata layer is positioned directly above the storage tier. This layer ensures data integrity by providing warehouse-grade **ACID** (Atomicity, Consistency, Isolation, Durability)

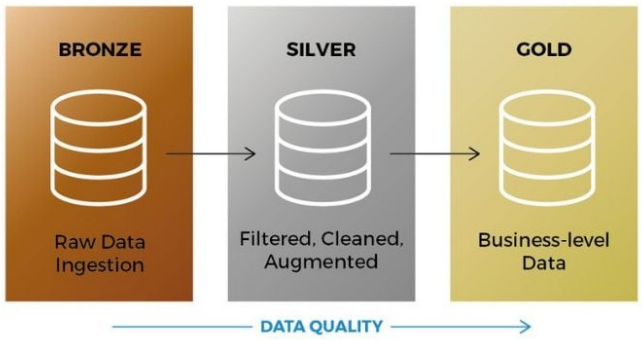


Figure 3.3: Medallion Architecture: Layered Data Refinement Model for AI-Ready Pipelines [28].

guarantees and critically supports schema enforcement, data versioning, and data lineage. To achieve this, open table formats such as Delta Lake, Apache Hudi, and Apache Iceberg are essential within this layer by bringing transactional capabilities to the data lake [29].

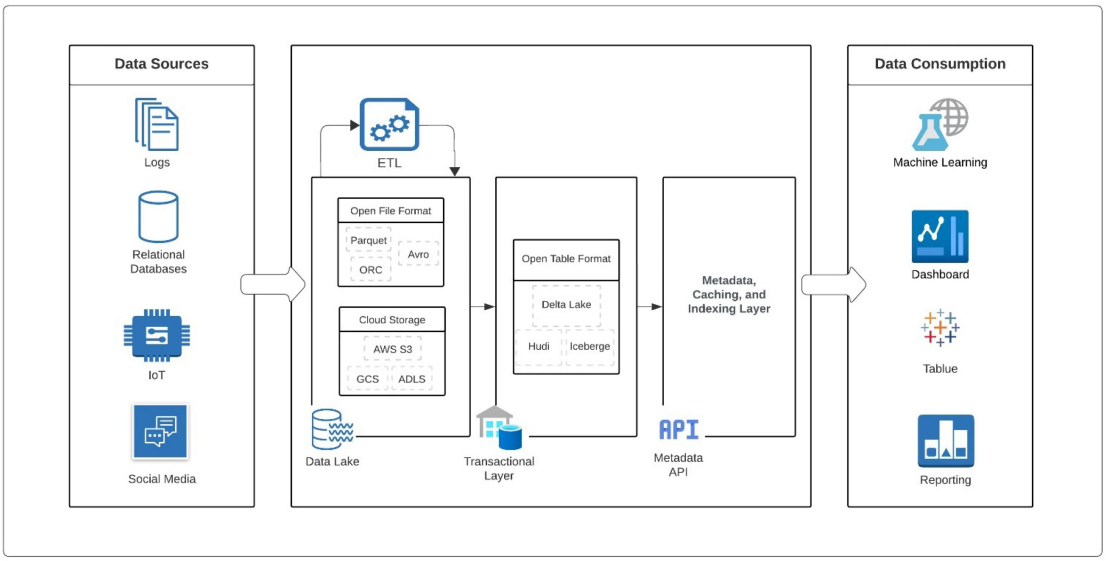


Figure 3.4: Data Lakehouse architecture [11].

3.1.4 Transactional Storage with Delta Lake

While the Lakehouse provides the conceptual framework, its practical implementation relies on overcoming the inherent limitations of cloud infrastructure. Cloud

object stores, such as Amazon S3 and Azure Blob Storage, are some of the largest as well as highly economical storage systems on the planet, making them an attractive solution for hosting extensive data warehouses and data lakes. Unfortunately, managing mutable tabular datasets on these systems is highly problematic, as they make it difficult to achieve ACID transactions and high performance [30].

To address these challenges and enable true DataOps workflows, specialized storage layers were developed. **Delta Lake** is an open-source storage layer that adds ACID transaction support on top of cloud object stores, maintaining information about data objects via a write-ahead log that is likewise retained within the cloud object store [30]. This architecture effectively removes the need to manage multiple fragmented storage systems, unifying streaming, batch, and analytical workloads, as shown in Figure 3.5.

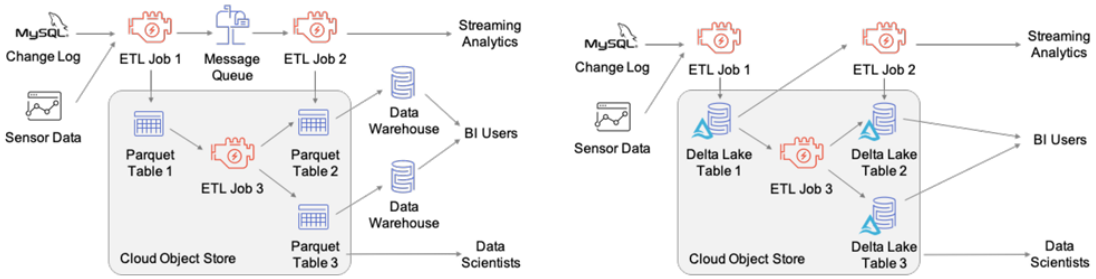


Figure 3.5: Comparison of a traditional data pipeline built on three storage systems (a message queue, object store and data warehouse) against a unified architecture employing Delta Lake for both stream and table storage. This unified approach eliminates redundant data copies by relying exclusively on low-cost object storage [30].

Based on this transactional design, Delta Lake introduces several high-level features that act as the technical backbone for automated data pipelines [30]:

- **Time Travel and Rollbacks:** It allows users to query historical point-in-time snapshots and revert accidental modifications across their data. This is a critical self-healing capability in DataOps, enabling engineers to easily undo the effects of a crashed update job or corrupted data ingestion.
- **Data Manipulation Language (DML) Operations** (UPSERT, DELETE, MERGE): Unlike traditional immutable data lakes, Delta Lake efficiently handles modifications, rewriting the relevant objects to implement updates to archived data and compliance workflows.
- **Schema Evolution and Enforcement:** Because enterprise datasets are continuously updated, schema drift is a major cause of pipeline failure. Delta

Lake solves this by providing schema evolution, which enables queries over legacy Parquet files without rewriting them when the table structure evolves.

- **Data Layout Optimization:** Performance in large-scale data lakes is heavily dependent on data layout. Delta Lake leverages advanced indexing techniques like Z-Ordering, which works hand-in-hand with data statistics to let queries read less data, ensuring fast analytics without requiring separate, highly available serving systems.

Integrating these features into layered architectures is paramount. Delta Lake augments Spark's capabilities by providing ACID transactions, schema enforcement, and time travel—features essential for developing robust and reproducible AI pipelines. This integration guarantees that transformations across the bronze to gold layers preserve consistency, accuracy, and lineage, even during simultaneous write scenarios [12].

3.2 The Correct-by-Design Lakehouse

While technologies like Delta Lake ensure table-level ACID properties, modern DataOps environments face broader structural challenges. Although the data lakehouse serves as the **de facto** cloud infrastructure across analytics and AI workloads, current implementations exhibit fragility when subject to concurrent, unverified changes: schema discrepancies frequently emerge only at runtime, development and production environments easily diverge, and multi-table pipelines risk leaking partial states upon unexpected failures [22].

The current literature identifies three primary failure modes in traditional platforms that compromise data reliability:

- **Schema failures:** It is observed that a large proportion of pipeline errors derive from schema changes at the boundary between two nodes [31], as columns get dropped or replaced, data types evolve, or semantic definitions shift.

- **Collaboration failures:** Standard software development workflows struggle because the analogy with data teams breaks down: pipeline code is inherently stateful, with its outcome directly dictated by the underlying condition of the data lake.

- **Correctness failures:** Researchers highlight that even if a lakehouse guarantees table-level atomicity through open formats, pipelines are inherently multi-table and frequently multi-language. In the absence of formal "transactional pipeline" guarantees, downstream readers may observe an inconsistent

state combining old and new tables without knowing whether execution completed. While these failure modes are already challenging for workloads run by trusted human experts, their severity is amplified when autonomous, potentially unsafe agents operate on production data [22].

To mitigate these vulnerabilities, the frontier of DataOps architecture proposes a shift toward a correct-by-design lakehouse paradigm. This approach aims to eliminate failures by construction rather than patching them after the fact. The paradigm explores the design space along three distinct axes:

- **Data contracts through type checks:** Just as static type checks prevent many software bugs, lightweight table contracts must prevent interface discrepancies across pipeline nodes. These explicit interfaces allow the system to catch mismatches immediately during development rather than during execution.
- **Collaboration through Git-for-data:** The paradigm suggests that extending Git-like semantics to data assets facilitates point-in-time debugging and pull requests (PRs) that support human-in-the-loop verification. By treating tables and branches with the same mental model as code repositories, data teams can safely test and isolate changes without impacting production [22].
- **Correctness through transactional pipelines:** To prevent multi-table inconsistencies, the correct-by-design architecture automatically snapshots reads and sandboxes writes during a specific pipeline execution. Rather than updating tables sequentially—where a failure would leave the system in an inconsistent state with mixed versions of parent and child tables—this approach guarantees pipeline atomicity. Downstream systems observe either all outputs of a successful run or none, completely isolating partial failures [22].

Figure 3.6 illustrates the Git-for-data paradigm, showing how branch-based isolation enables point-in-time debugging and safe merging of data changes. Figure 3.7 demonstrates the transactional pipeline mechanism, which guarantees atomic publication of all output tables upon success and complete rollback in case of failure.

Taken together, these primitives represent a fundamental evolution in DataOps, producing pipelines that are statically verifiable, safe to run, readily reviewable, and effortlessly reproducible [22].

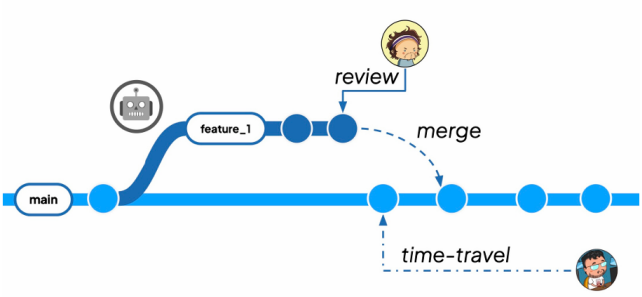


Figure 3.6: Git-for-data and collaboration: point-in-time debugging and branch merging [22].

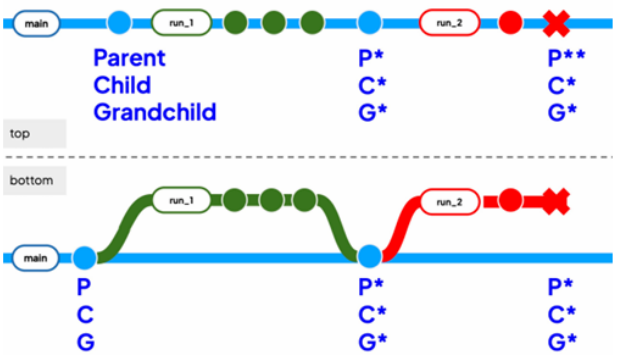


Figure 3.7: Transactional pipelines: guarantee of atomic publication for all target tables on success, alongside complete isolation in case of failure [22].

3.3 The Transformation Framework: Data Build Tool (dbt)

Within the modern data stack, organizations increasingly rely on cloud-based tools that handle the full data journey. While ELT platforms automate data movement into the warehouse, the transformation step requires specialized frameworks [32]. **Data Build Tool (dbt)** is an open-source analytics engineering framework that enables teams to transform raw data loaded into a data warehouse or lakehouse using SQL-based workflows [33].

As highlighted by dbt Labs [32], dbt is explicitly not an ETL tool, a BI platform, or a data ingestion service; rather, it is a dedicated transformation and modeling layer built for analysts using SQL, designed for modular, testable, and documented workflows. The tool acts as an orchestration layer on top of the data warehouse to improve and accelerate the data transformation process. As illustrated in Figure 3.8,

it works by pushing down the code—doing all the calculations at the database level—making the entire transformation process faster, more secure, and easier to maintain [34].

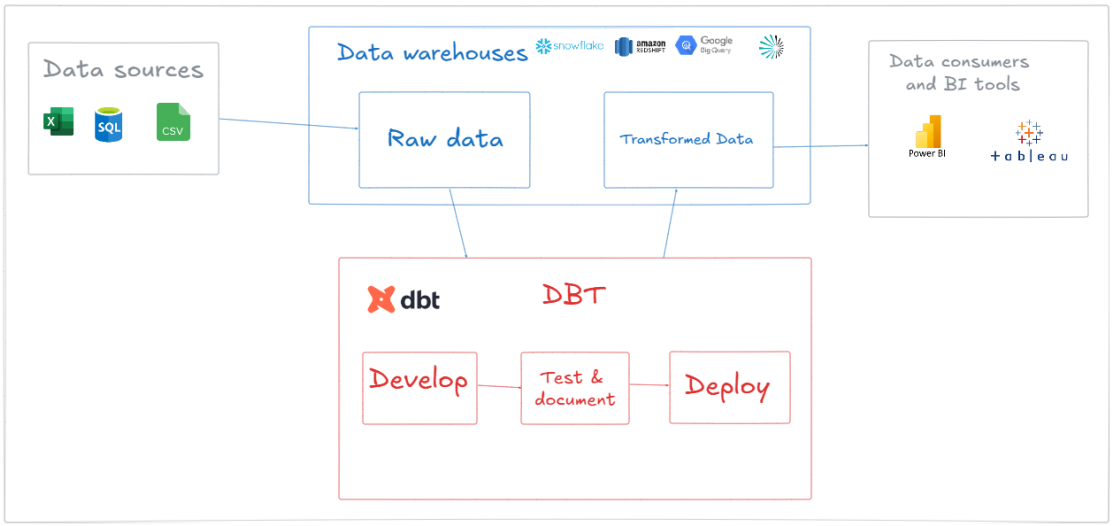


Figure 3.8: The role of dbt in the modern data stack as a transformation layer [35].

3.3.1 Software Engineering Principles in Analytics

The core philosophy of dbt is that it turns data analysts into engineers, allowing them to own the entire analytics engineering workflow by combining modular SQL with software engineering best practices [34]. By abstracting complex transformations into reusable models and integrating seamlessly with the modern data stack, dbt enables teams to build scalable, trustworthy, and auditable data pipelines [33].

This is achieved through several key methodological advantages:

- **Modularity and Reusability with Jinja:** dbt lets teams build data models directly inside data warehouses using SQL `SELECT` statements that, instead of being temporary queries, are stored, versioned, and reusable [32]. Using Jinja templating, dbt enables developers to inject logic such as loops and conditionals into their SQL models, reducing duplication and improving maintainability across datasets [33].
- **Automated Testing and Quality Assurance:** To support data quality, dbt includes built-in testing for critical fields, such as checking for null values or uniqueness. These automated tests catch bad data before it reaches dashboards or downstream processes [33].

- **Dependency Tracking and Lineage:** The framework brings visibility to transformation pipelines with automatic dependency tracking and lineage graphs. Its model dependency graphs ensure that transformations are executed in the correct order, improving transparency and reducing the surface area for debugging [33, 34].
- **Version Control and CI/CD:** Because dbt projects usually connect with Git, teams can review updates, roll back mistakes, and automate deployments [32]. This integration lowers the technical barrier to modern pipeline development and brings software engineering best practices to analytics workflows [33].

3.3.2 Layered Data Modeling with dbt

To get the most out of dbt, organizations adopt a structured project layout for clarity, maintainability, and scale. In environments built on the data lakehouse, dbt seamlessly aligns its transformation workflows with the **Medallion Architecture** introduced earlier in this chapter [33, 32].

Rather than redefining the data architecture, dbt acts as the operational engine that executes the progression of data through the established tiers. It references the raw data stored in the **Bronze Layer** as its starting point [33]. It then applies its modular SQL models, Jinja templating, and automated tests to perform the cleansing and integration required for the **Silver Layer**. Finally, dbt handles the complex aggregations and business logic necessary to deliver the trusted, analytics-ready tables of the **Gold Layer** [33, 32].

Ultimately, by relying on these structured layers and software engineering principles, dbt simplifies and speeds up the process of transforming data [34], ensuring that all team members can collaborate effectively on a single, version-controlled project structure [32].

3.4 Infrastructure as Code: Databricks Asset Bundles (DABs)

A typical modern data project comprises multiple distinct components: the *code* (such as notebooks and Python files containing business logic), the *resources* (including workflows, pipelines, and machine learning models), and the *execution environment* spanning development, staging, and production workspaces [36, 37]. Historically, orchestrating the Continuous Integration and Continuous Deployment (CI/CD) journey across these components presented significant challenges. Manual orchestration via user interfaces is time-consuming, highly error-prone, and not a viable option for automated CI/CD processes. Conversely, programmatic approaches using REST APIs or SDKs require advanced programming knowledge,

while tools like Terraform can be challenging for data scientists and engineers who may lack a deep background in infrastructure management [36].

To simplify this CI/CD process and manage complex projects where multiple contributors and automation are essential, **Declarative Automation Bundles** (formerly known as Databricks Asset Bundles) have been introduced, providing an Infrastructure-as-Code (IaC) approach that promotes core software engineering best practices—including version control, code review, testing, and continuous delivery—for data and AI projects [38].

The core philosophy of a bundle is to provide an end-to-end definition of a project, detailing exactly how the project should be structured, tested, and deployed [38, 37]. Instead of treating code, resources, and environments as siloed entities, bundles use human-readable YAML files to specify and co-version the artifacts, resources, and configurations together [36, 38]. This approach enables teams to write code once and deploy it everywhere, managing all necessary components in a consistent and efficient way across multiple environments [36], as illustrated in Figure 3.9.

3.4.1 Environment Isolation and CI/CD Pipelines

One of the primary advantages of utilizing Declarative Automation Bundles is the ability to securely orchestrate the journey to production through strict environment isolation. By specifying environment-based overrides and variables within the YAML configurations, DABs easily override values based on the target environment, ensuring flexibility while guaranteeing user isolation during deployment [36].

In a typical CI/CD pipeline governed by a version control strategy, this journey progresses through three isolated stages [36]:

- **Development:** The project is deployed to the development environment from a feature branch. This allows developers to safely test their changes away from production, often utilizing specific, cost-effective configurations (such as single node compute) and personal user identities.
- **Staging:** Upon creating a pull request, the bundle is deployed to the staging environment. The aim here is to closely resemble the production environment. Testing continues on new staging data using robust configurations, and the execution is typically performed via a Service Principal Identity (Functional SSO) to offer better security than individual user access.
- **Production:** Once all tests pass in staging, the bundle is fully deployed to the production environment utilizing production data. Automated schedules and serverless computing resources ensure the reliable delivery of data products to consumers, governed strictly by service principals.

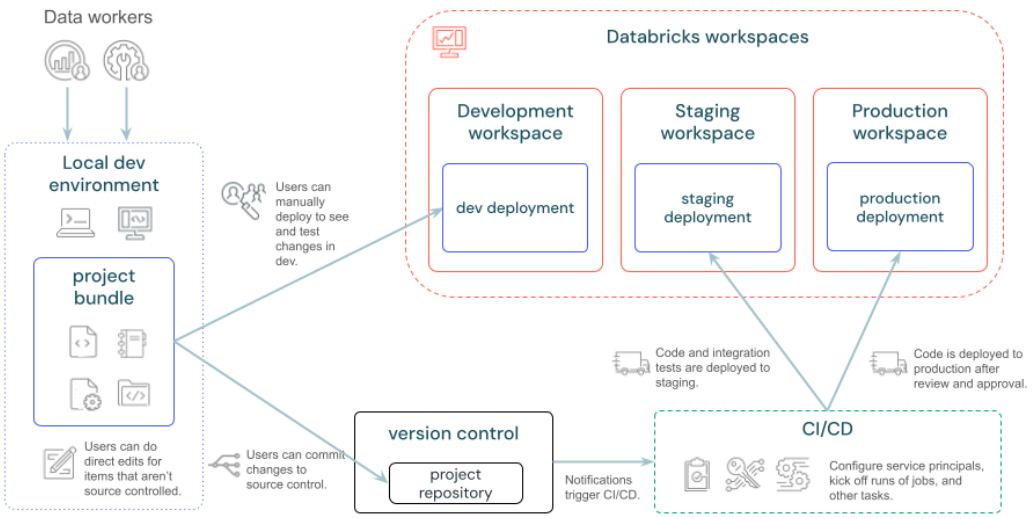


Figure 3.9: High-level view of a development and CI/CD pipeline with Declarative Automation Bundles [38].

By standardizing these environments through code, bundles help maintain a versioned history of both the application logic and the infrastructure configuration. In industries where regulatory compliance is a significant concern, maintaining a comprehensive version history is essential, as it supports governance and ensures that necessary compliance standards are rigorously met [38]. Ultimately, Declarative Automation Bundles make it significantly easier to collaborate on projects during active development, setting organizational standards through consistent, code-driven deployments [38].

3.5 Agentic AI in Data Operations

While the infrastructure and orchestration components discussed so far automate the execution of data pipelines, they remain fundamentally reactive: failures still require human engineers to detect, diagnose, and resolve them [39]. This section examines how autonomous, Large Language Model (LLM)-driven agents are reshaping this reactive paradigm into a proactive, self-healing one. It first discusses the shift from traditional DataOps to AIOps [39], then introduces a structured failure taxonomy and diagnostic workflow proposed by IBM Research [40], the ReAct reasoning paradigm [41], the cognitive architecture underlying modern LLM-based agents [42], and finally the governance mechanisms that keep such autonomous systems safely bounded within production environments [40].

3.5.1 Evolution from DataOps to AIOps

While traditional DataOps frameworks have introduced agility into data management, they still rely heavily on human intervention, manual monitoring, and reactive troubleshooting [39]. As data velocity and pipeline complexity increase in modern cloud infrastructures, reactive problem-solving approaches—where data engineers are manually alerted only after downstream corruptions have occurred—become increasingly impractical.

This operational limit has driven a paradigm shift from traditional DataOps toward AI-powered DataOps (AIOps). As highlighted by Chakraborty [39], AIOps leverages artificial intelligence to create self-learning systems that autonomously monitor, detect, and address potential issues before they impact business operations. The culmination of this paradigm is the **self-healing data pipeline**: an autonomous infrastructure capable of detecting execution failures, diagnosing root causes, and executing corrective actions with minimal human overhead [39].

3.5.2 Failure Taxonomy and Multi-Phase Diagnosis Workflow

To systematically classify and resolve operational breakdowns in complex data workflows, recent research by IBM Research [40] formalizes a comprehensive **Failure Taxonomy for Data Pipelines**. As detailed in Table 3.1, this taxonomy organizes pipeline failures along eight primary operational dimensions: Invalid Connection Request, Invalid References, Schema & Data Quality, SQL Errors, Infrastructure Bottlenecks, Access Rights, Data Transformation Logic, and Configuration & Runtime Errors.

Following this taxonomy, Vijaykeerthy et al. [40] structure the self-healing workflow into three sequential phases mirroring human engineering reasoning, as illustrated in Figure 3.10:

1. **Input Processing and Issue Classification**: Execution logs, execution context, and pipeline schemas are normalized to categorize the primary failure into the taxonomy.

2. **Root Cause Analysis**: The agent traces implicit causal chains across pipeline stages, explicitly distinguishing the node where the issue originated from downstream nodes exhibiting cascading symptoms.

3. **Remediation Recommendation**: Actionable code or configuration corrections are generated and prioritized based on operational constraints.

Table 3.1: Data Pipeline Failure Taxonomy (adapted from Vijaykeerthy et al. [40]).

Category	Manifestations
Invalid Connection	Invalid user/password, unreachable endpoint parameters.
Invalid References	Table or model not found, unresolvable column names, broken `ref()` macros.
Schema & Data Quality	Column data type mismatch, constraint violations (`not_null`, `unique`), format parsing errors.
SQL	Syntax errors, missing keywords (`END`, commas), deadlocks, division by zero.
Infrastructure	Connection/query timeouts, network failures, compute resource constraints.
Access	Permission denied, missing tokens, invalid API keys.
Data Transformation	Invalid or semantically incorrect transformation logic (e.g., wrong aggregation).
Configuration & Runtime	Missing runtime parameters, invalid environment variables.

3.5.3 Agentic Reasoning and the ReAct Framework

To move beyond static single-turn prompt engineering, modern autonomous debugging architectures rely on the **ReAct (Reasoning + Acting)** paradigm introduced by Yao et al. [41]. ReAct interleaves verbal reasoning traces (*Thoughts*) with tool executions (*Actions*) and environment feedback (*Observations*). This iterative loop enables the agent to dynamically track execution states, dynamically inspect broken artifacts, and adapt its repair strategy when intermediate steps encounter compiler errors, as shown in Figure 3.11.

To govern complex multi-turn trajectories without falling into non-deterministic loops, state-of-the-art implementations represent agent execution as cyclic state graphs using state-management frameworks. Within this graph architecture, state transitions are strictly governed by conditional edges. Furthermore, as advocated by IBM Research [40], production-grade agents incorporate an explicit *Self-Verification* step before outputting fixes, validating that inferred causal chains match the physical DAG nodes and that proposed code patches pass compilation checks.

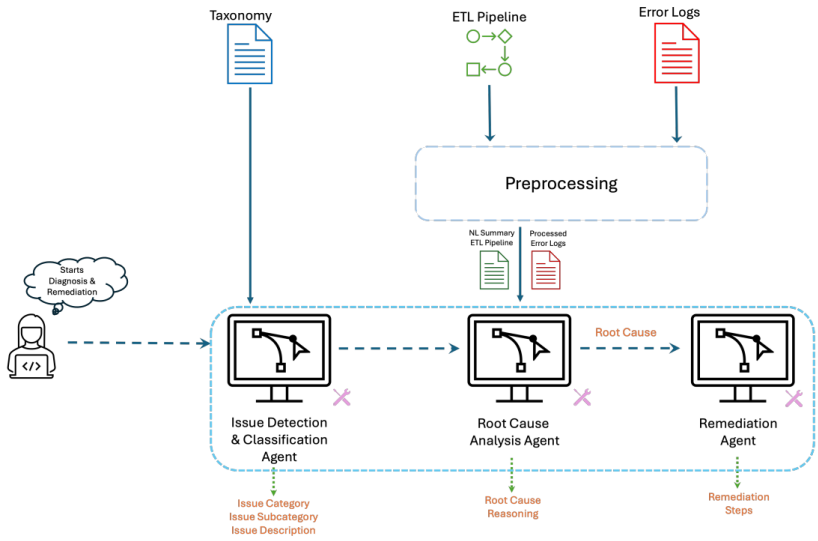


Figure 3.10: System architecture of the diagnosis and remediation framework for ETL pipelines [40].

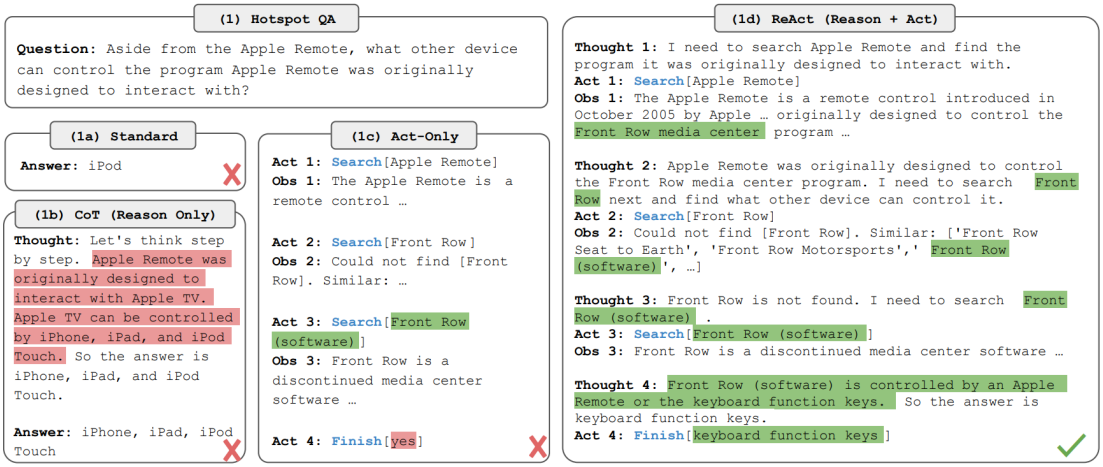


Figure 3.11: The ReAct (Reasoning + Acting) loop interleaving Thought, Action, and Observation cycles [41].

3.5.4 Cognitive Architecture of LLM-Based Agents

While the ReAct framework defines the execution loop, building a fully autonomous agent requires a comprehensive system architecture. Recent literature systematically conceptualizes LLM-based agents through a modular cognitive framework. As formalized by Wang et al. [42], a unified agent architecture consists of a central Brain (the LLM itself) surrounded by four primary functional modules:

- **Profiling Module**: Defines the persona and the specific role the agent must assume (e.g., a DataOps engineer), dictating its operational boundaries.
- **Memory Module**: Enables the agent to perceive its dynamic environment by storing short-term contextual data (such as execution logs) and recalling past interactions.
- **Planning Module**: Formulates strategies to solve complex tasks. This is where execution paradigms like ReAct are instantiated to iteratively chart a path to resolution.
- **Action Module**: Translates the agent's internal decisions into physical operations by invoking specific tools (e.g., file system access, API calls) that interact with the external environment.

This modular decomposition provides a rigorous engineering blueprint for transitioning from static LLM prompts to stateful, interactive AI systems capable of complex diagnostic workflows.

3.5.5 Bounded Autonomy and Governance Guardrails

Despite the diagnostic accuracy of LLM-based reasoning systems, deploying unconstrained autonomous agents directly into production enterprise data platforms poses severe operational risks, including silent data corruption, unvetted schema modifications, and non-deterministic logic changes. Consequently, modern self-healing pipeline architectures enforce a model of **Bounded Autonomy** governed by strict execution isolation and **Human-in-the-Loop (HITL)** review [40].

Under this governance model, the agent's autonomy is strictly contained within isolated development workspaces (*Sandbox Isolation*), where proposed SQL or YAML logic patches are compiled and verified against non-production sandbox schemas without mutating production Delta tables. Furthermore, as advocated by IBM Research [40], remediation actions are intentionally presented for user review rather than applied automatically to production, ensuring that automated systems empower rather than bypass human decision-making.

Following Git-driven software engineering standards [22, 43], the validated patch is packaged and submitted via version control APIs (e.g., GitLab Merge Requests). The Merge Request encapsulates the modified code artifacts, the underlying IBM failure taxonomy classification, and the structured reasoning trace, ensuring that ultimate approval and production merge authority remain strictly under the control of human data engineers [40, 43].

Chapter 4

Architectural Design: The Soccer Domain

This chapter describes the architectural design of the proposed Self-Healing DataOps platform, translating the methodological principles discussed in Chapter 2 into a practical system built around the European Soccer Database (ESD). Starting from this dataset as the operational System of Record, the design applies Domain-Driven Design (DDD) principles to divide the monolithic source into source-aligned and consumer-aligned bounded contexts organized within a Logical Data Mesh. This logical decomposition is then implemented as an end-to-end DataOps orchestration topology, using Databricks Asset Bundles and an autonomous Self-Healing Agent capable of diagnosing and remediating pipeline failures without human intervention.

4.1 The European Soccer Database

The first step of this project required identifying a source dataset realistic enough to reproduce the typical dynamics of a legacy enterprise system, while remaining open and reproducible. A dataset was considered suitable if it combined a normalized multi-table relational schema with no built-in mechanism for versioning or temporal traceability. The chosen dataset is the **European Soccer Database** (ESD), created by Hugo Mathien and published on Kaggle [44]. An examination of its schema confirmed that it met these conditions, reflecting precisely the kind of source a modern DataOps pipeline is designed to govern and transform. Distributed as a single SQLite file, the ESD is structured as an Online Transaction Processing (OLTP) system including seven interrelated tables normalized around integer foreign keys.

Dubitzky et al. [45] report that the database contains approximately 25,000 soccer matches from the top leagues of eleven countries, along with detailed

information on team formations and player attributes for over 10,000 athletes. The dataset has also supported academic research: for example, Golia and Carpita [46] applied Bayesian Network models to its Italian Serie A data to predict match outcomes from role-based player performance indicators.

4.1.1 Database Structure

Each of the seven tables introduced above covers a distinct domain of the football world and is described below:

- **Match**: the main table of the database, containing one record per game. Each row stores the league and season identifiers, the home and away team identifiers, the match date, the final score, and the championship round. A subset of matches also includes semi-structured event data (ball possession, corners, fouls, cards, and goal types) embedded within dedicated XML columns.
- **Player** and **Player_Attributes**: the `Player` tables lists the biographical records of 11,060 athletes, while `Player_Attributes` stores their performance rating, measured on a 0–100 scale and updated on a weekly basis, derived from EA Sports' FIFA video game series. Specifically, the `Player_Attributes` table contains 33 numeric variables describing physical and technical player skills across four broad tactical roles: goalkeeper, defender, midfielder, and forward [46].
- **Team** and **Team_Attributes**: analogously, the `Team` table holds the club master data, while `Team_Attributes` describes each team's tactical profile (preferred formation, playing style, pressing intensity) as derived from the same video game series and update periodically across seasons.
- **League** and **Country**: lookup tables that map the numeric identifiers used across the database to the names of the national leagues and countries they belong to.

Table 4.1 summarizes these seven entities, together with their record counts, which range from small reference tables (eleven records each) to nearly 184,000 weekly player performance updates.

In addition, each match record includes betting odds from up to ten international bookmakers, making the dataset particularly rich for predictive modeling and analysis.

Table 4.1: Main entities of the European Soccer Database.

Table	Description	Records
`Country`	Nations covered by the database	11
`League`	Top-flight national competitions	11
`Match`	Individual match records	25,979
`Team`	Club master data	299
`Team_Attributes`	Tactical profiles per team per season	1,458
`Player`	Player biographical records	11,060
`Player_Attributes`	Weekly performance ratings per player	183,978

4.1.2 Dataset Coverage and Sample Records

This project uses version 10 of the ESD, the latest release published on Kaggle [44]. The dataset covers the top-flight competitions of eleven Europena countries, spanning eight season from 2008/2009 to 2015/2016, with individual league sizes ranging from 1,422 matches (Switzerland) to 3,040 matches (England, France, Spain), as detailed in Table 4.2.

To illustrate the raw source format before transformation, Table 4.3 presents a simplified extract of ten records from the `Match` table, omitting the extensive betting odds and XML event columns for visual readability.

4.1.3 Dataset Role in the Ingestion Pipeline

The structural properties discussed above make the ESD a suitable foundation for a DataOps architecture that lifts raw, untracked relational data into a governed, versioned, and testable analytical platform.

In the implemented architecture, the SQLite file acts as the operational System of Record (SoR), the authoritative source from which all downstream copies of the data are derived. PySpark ingestion jobs extract the main tables and load them into the Bronze Layer on Databricks in Delta format, injecting the first set of audit metadata columns (such as `_ingest_timestamp` and `_batch_id`) to ensure full lineage traceability throughout the pipeline. Since the dataset is static by nature, the ingestion process is deliberately structured into consecutive seasonal loads, processing one season at a time: this design simulates the incremental data arrival patterns typical of real enterprise environments, where new records flow continuously rather than being available all at once. This ingestion strategy motivates the use of Delta Lake's `MERGE` operation over a simple overwrite, enabling

Table 4.2: Leagues and countries covered by the European Soccer Database.

Country	Competition	Matches	Seasons
Belgium	Jupiler League	1,728	2008/09–2015/16
England	Premier League	3,040	2008/09–2015/16
France	Ligue 1	3,040	2008/09–2015/16
Germany	1. Bundesliga	2,448	2008/09–2015/16
Italy	Serie A	3,017	2008/09–2015/16
Netherlands	Eredivisie	2,448	2008/09–2015/16
Poland	Ekstraklasa	1,920	2008/09–2015/16
Portugal	Liga ZON Sagres	2,052	2008/09–2015/16
Scotland	Premier League	1,824	2008/09–2015/16
Spain	Liga BBVA	3,040	2008/09–2015/16
Switzerland	Super League	1,422	2008/09–2015/16
Total		**25,979**	

idempotent updates across repeated pipeline runs.

4.2 Domain Decomposition into Bounded Contexts

As established in Chapter 2, centralized data architectures frequently become operational bottlenecks in complex, multi-domain environments. To overcome this limitation, the architecture adopted in this project follows the **Data Mesh** paradigm, based on the principles proposed by Dehghani [23] to decentralize the ownership of analytical data by partitioning monolithic legacy databases into autonomous, domain-oriented data products.

In this project, the principle is implemented by applying a **Domain-Driven Design (DDD)** approach [24] to the logical decomposition of the ESD's operational entities: rather than ingesting the raw SQLite tables as a single monolithic schema, the source dataset is partitioned into four distinct **Bounded Contexts**, each defining an explicit boundary that ensures internal consistency and semantic cohesion within its domain model [47].

Table 4.3: Sample records from the `Match` table (Italian Serie A, Season 2010/2011, Matchday 31).

Home Team	Away Team	Home Goals	Away Goals	Date
Milan	Inter	3	0	2011-04-02
Brescia	Bologna	3	1	2011-04-02
Napoli	Lazio	4	3	2011-04-03
Catania	Palermo	4	0	2011-04-03
Lecce	Udinese	2	0	2011-04-03
Roma	Juventus	0	2	2011-04-03
Chievo Verona	Sampdoria	0	0	2011-04-03
Cesena	Fiorentina	2	2	2011-04-03
Genoa	Cagliari	0	1	2011-04-03
Parma	Bari	1	2	2011-04-03

4.2.1 Methodology and Criteria

To partition the European Soccer Database into distinct bounded contexts, the architectural design reflects a domain-oriented decomposition based on functional responsibilities: a *Competitions* domain maintaining master reference data, a *Roster* domain tracking athlete profiles and their historical evolution, a *Match* domain recording transactional game outcomes and tactical attributes, and a *BI Analytics* domain combining upstream data to provide reporting dashboards. This functional partitioning was formalized through two engineering criteria:

1. **Data Volatility and Lifecycle Dynamics**: entities exhibiting substantially different update frequencies and lifecycle patterns are kept separate. Reference data, such as countries, leagues, and team identities, is relatively stable and therefore requires limited temporal management, whereas player attributes evolve regularly and are modeled using **Slowly Changing Dimensions (SCD Type 2)** [1] to preserve their historical state across seasons. Match records, in contrast, are treated as immutable once ingested into the analytical pipeline, while tactical team attributes (`Team_Attributes`), although time-varying, are grouped within the Match context because their historical profiles are primarily used as match-level analytical features. This separation of lifecycle characteristics prevents historical versioning logic from being coupled with low-volatility reference data and immutable match records.

2. **Source vs. Consumer Orientation**: following Dehghani's Data Mesh

Table 4.4: Mapping of the European Soccer Database entities to Data Mesh bounded contexts.

Bounded Context	Primary Entities / Inputs	Domain Characteristics
Competitions	`Country`, `League`, `Team`	Source-Aligned; static master data; resolves numeric IDs to names
Roster	`Player`, `Player_Attributes`	Source-Aligned; high volatility; historical versioning via SCD Type 2
Match	`Match`, `Team_Attributes`	Source-Aligned; append-heavy; consumes Competitions & Roster metadata
BI Analytics	`obt_matches_denormalized`, `gold_bi_*`	Consumer-Aligned; OBT model to minimize runtime joins; BI reporting marts

taxonomy [23], domains are categorized as either **source-aligned**, directly ingesting operational entities from the source system, or **consumer-aligned**, combining data across multiple source domains to serve analytical dashboards and business intelligence.

While a fully distributed Data Mesh materializes domain ownership across autonomous teams and cloud accounts, this project adopts a **Logical Data Mesh**: domain boundaries are explicitly enforced at the codebase level through isolated dbt directories, dedicated models, and independent target database schemas, while orchestration remains unified within a single Databricks Asset Bundle.

Applying these criteria leads to three source-aligned domains (*Competitions*, *Roster*, and *Match*) and one consumer-aligned domain (*BI Analytics*), summarized in Table 4.4.

4.2.2 Source-Aligned: Competitions Domain

The Competitions domain provides the foundational master data layer, integrating the `Country`, `League`, and `Team` entities. This domain is characterized by very low volatility: country and league records remain stable across the eight covered seasons, while club master data changes only occasionally as teams enter, leave, or change their associated competitions.

Architecturally, this bounded context supplies the reference data required by other domains to map numeric identifiers to human-readable names: centralizing this low-volatility reference data allows downstream models to consume clean, standardized information without duplicating reference-resolution logic across the pipeline.

4.2.3 Source-Aligned: Roster Domain

The Roster domain governs player-related data by managing the `Player` and `Player_Attributes` entities: unlike the Competitions domain, player attributes exhibit high volatility, changing regularly to reflect evolving performance-related characteristics.

To maintain historical accuracy over time, this domain models attribute updates as Slowly Changing Dimensions (SCD Type 2). Isolating player data keeps this temporal tracking logic contained within the Roster domain, providing downstream users with a straightforward interface for retrieving a player's historical rating at a given point in time.

4.2.4 Source-Aligned: Match Domain

The Match domain forms the transactional core of the architecture, mapping to the `Match` entity and recording individual matches and their associated match-level attributes.

This is an append-heavy domain in which each record represents a single match associated with a date, home team, away team, and league, and is treated as immutable once ingested into the analytical pipeline. In addition to `Match`, this domain includes `Team_Attributes`, which captures tactical profiles such as formation and pressing style; these evolve across seasons in response to changes in team characteristics and are modeled as time-varying dimensions within the Match context because their historical profiles are primarily used as match-level analytical features. Architecturally, this domain depends on the Competitions and Roster domains for contextual enrichment, forming a unidirectional data flow in which transactional models consume upstream reference and player data.

4.2.5 Consumer-Aligned: BI Analytics Domain

While the previous three domains are source-aligned, the architecture also requires data products tailored for analytical reporting. In accordance with the Data Mesh principles described by Dehghani [23], consumer-aligned domains combine and transform data from multiple source domains to serve specific end-user needs.

In the Gold layer of the dbt project, this consumer-aligned context is structured into two complementary model types:

- **One Big Table (`obt`)**: adopts the **OBT pattern** [48] to produce a single denormalized table (`obt_matches_denormalized`) that combines match records with team profiles, league metadata, ad player ratings. This design is particularly effective for datasets of moderate volume, where the additional storage overhead introduced by denormalization remains acceptable [48], while reducing the need for runtime `JOIN` operations on columnar storage systems such as Delta Lake [49].
- **BI Analytics (`bi_analytics`)**: combines and transforms data from the underlying domains to build specialized analytical models, such as seasonal league standings, home advantage metrics, and tactical playstyle indicators.

By combining transactional match data with enriched reference and player data from the Competitions and Roster domains, this consumer-aligned layer keeps complex analytical queries separated from upstream data preparation, thus preserving the integrity of the source-aligned bounded contexts.

4.3 DataOps Orchestration via Databricks Asset Bundles

Implementing the logical domain boundaries established in Section 4.2 as an automated pipeline requires a centralized orchestration layer following the DataOps and continuous integration/continuous delivery (CI/CD) principles discussed in Chapter 2. In this project, orchestration workflows are defined as Infrastructure as Code (IaC) using Databricks Asset Bundles (DABs) [38], introduced in Chapter 3. The end-to-end workflow is defined in the declarative YAML file `soccer_dataops_job.yml` and version-controlled within the project repository.

In this architecture, GitLab provides the CI/CD infrastructure, storing and versioning the infrastructure definitions directly in the repository rather than relying on manual UI configurations. Modifications to the codebase trigger automated CI/CD pipelines that validate and deploy the bundle to the target Databricks workspace, promoting reproducibility and environment isolation [36].

4.3.1 Job Centralization and Topology

Consistent with the Logical Data Mesh adopted in Section 4.2, all four domains are orchestrated within a single, centralized Databricks Job (`soccer_dataops_job.yml`). This choice reduced the overhead associated with running separate jobs for a dataset of this scale, such as potential cluster cold-start latency, while

still providing task-level isolation within the DAG: source ingestion tasks run independently in parallel for each source entity, followed by unified Silver and Gold transformation stages within the same job execution.

Once deployed, the job is represented as a Directed Acyclic Graph (DAG) that integrates custom Python ingestion scripts with declarative SQL transformations executed by dbt. Figure 4.1 illustrates this end-to-end topology, together with the conditional triggering of the Self-Healing Agent's React cycle upon relevant task failure. Each stage of the workflow is detailed below:

1. **Parallel Bronze Ingestion**: the workflow starts by concurrently executing Python ingestion tasks for each raw source entity. Each task runs in the dedicated Bronze Layer and applies the idempotent `MERGE` logic discussed in Section 4.1. This parallel execution allows the source entities to be ingested independently while maintaining the source-aligned boundaries reflected in the architecture.

2. **Silver Layer Transformation**: once all ingestion tasks successfully complete, the `dbt_silver` task is triggered. Executing on the Databricks SQL Warehouse, this task performs data cleansing, quality testing, and temporal tracking through a three-step dbt workflow [32]:

   - `dbt source freshness`: checks ingestion timestamps against predefined freshness thresholds, causing the Silver task to fail if raw source data arrives late.
   - `dbt snapshot`: executes historical state tracking for time-varying attributes (SCD Type 2).
   - `dbt build -select silver`: materializes the Silver transformation models while enforcing the data contracts introduced in Chapter 3, such as `not_null` and uniqueness constraints, making each domain's transformed data independently consumable.

   Each dbt execution produces diagnostic artifacts and execution logs, including the structured `run_results.json` artifact and the `dbt.log` execution log, recording model statuses, compiled queries, and error stack traces; these artifacts form the diagnostic interface consumed by the Self-Healing Agent.

3. **Gold Layer Analytics**: the `dbt_gold` task depends strictly on the success of the Silver layer. It executes `dbt build -select gold`, applying data contracts—such as metric non-nullability and schema integrity checks—to the aggregated, consumer-aligned models before generating the denormalized One Big Table (OBT) for analytical consumption.

4. **Conditional Self-Healing Remediation**: if either the `dbt_silver` or `dbt_gold` task fails, the `self_healing` task is conditionally triggered via the `run_if: AT_LEAST_ONE_FAILED` policy. This task activates the autonomous LLM-driven remediation agent described in Section 4.4, which parses the generated `run_results.json` and `dbt.log` artifacts to diagnose the root cause, creates a code patch, and opens a validated Merge Request for human review.

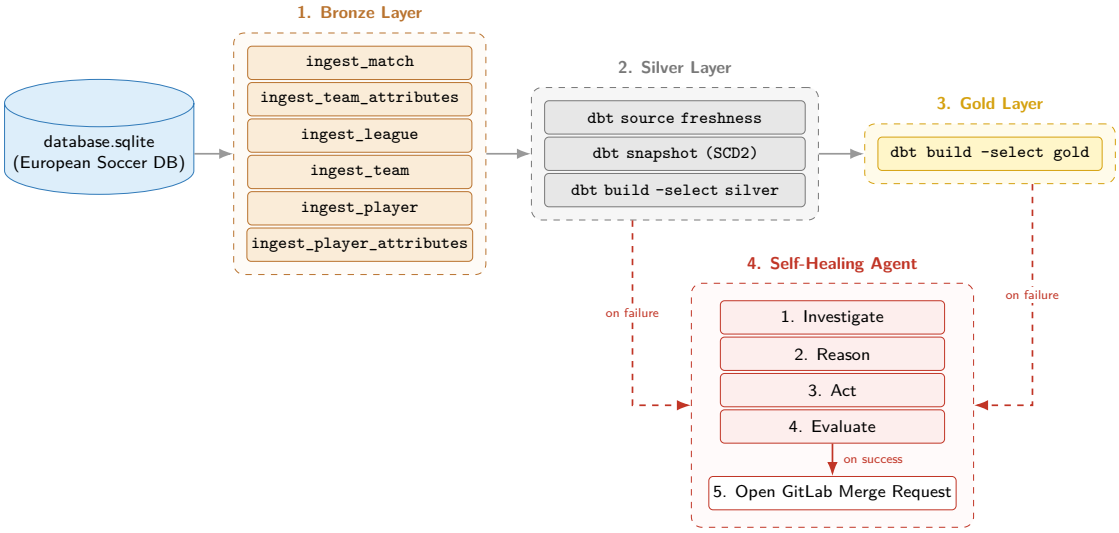


Figure 4.1: Centralized Databricks Job DAG topology across Medallion layers with conditional Self-Healing Agent triggering.

4.4 Self-Healing LLM Agent Architecture

As introduced in Section 3.5, modern DataOps frameworks support a paradigm shift toward proactive, self-healing infrastructures. To translate this concept into an operational system, the orchestration pipeline described in Section 4.3 is augmented with an autonomous, Large Language Model (LLM)-driven remediation agent.

To structure the agent's capabilities, the system adapts the unified cognitive architecture proposed by Wang et al. [42] to the DataOps domain, in which the autonomous agent is decomposed into a central Core Brain (LLM) surrounded by four functional modules (Profiling, Memory, Planning, and Action), as illustrated in Figure 4.2.

4.4.1 In-Job vs. Out-Job Approach

Before detailing the internal cognitive modules, a fundamental architectural decision involves positioning the autonomous agent relative to the data pipeline, choosing between an **Out-Job** architecture, operating as an external microservice, and an **In-Job** strategy, embedded directly within the pipeline DAG.

In an Out-Job architecture, the AI agent operates as an external service or execution component: for instance, evaluation environments like SWE-ReX and SWE-bench Live execute code generation agents within isolated Docker containers or dedicated execution pipelines [50, 51]. Although this design preserves separation of concerns, it can introduce additional operational complexity in production pipelines, including network latency, authentication overhead, and the challenge of securely transferring diagnostic artifacts across compute boundaries.

Building on the centralized job topology described in Section 4.3, this project implements the In-Job approach, embedding the remediation agent directly as a downstream task within the same DAG. This paradigm is conceptually aligned with the architecture of GitHub Agentic Workflows, where AI coding agents execute within a sandboxed environment directly inside the CI/CD workflow [52], as well as the AutoGen framework, where agents execute and validate code natively within their operating environment [53].

By setting the `self_healing` task to execute exclusively under the condition `run_if: AT_LEAST_ONE_FAILED`, this design yields three key architectural advantages:

1. **Implicit Authentication**: the agent inherits the Databricks Workspace environment and security context, reducing the need for additional authentication mechanisms for managing external API tokens or authentication endpoints.

2. **Zero-Latency Access**: co-located on the same compute plane as the transformation layer, the agent immediately accesses local execution artifacts (`run_results.json` and `dbt.log`) persisted within the workspace storage layer, without network transfer overhead.

3. **Architectural Simplicity**: executing within the unified job DAG, the agent can be triggered directly by the job's failure conditions without requiring an external event-routing mechanism.

4.4.2 Profiling and Memory Modules

Within the cognitive framework of Wang et al. [42], the **Profiling module** defines the agent's operational scope and boundaries, while the **Memory module** maintains state awareness across execution steps.

When a pipeline failure triggers the workflow, the Profiling module first analyzes the raw stack trace by executing the *Input Processing and Issue Classification* phase introduced in Section 3.5. Using a structured-output LLM invocation, this module categorizes the error according to the adapted failure taxonomy (Table 4.5), reducing contextual noise and limiting unsupported diagnostic inferences before the diagnostic context is assembled.

To bridge the gap between traditional imperative ETL pipelines and modern declarative SQL DAGs, the taxonomy consolidates foundational categories from IBM Research [40] (SQL, Invalid References, Schema & Data Quality, Infrastructure & Access) and extends them with three dbt-specific failure nodes:

- **Cascading Dependencies**: captures multi-hop DAG failures where upstream schema changes propagate failures to downstream consumers through model dependencies and `ref()` relationships, guiding the agent to trace lineage back to the root cause model rather than applying local workarounds.
- **Data Quality Contracts**: captures violations of declarative YAML contract assertions (such as `not_null` and uniqueness tests) enforced during execution.
- **Macro & Refactoring**: addresses interface or logic changes in shared Jinja macros that require coordinated updates across multiple consuming SQL models.

Once the failure is classified as in-scope, the Memory module scans the Databricks execution artifacts to extract the failing model identifier, the faulty file path, and compilation details, structuring this runtime context within short-term memory.

To preserve system safety, the operational scope of the remediation agent is strictly confined to DataOps-centric failures: if the Profiling module identifies an infrastructure or access anomaly, the agent immediately halts and escalates the incident for human review.

4.4.3 Planning Module

Once the runtime context is loaded into memory and the failure is profiled, execution shifts to the **Planning module**, which charts the trajectory toward resolution [42]. In this architecture, the planning engine is built upon **LangGraph**, a framework designed to construct stateful LLM workflows represented as graphs.

The planning logic materializes the **ReAct** (Reasoning and Acting) paradigm [41] introduced in Section 3.5. The theoretical ReAct loop of *Thought-Action-Observation* is specifically instantiated for the DataOps domain through a four-step execution flow:

Table 4.5: Failure taxonomy for DataOps pipelines, adapted and extended from Vijaykeerthy et al. [40] with dbt-specific categories.

Failure Category	**Description**	**Agent Scope**
SQL Syntax & Runtime	Malformed SQL syntax, compilation errors, or Spark SQL runtime exceptions.	In-Scope
Invalid References	Unresolvable model identifiers, unclosed Jinja tags, or invalid `ref()` calls.	In-Scope
Schema & Data Quality	Column ambiguity, non-existent fields, data type mismatches, or SLA freshness breaches.	In-Scope
Cascading Dependencies	Upstream schema drift propagating errors downstream across the dbt DAG.	In-Scope
Data Quality Contracts	Violations of declarative YAML test expectations (`not_null`, `unique`, contracts).	In-Scope
Macro & Refactoring	Jinja macro interface changes requiring coordinated updates across dependent models.	In-Scope
Infrastructure & Access	Cluster timeouts, driver out-of-memory errors, and workspace permission denials.	*Out-of-Scope*

1. **Investigate**: the agent invokes read tools to inspect the failing dbt SQL models or schema YAML configurations.
2. **Reason**: leveraging the taxonomy classification and the retrieved codebase context, it traces the root cause across the DAG and formulates a targeted repair hypothesis.
3. **Act**: it executes patch operations to modify the local code artifacts within the isolated workspace.
4. **Evaluate**: it triggers sandbox compilation and test execution to observe environment feedback and verify patch correctness prior to finalization.

To govern this iterative ReAct workflow and prevent non-deterministic infinite loops, the LangGraph state machine enforces a strict upper bound via a `MAX_ITERATIONS` threshold, providing a bounded number of iterations and predictable execution limits.

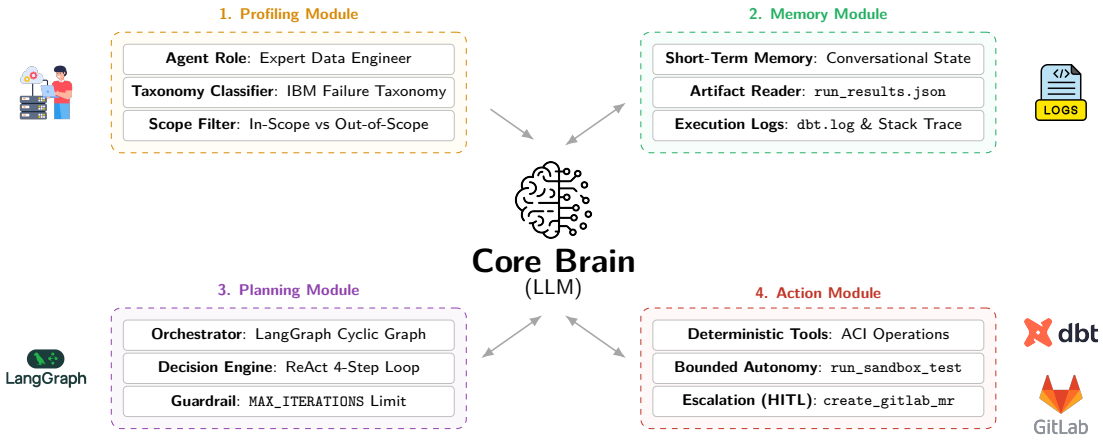


Figure 4.2: Cognitive architecture of the Self-Healing AI Agent (based on Wang et al. [42]), showing the Core Brain (LLM) and its interaction with the Profiling, Memory, Planning, and Action modules.

4.4.4 Action Module

The final component of the cognitive architecture is the **Action module**, responsible for translating the agent's decisions into concrete actions that interact with the execution environment [42]. Following the **Agent-Computer Interface (ACI)** design principles established by SWE-agent [54], this module equips the LLM with a set of deterministic tools, including local file system operation (`read_local_file`, `patch_local_file`) and orchestration commands.

To ensure enterprise-grade safety, the Action module enforces the model of **Bounded Autonomy** discussed in Section 3.5. The agent operates exclusively within a secure sandbox environment: when applying a fix, it must invoke the `run_sandbox_test` tool, which compiles and executes the modified dbt models against a dedicated, isolated Databricks schema.

If the sandbox validation tool returns a successful result, the Action module proceeds to the escalation phase, invoking the `create_gitlab_mr` tool to commit the repaired code to a new branch and open a Merge Request on the repository. By terminating execution at the Merge Request stage, the architecture enforces a strict **Human-in-the-Loop (HITL)** workflow: the autonomous agent accelerates triage and code generation, while human data engineers maintain final approval authority over changes merged into the production codebase.

Chapter 5

DataOps Implementation and Empirical Evaluation

This chapter connects the architectural design presented in Chapter 4 to its software implementation and empirical evaluation. It details the implementation and deployment of the DataOps infrastructure within Databricks, with data governed through Unity Catalog, demonstrating how the logical Data Mesh decomposition is operationalized into testable data products using dbt Core model contracts and automated data quality tests. It then presents the architecture of the LangGraph-based self-healing agent framework and evaluates its operational resilience across a benchmark suite of realistic failure scenarios using multiple Large Language Models (LLMs).

5.1 DataOps Infrastructure Implementation

The foundational layer of the platform is a DataOps pipeline deployed on Databricks and governed through Unity Catalog [55], the unified governance layer for data and AI assets in Databricks. It enforces fine-grained access control, tracks data lineage, and logs data access events for auditing. All governed data assets follow a structured three-level namespace (`catalog.schema.object`), supporting the separation of development, production, and transient sandbox schemas required by the self-healing agent. The engineering implementation materializes the Logical Data Mesh architecture established in Section 4.2, translating conceptual domain boundaries into physical, governed data assets.

To support operational resilience and prevent unvalidated changes from reaching production analytics, the infrastructure is built around four DataOps principles: automated CI/CD through GitLab pipelines, declarative environment isolation through dedicated schemas per Medallion layer, data quality enforcement via dbt

contracts and automated tests, and execution observability through structured artifacts and logs. The following subsections detail the physical materialization of these principles across the three-tier Medallion architecture (Section 5.1.1), the separation of concerns between Databricks Asset Bundles and dbt Core (Section 5.1.2), and the multi-tiered quality and telemetry framework (Section 5.1.3).

5.1.1 Physical Implementation of the Medallion Architecture

The Medallion architecture is physicalized across three processing tiers:

1. **Bronze Layer**: The raw relational tables extracted from SQLite are stored as Delta Lake tables. Ingestion is performed by the parallel Python tasks specified in Section 4.3.1, which are packaged as Python Wheels using `uv build` [56], a Python package and environment management tool. To ensure pipeline idempotency under re-execution, these tasks perform Delta Lake `MERGE` operations conditioned on primary entity keys.

2. **Silver Layer**: Data cleansing and standardization are implemented via declarative SQL models executed by **dbt Core** against a Databricks SQL Warehouse. The Silver task executes a three-step sequential workflow: it first validates source arrival timestamps against SLA boundaries declared in the domain source configuration files (e.g., `src_match_ops.yml`) using `dbt source freshness`, halting execution if ingestion delays exceed the declared freshness thresholds; it then materializes historical attribute snapshots; and finally compiles and tests all Silver transformation models via `dbt build`. Within this transformation layer, historical attribute tracking is implemented through dedicated **dbt Snapshots (SCD Type 2)**. The snapshot definitions `player_rating_snapshot.sql` within the Roster domain and `team_tactics_snapshot.sql` within the Match domain leverage a timestamp strategy anchored to the entity `date` column. Through this mechanism, dbt derives the `dbt_valid_from` and `dbt_valid_to` temporal boundary columns, preserving the version history across successive seasonal ingestion cycles.

3. **Gold Layer**: Analytical aggregations and domain metrics are synthesized into curated data products—including `gold_match_results`, `gold_player_rating_history`, and `gold_team_tactical_profile`—and consolidated into a denormalized One Big Table (`obt_matches_denormalized`). To preserve historical consistency and prevent temporal data leakage—where future attribute updates erroneously leak into past match analytical records—`obt_matches_denormalized` incorporates a **Point-in-Time Join** [57] against

`team_tactics_snapshot`. This transformation enforces point-in-time correctness by matching each match date against the active validity window (`match_date >= dbt_valid_from and match_date < dbt_valid_to`), thereby ensuring that historical match records are enriched exclusively with the tactical attributes that were valid at the date on which the match was played.

5.1.2 Declarative Infrastructure Governance

A core strength of the platform architecture lies in the operational separation of concerns between **Databricks Asset Bundles (DABs)** and **dbt Core**, establishing a clean boundary between infrastructure lifecycle orchestration and data transformation.

As the Infrastructure-as-Code (IaC) control plane, Databricks Asset Bundles (DABs) govern the declarative deployment lifecycle of the platform through `databricks.yml`. The bundle configuration also defines the Python Wheel build process using `uv build -wheel`, configures compute resources and environment dependencies, and enforces Unity Catalog access permissions. It further injects target environment variables (`catalog`, `schema`), configures job execution parameters (`target_season`, `window_start`, `window_end`), and defines the master orchestration Directed Acyclic Graph (DAG) inside `soccer_dataops_job.yml`.

Complementing this control plane, dbt Core operates as the data transformation engine inside Databricks SQL Warehouses, governing *how* relational data is transformed and validated while DABs dictate *where* and *when* compute resources are allocated. In this capacity, dbt compiles Jinja-templated models into SQL executed on the Databricks SQL Warehouse, manages snapshot versioning, resolves model dependencies, and executes declarative data quality assertions.

This separation is reinforced by explicit branch-to-schema mappings in the deployment configuration. Following official Databricks Asset Bundles specifications [38], the architecture configures target-specific behaviors: the `dev` environment operates under `mode: development`, with schemas dynamically scoped to individual developer identities (`data_ops.<user_name>`) for collision-free iteration. Conversely, the `prod` environment operates under `mode: production`, enforcing strict branch validation against `main` and protected target schemas (`data_ops.prod`). Dedicated integration targets (`test_soccer`) and agent-driven ephemeral validation schemas (`sandbox`) complement this topology, as summarized in Table 5.1.

The `sandbox` environment plays a critical role in the self-healing architecture: when a pipeline failure occurs, the Self-Healing Agent dynamically generates an isolated Unity Catalog schema (`data_ops.sandbox_<run_id>`) and operates within an ephemeral Git patch branch, compiling and testing its candidate SQL remediation code by running `dbt build` directly against this sandbox schema. Only after the candidate fix successfully passes the configured contract assertions

Table 5.1: Environment Segregation Topology in Databricks Unity Catalog.

Environment	Git Branch	Target Schema	Governance Tier
`dev`	`develop`	`data_ops.<user_name>`	Manual development and feature testing
`test`	`develop`	`data_ops.test_soccer`	Automated CI/CD integration testing
`prod`	`main`	`data_ops.prod`	Protected production (Merge Request)
`sandbox`	Dynamic	`data_ops.sandbox_<id>`	Ephemeral agent patch validation

within the sandbox does the agent open a formal Merge Request targeting the `develop` branch.

The job topology declared in `soccer_dataops_job.yml` unifies task execution across all processing tiers into a single Databricks workspace specification, implementing the conditional task dependency model established in Section 4.3.1.

Continuous Integration and Continuous Deployment (CI/CD) automation is governed by GitLab CI/CD pipelines (`.gitlab-ci.yml`), where development pushes to the `develop` branch execute a sequential validation pipeline spanning four stages:

1. **Testing Stage (`test`)**: Runs automated unit test suites via `uv run pytest` to verify Python ingestion logic and helper modules prior to deployment.
2. **Validation Stage (`validate`)**: Executes `databricks bundle validate` to verify resource definitions, variable bindings, and DABs YAML syntax.
3. **Deployment Stage (`deploy`)**: Compiles Python wheel distributions and deploys the bundle resources to the target workspace.
4. **Execution Stage (`execute`)**: Triggers automated job runs on the Databricks cluster via `databricks bundle run`.

Listing 5.1 demonstrates the core validation and deployment logic implemented in the pipeline.

```yaml
validate_bundle:
  stage: validate
  script:
    - databricks bundle validate -t $BUNDLE_TARGET
  needs:
    - run_unit_tests
  only:
    - develop

deploy_bundle:
  stage: deploy
  script:
    - databricks bundle deploy -t $BUNDLE_TARGET
  needs:
    - validate_bundle
  only:
    - develop

deploy_prod:
  stage: deploy
  script:
    - databricks bundle deploy -t prod
  only:
    - main
```

Listing 5.1: Extract from `.gitlab-ci.yml` demonstrating automated validation, development deployment, and production deployment of DABs.

In contrast, deployments to the `main` production branch skip early unit testing and syntax validation stages, proceeding directly to production deployment (`deploy_prod`) and execution (`run_prod`) targeting `data_ops.prod`. Because code can reach `main` exclusively through an approved Merge Request from `develop`—where unit tests and bundle validations have already been satisfied—the production workflow avoids redundant test execution during production deployment while preserving the quality gates already validated during the development workflow.

To maintain rigorous credential security across the DataOps lifecycle, non-sensitive configuration parameters (catalogs, schemas, job execution parameters) are centralized inside `config.yaml`, whereas sensitive credentials are managed securely within the cloud platform. At the CI/CD level, Databricks CLI authentication parameters (`DATABRICKS_TOKEN` and `DATABRICKS_HOST`) are injected into the runner via GitLab Masked Environment Variables, dynamically generating the `~/.databrickscfg` configuration file at runtime without exposing secrets in build logs. At application runtime, Python ingestion modules and the Self-Healing Agent retrieve sensitive API keys (such as GitLab access tokens and LLM provider

keys) dynamically from **Databricks Secret Scopes** (`dbutils.secrets.get()`), enforcing a policy of zero hardcoded credentials across repository manifests and dbt projects.

5.1.3 Data Contracts and Observability Integration

In a production DataOps ecosystem, data quality testing is not an auxiliary validation step but a first-class engineering contract embedded directly within the transformation pipeline. Rigorous automated validation helps ensure that corrupted or unvalidated inputs cannot propagate across data product boundaries, protecting downstream analytical dashboards and business intelligence assets.

The platform guarantees data reliability across domain boundaries through a three-tiered automated testing hierarchy:

- **Schema Contracts and Generic Tests**: Data contracts are enforced directly within model YAML specifications (e.g., `_gold_match_ops__models.yml`), as shown in Listing 5.2. By setting `contract: enforced: true`, dbt validates the model schema during execution, while generic assertions enforce core relational invariants including primary key completeness (`not_null`), entity uniqueness (`unique`), and cross-domain referential integrity (`relationships`).
- **Package-Based Advanced Expectations**: Use the `dbt-expectations` library to validate data distributions and statistical boundary conditions (e.g., `expect_column_values_to_be_between`), detecting data drift before it propagates to analytical layers.
- **Custom Business Logic Tests**: Execute domain-specific SQL assertions stored inside the `tests/` directory. These custom queries enforce invariants such as verifying that goal counts cannot be negative and ensuring that match score differences remain mathematically consistent with recorded match outcomes.

Operationalizing the execution model established in Chapter 4, transformation tasks are executed exclusively via `dbt build`, which unifies SQL model compilation, snapshot materialization, and test assertion execution into a single Directed Acyclic Graph (DAG) pass. This execution workflow enforces a strict **Circuit Breaker Pattern** [58]: if any schema contract, expectation test, or business logic assertion fails during execution, dbt immediately trips the circuit, marking the target model as failed and short-circuiting the execution of all dependent downstream models. By halting DAG execution at the exact point of failure, the pipeline limits cascading faults, preventing failed upstream models from propagating invalid results to dependent Gold models or feeding inaccurate context into downstream autonomous reasoning loops [59].

```yaml
models:
  - name: gold_match_results
    description: "Transactional Fact Table containing individual soccer match results."
    config:
      contract:
        enforced: true
    columns:
      - name: match_id
        data_type: int
        data_tests:
          - unique
          - not_null
      - name: home_team_id
        data_type: int
        data_tests:
          - relationships:
              to: ref('silver_roster__teams')
              field: team_api_id
```

Listing 5.2: Extract from `_gold_match_ops__models.yml` enforcing strict schema contracts and relational data tests.

When a transformation failure triggers the pipeline circuit breaker, `dbt Core` automatically generates structured diagnostic execution artifacts, which are persisted directly to dedicated Unity Catalog Volumes (`/Volumes/data_ops/${bundle.target}/raw_files/dbt_artifacts/`). Two primary artifacts capture the failure state: `run_results.json`, which records structured execution telemetry, model latencies, compiled test queries, and violating record counts; and `dbt.log`, which captures detailed execution logs and error traces returned during compilation and execution. These structured artifacts constitute the primary diagnostic payload consumed by the autonomous Self-Healing Agent presented in Section 5.2. To complement pipeline-level diagnostics with agent-level observability, the agent's internal reasoning trajectories, LLM token consumption, tool invocation sequences, and ReAct decision paths are continuously streamed and logged through native **LangSmith** [60] telemetry integration.

5.2 Self-Healing Agent Implementation

Building upon the conceptual architecture outlined in Section 4.4, this section describes the software implementation of the autonomous self-healing agent. The

framework is developed in Python using **LangGraph** as the state machine engine, **Pydantic** for structured output validation, and the **Databricks SDK** for workspace and job operations. It consumes the diagnostic artifacts generated by `dbt Core` (`run_results.json` and `dbt.log`) as its primary input payload for root cause analysis and automated remediation.

The implementation evolved through an iterative engineering process across two architectural versions: a baseline framework (v1.0) that established the core ReAct decision cycle, and an enhanced guardrailed engine (v1.1) that introduced explicit lineage navigation constraints, bidirectional sandbox validation, and fallback telemetry parsing. The following subsections detail the deployment model (Section 5.2.1), the LangGraph state machine design and toolset (Section 5.2.2), and the structural architectural upgrades implemented between the two framework versions (Section 5.2.3).

5.2.1 In-Job Activation and Artifact Access

Following the In-Job deployment pattern discussed in Section 4.4, the self-healing framework is deployed as a conditional downstream task inside `soccer_dataops_job.yml`, physically implemented as a `spark_python_task` executing a packaged `run_agent.py` entrypoint.

At the authentication layer, the framework initializes workspace interactions via `WorkspaceClient()` from the **Databricks SDK for Python** (`databricks.sdk`). Executing within the job compute context, the client automatically inherits both the ambient OAuth credentials and the Service Principal identity assigned to the cluster, without requiring static API tokens or external key vault configuration.

At the data access layer, Databricks Unity Catalog Volumes provide access to governed object storage through a filesystem interface on the cluster. This allows the agent to read `run_results.json` and `dbt.log` from `/Volumes/data_ops/${bundle.target}/raw_files/dbt_artifacts/` using standard Python file I/O, without requiring cloud-specific SDKs, API calls, or external log exporters.

Activation is managed declaratively through `run_if: AT_LEAST_ONE_FAILED` within the job topology, ensuring the agent remains inactive during successful pipeline runs and consumes no LLM API tokens or additional compute unless an upstream task failure occurs, as illustrated in Listing 5.3.

If the agent task runs into a runtime exception, the job terminates in a failed state and falls back to standard operational alerts for manual engineering intervention, without directly modifying production infrastructure.

```yaml
- task_key: dbt_silver
  depends_on:
    - task_key: ingest_match
  dbt_task:
    commands:
      - "dbt build --select silver --target-path /Volumes/data_ops/${bundle.target}/raw_files/dbt_artifacts/{{job.run_id}}/silver"

- task_key: dbt_gold
  depends_on:
    - task_key: dbt_silver
  dbt_task:
    commands:
      - "dbt build --select gold --target-path /Volumes/data_ops/${bundle.target}/raw_files/dbt_artifacts/{{job.run_id}}/gold"

- task_key: self_healing
  environment_key: ai_agent
  depends_on:
    - task_key: dbt_silver
    - task_key: dbt_gold
  run_if: AT_LEAST_ONE_FAILED
  spark_python_task:
    python_file: ../self_healing_agent/run_agent.py
    parameters:
      - "/Volumes/data_ops/${bundle.target}/raw_files/dbt_artifacts/{{job.run_id}}"
```

Listing 5.3: Extract from `soccer_dataops_job.yml` illustrating the pipeline DAG sequence and the conditional activation of the Self-Healing agent.

5.2.2 LangGraph Cognitive Engine

The agent's cognitive architecture is structured as a stateful cyclic graph built on the LangGraph framework. To maintain execution context across iterative reasoning cycles, the agent's short-term memory is represented by a Python `TypedDict` state container that accumulates the message history, the file path under investigation, and the raw error payload extracted from the dbt artifacts across each reasoning step. This stateful structure supports the agent's workflow through three primary execution phases:

- **Profiling and State Initialization**: Upon activation, the state machine executes a memory initialization node that parses `run_results.json` to extract the failing dbt model identifier, the compiled SQL query, and the raw Spark exception string. Execution then transitions to the Profiling node, which invokes a structured LLM call via the `ChatDatabricks.with_structured_output()` interface. By enforcing a strict Pydantic output schema, the agent deterministically maps the error against the IBM failure taxonomy defined in Chapter 4. If the classifier detects an infrastructural anomaly (e.g., `category = 'Infrastructure'` or `category = 'Access'`), a conditional edge triggers a controlled early exit, concluding the LangGraph execution and notifying human engineers without attempting codebase modifications.
- **Planning and ReAct Execution**: For data quality, schema, or logic errors, the state machine enters the ReAct decision cycle introduced in Section 4.4. Within LangGraph, this is implemented as a cyclic loop driven by conditional edges: a central LLM node reasons about the root cause, selects and invokes an appropriate tool from the bound toolset. If a tool call is generated, the graph routes execution to the Action node, and once the tool output is returned as an observation, execution loops back to the LLM node for the next reasoning step. To ensure bounded execution and computational cost, the graph enforces a strict iteration limit of `MAX_ITERATIONS = 10`.
- **Action and Tool Binding**: The agent interacts with the Databricks environment exclusively through six explicit Python tools, connected to the LLM via the `bind_tools()` API. The toolset includes both a full-file overwrite tool (`write_local_file`) and a targeted diff-style patch tool (`patch_local_file`), the latter introduced to reduce the risk of truncation or unsupported modifications when rewriting lengthy SQL files. Consistent with the Bounded Autonomy principle discussed in Section 4.4, the toolset is strictly limited to local file operations and isolated sandbox commands, preventing unconstrained code execution outside the defined workspace boundaries. The complete tool specification is reported in Table 5.2.

5.2.3 Architectural Evolution: v1.0 vs. v1.1

During the engineering of the self-healing framework, initial prototyping of the baseline architecture (v1.0) exposed a recurring limitation observed in autonomous LLM coding agents: a restricted edit scope that fails to account for inter-file dependencies [43]. This pattern manifests when an agent applies a superficial workaround to a downstream symptom rather than tracing the failure upstream across the execution DAG to resolve the root cause [40]: when an upstream schema

Table 5.2: Action Toolset Scoping and Execution Contract.

Tool	Input Parameters	Description
`list_local_directory`	`dir_path`	Lists files and subdirectories within a local folder; returns directory structure or missing path error.
`read_local_file`	`file_path`	Reads target SQL, Jinja, or YAML file contents; returns full text or missing file error.
`write_local_file`	`file_path`, `new_content`	Overwrites or creates a local file with complete code contents; returns modification confirmation or I/O error.
`patch_local_file`	`file_path`, `old_text, new_text`	Performs exact string replacement of specific lines in a local file; returns success confirmation or line mismatch error.
`run_sandbox_test`	`model_name`	Executes `dbt build` in isolated `sandbox` schema; returns `SUCCESS` or raw `stderr/stdout` compilation stack trace.
`create_gitlab_mr`	`branch_name`, `commit_msg, title`	Commits the validated patch to a new Git branch and opens a GitLab Merge Request; returns MR web URL for human review.

modification broke downstream transformation contracts, v1.0 frequently modified only the downstream SQL model in isolation, submitting a Merge Request that satisfied isolated sandbox compilation while leaving the upstream defect unresolved for all other consuming models in the DAG. The specific failure scenarios that exposed this vulnerability are detailed in Section 5.3; this subsection documents the three architectural guardrails introduced in v1.1 to address it:

- **Bidirectional Sandbox Isolation**: This vulnerability is inherent to multi-model DAG architectures, where a breaking schema change in an upstream Silver model propagates to a downstream Gold model. In framework v1.0, the sandbox validation tool validated only the failing downstream model and its upstream dependencies using the dbt selection `+gold_model`. Because

the `+model` selector selects the target model and its upstream parents, v1.0 validated the Gold model together with its upstream lineage, but without its downstream consumers, allowing the agent to apply a local workaround—such as creating a column alias or hardcoding a fallback value. The sandbox returned a `SUCCESS` status, prompting the agent to open a Merge Request that left the upstream Silver model broken for all other consuming tables in the DAG. To enforce lineage-wide integrity, framework v1.1 upgraded the validation engine to bidirectional selection (`dbt build -select +silver_model+ -indirect-selection cautious -t sandbox`). The `+model+` syntax forces dbt to compile upstream parents, the target model, and all downstream children, while the `-indirect-selection cautious` flag prevents false-positive test failures on unbuilt external entities. Under v1.1, if the agent attempts a local downstream patch, the sandbox compilation of Silver and its dependent children fails, structurally forcing the agent to trace the defect upstream and resolve the root cause at the source.

- **Root-Cause Analysis System Prompt Guardrail**: To reinforce the structural constraints of the bidirectional sandbox at the reasoning level, the agent's system prompt was updated with explicit lineage navigation constraints. In framework v1.0, when presented with a failure log, the agent frequently attempted an immediate patch on the file reported in the stack trace. In v1.1, the system prompt explicitly mandates that, prior to attempting any local code edit, the agent must inspect the upstream `ref()` parent model, identify the origin of the schema drift in the source file, and execute `run_sandbox_test` directly on the upstream model. This prompt-level guardrail complements the structural constraints of the bidirectional sandbox by aligning the agent's initial reasoning strategy with its validation scope, preventing premature patch attempts on downstream files.

- **Stderr Fallback Parser**: In framework v1.0, if a SQL syntax error or Jinja template compilation failure prevented `dbt Core` from emitting a valid `run_results.json` diagnostic artifact, the agent halted due to an unreadable diagnostic payload. Framework v1.1 resolved this limitation by introducing a fallback parser within `error_parser.py`. When `run_results.json` is missing or corrupted, the parser scans raw `stderr` output, extracting the Spark exception line and target file path to reconstruct a structured diagnostic payload for the LLM.

Together, these three guardrails constitute the complete v1.1 architecture evaluated in subsequent sections.

5.3 Experimental Setup and Benchmark Design

Systematically evaluating an autonomous self-healing agent within an enterprise DataOps platform requires an experimental framework that reflects the multi-component nature of production data pipelines. Established code generation benchmarks such as HumanEval [61] focus on isolated Python function synthesis from docstrings, measuring function-level code generation rather than system-level debugging. Similarly, repository-level benchmarks like SWE-bench [43] evaluate general software bug fixing within traditional Python codebases, without accounting for Spark SQL execution semantics, declarative dbt data quality contracts, or platform governance rules.

To address this evaluation gap, a dedicated benchmark suite comprising 15 production-inspired failure scenarios was developed specifically for the platform deployed on Databricks and governed through Unity Catalog. The suite operationalizes the adapted and extended IBM failure taxonomy introduced in Chapter 4, enabling a structured comparative evaluation across multiple Large Language Models.

5.3.1 Benchmark Taxonomy and Failure Scenarios

The 15 benchmark scenarios are structured across six levels of increasing complexity, progressing from isolated single-file syntax errors to multi-file macro refactoring, as detailed in Table 5.3.

This structure evaluates five key operational capabilities: interpreting Spark SQL compilation and runtime errors, navigating DAG dependencies to locate root causes across models, enforcing data quality contracts without modifying test assertions, rectifying silent domain logic errors, and executing multi-file macro refactoring while preserving DRY principles.

5.3.2 Candidate Language Models

To systematically evaluate the self-healing framework across the 15 failure scenarios, three foundation models representing different model scales and architectures were benchmarked from version v1.0 onwards. The primary objective of this empirical comparison is to validate the effectiveness of the architectural guardrails and ReAct workflow across versions v1.0 and v1.1, rather than to benchmark raw LLM leaderboard performance:

- **Claude Opus 4.8**: Hosted via the `databricks-claude-opus-4-8` serving endpoint, this high-capacity model served as the reference foundation model available within the Data Reply enterprise sandbox environment during initial

Table 5.3: Benchmark Test Suite Taxonomy across 15 Failure Scenarios.

ID	Injected Fault & Objective
Level 1: Single-File SQL Syntax & Runtime Errors	
`L1-A`	Missing comma; tests Spark SQL `ParseException` repair.
`L1-B`	Omitted `END` in `CASE WHEN`; tests parser error recovery.
`L1-C`	Division by zero; tests runtime exception handling.
Level 2: Cross-File Invalid References	
`L2-A`	Invalid column name; tests upstream Silver schema tracing.
`L2-B`	Typo in Jinja `ref()`; tests dbt model dependency resolution.
Level 3: Data Contracts & Governance SLAs	
`L3-A`	Spurious `LEFT JOIN`; tests `not_null` fix without contract edits.
`L3-B`	Invalid win points (4 vs 3); tests `accepted_values` contract fix.
`L3-C`	String cast in Silver model; tests Gold contract root-cause resolution.
`L3-D`	Freshness SLA breach in `target/sources.json`; tests non-code remediation via `write_local_file`.
Level 4: Cascading Dependencies & Multi-Hop DAGs	
`L4-A`	Upstream column rename; tests multi-turn ReAct regression handling.
`L4-B`	Alias rename breaking Gold stats; tests upstream DAG navigation.
`L4-C`	Multi-hop Silver → Gold cascade; tests bidirectional sandbox (`+model+`).
Level 5: Silent Domain Logic Bugs	
`L5-A`	Win score set to 2 points; tests soccer domain logic correction.
`L5-B`	Goal difference calculated as addition; tests domain math logic.
Level 6: Multi-File Macro Refactoring	
`L6-B`	Jinja macro interface change; tests multi-file DRY patch execution.

prototyping, providing a stable, high-capacity reference baseline across the full benchmark suite.

- **Claude Haiku 4.5**: Hosted via the `databricks-claude-haiku-4-5` serving endpoint, this lightweight model was evaluated alongside Opus from framework v1.0 onwards to provide a direct comparison. Its inclusion in the benchmark assesses whether a high-speed model optimized for low latency and lower inference costs can achieve reliable multi-step DataOps remediation when guided by structured architectural guardrails.
- **Qwen3.5 122B A10B**: Hosted on Databricks Model Serving as `databricks-qwen35-122b-a10b`, this Mixture-of-Experts model featuring 122 billion total parameters with approximately 10 billion active parameters was evaluated from framework v1.0 onwards to test whether an open-weight model can deliver effective self-healing capabilities within an isolated enterprise DataOps

environment.

5.3.3 Evaluation Metrics

Adapting established evaluation protocols from software engineering agent literature, each benchmark run is evaluated across six objective metrics combining automated execution telemetry and qualitative code analysis.

1. Merge Request Creation Rate (%) Altering the execution-based resolution protocol from SWE-bench [43], this metric calculates the percentage of benchmark runs where the agent successfully passes isolated sandbox compilation and triggers the `create_gitlab_mr` action:

$$\text{MR Creation Rate (\%)} = \left(\frac{N_{\text{MR Created}}}{N_{\text{Total Runs}}} \right) \times 100 \tag{5.1}$$

2. Root Cause Resolution Rate (%) Following the patch evaluation framework of Repairnator [62], which explicitly distinguishes test-suite adequate patches from overfitting workarounds, this qualitative engineering metric evaluates whether the generated patch resolves the underlying root cause in upstream Silver or Gold models without introducing superficial workarounds:

$$\text{Root Cause Resolution Rate (\%)} = \left(\frac{N_{\text{Root Cause Fixed}}}{N_{\text{Total Runs}}} \right) \times 100 \tag{5.2}$$

3. ReAct Reasoning Turns Following the ReAct agent framework [41], where an agent interacts with an environment through interleaved thought, action, and observation steps, this metric measures search efficiency as the average number of turns S_i required to reach termination. In this context, a single turn represents a complete ReAct iteration comprising an LLM reasoning step (Thought and Tool Call decision) followed by the environment execution response (Observation):

$$\text{Mean Turns} = \frac{1}{N_{\text{Total Runs}}} \sum_{i=1}^{N_{\text{Total Runs}}} S_i \tag{5.3}$$

4. Token Consumption and Estimated Inference Cost ($) Following SWE-agent [54], this metric measures input prompt tokens T_{in} and output completion tokens T_{out} via LangSmith telemetry [63], computing average financial inference cost per run based on published API token rates P_{in} and P_{out} for each model family [64, 65]:

$$\text{Cost (\$)} = \left(\frac{T_{\text{in}}}{10^6} \times P_{\text{in}} \right) + \left(\frac{T_{\text{out}}}{10^6} \times P_{\text{out}} \right) \tag{5.4}$$

5. Remediation Latency (MTTR) Following the response time metric established for automated repair bots [62], this metric measures Mean Time to Recovery as the wall-clock execution time in seconds from initial pipeline failure trigger t_0 to Merge Request creation t_{MR}:

$$\text{MTTR (s)} = t_{\text{MR}} - t_0 \tag{5.5}$$

6. Action Execution Success Rate (%) Adapting the admissible action metric from InterCode [66], this metric measures tool binding reliability as the percentage of ReAct tool calls executed with valid syntax and non-hallucinated arguments:

$$\text{Action Execution Success Rate (\%)} = \left(\frac{N_{\text{Valid Tool Calls}}}{N_{\text{Total Tool Calls}}}\right) \times 100 \tag{5.6}$$

5.4 Quantitative and Qualitative Benchmark Results

Building upon the Medallion DataOps infrastructure, fault injection scenarios, and metric definitions established in Section 5.3, this section presents the empirical evaluation of the self-healing framework across all 90 benchmark runs (15 scenarios $\times$ 3 candidate models $\times$ 2 framework iterations). Subsection 5.4.1 presents the aggregate quantitative performance results, while Subsection 5.4.2 interprets the underlying architectural mechanisms, qualitative operational patterns, and scope limitations.

5.4.1 Quantitative Performance Metrics

Table 5.4 summarizes the quantitative evaluation metrics across all 90 benchmark runs (detailed scenario-level logs are provided in Appendix A). Overall, the empirical results showed a clear improvement in remediation quality between framework iterations: while v1.0 achieved high raw completion rates, v1.1, through structured architectural guardrails, systematically converted superficial execution workarounds into true root-cause engineering repairs.

A main observation emerging from Table 5.4 is the significant shift in remediation quality between framework iterations. In framework v1.0, baseline models exhibited high raw Merge Request Creation Rates (`MR CR` $\geq 93.3\%$), yet a significant proportion of these executions represented superficial *Workarounds*—accounting for 33.3% of Haiku runs and 40.0% of Qwen runs. In these instances, agents satisfied isolated sandbox compilation by applying localized SQL aliases or relaxing YAML validation thresholds without addressing the upstream fault. The introduction of

Table 5.4: Comprehensive Benchmark Performance Summary across 15 Failure Scenarios per Model (v1.0 vs. v1.1). *MR CR* = Merge Request Creation Rate (%); *RCRR* = Root Cause Resolution Rate (%); *Turns* = Mean reasoning steps; *MTTR* = Mean remediation latency (s); *Cost* = Estimated mean inference cost per scenario ($).

Model	**Ver.**	**MR CR**	**RCRR**	**Turns**	**MTTR**	**Cost**
Claude Opus 4.8	v1.0	93.3	73.3	7.67	77.35	0.2715
Claude Opus 4.8	v1.1	100.0	93.3	6.33	95.67	0.2701
Claude Haiku 4.5	v1.0	100.0	66.7	6.33	61.96	0.0377
Claude Haiku 4.5	v1.1	100.0	86.7	6.27	88.10	0.0490
Qwen3.5 122B A10B	v1.0	100.0	60.0	5.40	58.49	0.0114
Qwen3.5 122B A10B	v1.1	93.3	73.3	7.47	105.19	0.0205

framework v1.1 successfully redirected agent behavior toward *True Root Cause Resolution*, increasing the RCRR by +20.0 percentage points for both Claude Haiku 4.5 (reaching 86.7%) and Claude Opus 4.8 (reaching 93.3%), while improving Qwen3.5 122B by +13.3 percentage points (reaching 73.3%).

The benchmark data further revealed a clear trade-off between diagnostic accuracy and financial inference costs: Claude Opus 4.8 under v1.1 established the highest absolute resolution rate (93.3% RCRR) with zero failed runs, operating at an estimated mean inference cost of $0.2701 per scenario. Conversely, Claude Haiku 4.5 achieved an 86.7% RCRR under v1.1 at $0.0490 per scenario—operating at approximately one-fifth of the inference cost of Claude Opus 4.8 for a modest 6.6 percentage point trade-off in resolution quality. Meanwhile, Qwen3.5 122B demonstrated the viability of open-weights models within private enterprise environments, delivering a 73.3% RCRR at an estimated inference cost of $0.0205 per scenario, calculated using publicly available OpenRouter pricing as a reference [65].

In terms of operational speed, mean remediation latency for all models remained below two minutes. Under framework v1.1, mean MTTR ranged from 88.10 seconds for Claude Haiku 4.5 (6.27 mean ReAct turns) to 95.67 seconds for Claude Opus 4.8 (6.33 mean turns), with Qwen3.5 122B averaging 105.19 seconds (7.47 mean turns) due to additional ReAct iterations required for lineage exploration. Compared to manual DataOps incident triage—which can involve substantial engineering effort for root-cause diagnosis, local testing, and Merge Request submission—the autonomous framework achieved a substantial reduction in recovery time while maintaining schema compliance.

5.4.2 Discussion and Qualitative Analysis

This subsection interprets the results through the lens of the underlying architectural mechanisms, examining how individual guardrail components shaped agent behavior across failure categories and model families. It further identifies emergent qualitative patterns, model-specific limitations, and cost-efficiency trade-offs that are not directly captured by the aggregate metrics in Table 5.4.

Architectural Guardrail Impact The structural changes introduced in framework v1.1 are consistent with the observed quality gains. First, **Bidirectional Sandbox Selection (`+model+`)** prevented localized patches from passing validation in the tested Level 4 scenarios. Under v1.0, single-model selection (`dbt build -select +gold_model`) validated only downstream Gold models, allowing agents to pass by adding local aliases while leaving Silver models broken upstream. Under v1.1, inspecting the full lineage (`+silver_model+`) caused isolated Gold patches to fail downstream validation, compelling agents to repair the Silver root cause directly. Second, **RCA System Prompting** constrained reasoning by requiring upstream `ref()` inspection via `read_local_file` prior to code generation. Third, **Stderr Fallback Parsing** enabled operational resiliency when compilation errors prevented creation of `run_results.json`. Notably, on scenario `L4-C`, this mechanism allowed Claude Opus 4.8 to recover from a Pydantic schema validation failure that had caused a tool-call validation loop; this failure had let the model exceed the maximum iteration limit in v1.0, successfully resolving the incident in 6 turns under v1.1.

Qualitative Behavioral Patterns Path analysis revealed two prominent operational patterns across the benchmark suite. In Level 6 macro scenarios (`L6-B`), v1.0 models frequently eliminated Jinja macros to bypass syntax errors, whereas v1.1 models preserved modular abstractions (DRY principles) by refactoring macro parameters and caller signatures across multi-file Merge Requests. Conversely, scenario `L3-D` (source freshness SLA breach) revealed a persistent behavioral pattern we term *Governance SLA Relaxation Bias*: across all models, agents systematically widened YAML freshness thresholds to force a sandbox `SUCCESS` status rather than initiating policy-compliant non-code escalation procedures. Additionally, Qwen3.5 122B exhibited a model-specific tool-adherence syntax limitation under v1.1, accounting for its single `FAILED` scenario (`L2-B`) and a lower Action Execution Success Rate (83.02% vs. 93.88% for Opus) due to repeated file-path syntax errors during Merge Request tool binding. Execution logs and SQL diffs for a representative failure case (`L6-B`) are detailed in Appendix B.

Cost-Efficiency and Model Selection The benchmark results provide empirical guidance for selecting agent LLMs based on operational priorities. Claude Haiku 4.5 emerged as the most cost-efficient option in this benchmark for high-volume automated remediation, achieving an 86.7% RCRR under v1.1 at an estimated mean inference cost of $0.0490 per scenario—delivering near-Opus quality at approximately one-fifth of the estimated inference cost of Claude Opus 4.8 ($0.2701). Claude Opus 4.8 appears most suitable for critical core pipelines where resolution accuracy is required (93.3% RCRR, zero failures) and cost is secondary. For enterprise environments subject to strict privacy or data-sovereignty policies, Qwen3.5 122B A10B provides a viable open-weights alternative that can, in principle, be self-hosted entirely within an organization's own infrastructure, without reliance on a third-party model provider, achieving a 73.3% RCRR at an estimated mean inference cost of $0.0205 per scenario. Since inference for all evaluated models was performed through Databricks-hosted endpoints, the benchmark did not require transmitting schema or data artifacts to external commercial APIs.

Methodological Limitations Two key methodological boundaries should be considered when interpreting the empirical findings:

- **Transformation-Layer Scope**: The agent's operational toolset is bounded strictly to dbt transformation artifacts (SQL models, Jinja macros, YAML contracts). Consequently, the evaluation targets pipeline data logic and schema governance faults, rather than infrastructure-level cloud failures such as cluster configuration or IAM permission drifts.
- **Single-Run Inference Budget Constraints**: Due to financial inference costs across commercial LLM endpoints, each benchmark scenario was evaluated once per model-version pair (with infrastructure-level retries recorded where triggered). As a result, stochastic variance across repeated LLM sampling cannot be formally quantified.

Chapter 6

Conclusions and Future Work

This concluding chapter summarizes the primary engineering contributions of this thesis, reflects on the architectural findings gained from developing a Self-Healing DataOps platform, and outlines concrete directions for future research in LLM-assisted data platform governance.

6.1 Main Contributions

The main objective of this thesis was to bridge the gap between enterprise DataOps infrastructure and LLM-assisted remediation, directly targeting the bottleneck of extended Mean Time to Recovery (MTTR) in production data pipelines. The proposed framework realizes this objective through four key contributions:

1. **Domain-Driven DataOps Infrastructure with Integrated Quality and Observability**: designed and deployed an end-to-end, three-tier Medallion architecture on Databricks, governed through Unity Catalog using the European Soccer Database as an experimental baseline. Following Domain-Driven Design (DDD) principles, infrastructure deployment is declared as code (IaC) through Databricks Asset Bundles (DABs), while data transformations and validation suites run natively in dbt. The pipeline enforces strict environment segregation (development, testing, sandbox, production) alongside automated data quality testing against core KPIs—completeness, validity, freshness, and consistency. In addition, Slowly Changing Dimensions (SCD Type 2) snapshots preserve historical traceability, while Point-in-Time Joins prevent temporal data leakage. Complementing this setup, LangSmith integration yields granular observability across agent execution.

2. **In-Job Self-Healing LLM Engine**: integrated an automated remediation agent directly into the pipeline orchestration workflow using the LangGraph state machine framework. Executed conditionally upon Databricks job failure (`run_if: AT_LEAST_ONE_FAILED`), the agent operates with zero LLM inference cost during successful runs. Once triggered, it identifies root causes from local execution logs, validates candidate patches within an isolated Sandbox environment, and submits verified fixes as GitLab Merge Requests to guarantee Human-in-the-Loop oversight before any production rollout.

3. **Architectural Guardrails for Upstream Root-Cause Resolution**: resolved a key vulnerability of autonomous LLM agents—the trend to apply superficial local workarounds to downstream surface models. The framework integrates structural guardrails, including bidirectional sandbox selection (`+model+`), root-cause analysis system prompting, and fallback error parsing, guiding the Self-Healing LLM agent to traverse pipeline DAG lineage upstream and target faults at their root cause.

4. **Empirical Evaluation and Multi-Model Benchmarking**: evaluated the Self-Healing framework across a structured benchmark suite of production failure scenarios. This empirical assessment examined multiple foundation models (Claude Haiku 4.5, Claude Opus 4.8, and Qwen3.5 122B A10B) across two framework iterations, quantifying performance in terms of resolution quality, reasoning efficiency, MTTR reduction, and financial inference costs.

6.2 Empirical Key Takeaways

The empirical evaluation presented in Chapter 5 produced three main quantitative takeaways regarding the effectiveness of LLM-assisted self-healing.

First, **architectural guardrails improve resolution accuracy**: moving from an unconstrained baseline (v1.0) to a guardrailed architecture (v1.1) systematically reduced superficial workaround behaviors. By enforcing bidirectional sandbox validation and explicitly guiding lineage traversal, the framework achieved a +20.0 percentage point improvement in Root Cause Resolution Rate (RCRR) for both Claude Opus 4.8 and Claude Haiku 4.5.

Second, **high-capacity models achieve high resolution accuracy**: under framework v1.1, Claude Opus 4.8 established the benchmark ceiling, achieving a 93.3% RCRR across the 15 production scenarios, showing that autonomous agents can reliably resolve complex multi-hop DAG failures when properly constrained by deterministic engineering boundaries.

Third, **significant cost-efficiency trade-offs emerge**: Claude Haiku 4.5

achieved an 86.7% RCRR at a mean inference cost of $0.0490 per scenario—approximately one-fifth of the estimated cost of Claude Opus 4.8 ($0.2701).

6.3 Limitations and Future Work

While the implemented Self-Healing framework demonstrated promising results within the evaluated scope, several methodological boundaries and operational limitations motivate the following directions for future research:

- **Fully Decentralized Data Mesh and Multi-Team Federation**: the current implementation models four bounded contexts within a single governed pipeline infrastructure. Thus, future research should extend this design toward a fully decentralized Data Mesh architecture aligned with multi-team enterprise organizations. Under this architecture, autonomous data product teams would maintain independent dbt repositories and dedicated Self-Healing LLM agents, collaborating asynchronously through schema contracts to coordinate the resolution of cross-domain pipeline failures.
- **Domain-Specific Fine-Tuning of LLMs for DataOps**: although general-purpose foundation models performed effectively without domain-specific fine-tuning, open-weights models like Qwen3.5 122B exhibited tool-adherence limitations (83.02% Action Execution Success Rate). Fine-tuning these models on specialized datasets of dbt compiler logs, execution traces, and enterprise SQL diffs represents a primary future direction that could produce specialized LLMs optimized for DataOps remediation and suitable for privacy-sensitive enterprise deployments.

6.4 Final Remarks

The integration of Large Language Models with declarative DataOps practices marks a meaningful contribution to data platform engineering. By complementing manual, reactive incident troubleshooting with automated, bounded-autonomy self-healing, data engineering teams may substantially reduce manual triage effort and remediation latency. The reference architecture established in this thesis demonstrates that when AI reasoning is constrained by rigorous engineering guardrails and Human-in-the-Loop governance, self-healing data platforms can be both technically feasible and operationally viable within the evaluated transformation boundaries.

Appendix A

Full Per-Scenario Benchmark Results

This appendix presents the complete per-scenario execution data for all 90 benchmark runs (15 fault scenarios × 3 candidate models × 2 framework iterations). The tables report the raw execution-level results underlying the aggregate metrics presented in Section 5.4.1, including remediation status, resolution classification, ReAct turns, remediation latency, and estimated inference cost. The reported cost values are estimates derived from publicly available pricing rates, while all model inference was executed within Databricks.

A.1 Claude Opus 4.8 Benchmark Results

Table A.1: Claude Opus 4.8 — Baseline Version v1.0.

ID	Status	Classification	Turns	MTTR (s)	Cost ($)
L1-A	RESOLVED	Root Cause	4	72.94	0.1244
L1-B	RESOLVED	Root Cause	4	54.95	0.1338
L1-C	RESOLVED	Root Cause	6	62.97	0.2367
L2-A	RESOLVED	Root Cause	8	61.23	0.2181
L2-B	RESOLVED	Root Cause	9	69.74	0.3041
L3-A	RESOLVED	Root Cause	7	60.63	0.1907
L3-B	RESOLVED	Root Cause	4	57.37	0.1167
L3-C	RESOLVED	Workaround	19	152.99	0.7408
L3-D	RESOLVED	Workaround	4	41.78	0.1267
L4-A	RESOLVED	Workaround	8	73.26	0.2443
L4-B	RESOLVED	Root Cause	6	63.51	0.2225
L4-C	FAILED	FAILED	20	135.67	0.5973
L5-A	RESOLVED	Root Cause	4	59.85	0.1273
L5-B	RESOLVED	Root Cause	4	83.11	0.2120
L6-B	RESOLVED	Root Cause	8	110.30	0.4778

Table A.2: Claude Opus 4.8 — Guardrailed Version v1.1.

ID	Status	Classification	Turns	MTTR (s)	Cost ($)
L1-A	RESOLVED	Root Cause	5	90.72	0.1722
L1-B	RESOLVED	Root Cause	4	78.91	0.1405
L1-C	RESOLVED	Root Cause	6	90.92	0.2494
L2-A	RESOLVED	Root Cause	8	140.58	0.3862
L2-B	RESOLVED	Root Cause	7	124.51	0.3471
L3-A	RESOLVED	Root Cause	8	90.89	0.2362
L3-B	RESOLVED	Root Cause	6	88.03	0.2346
L3-C	RESOLVED	Root Cause	11	107.86	0.4975
L3-D	RESOLVED	Workaround	4	43.19	0.1315
L4-A	RESOLVED	Root Cause	5	94.42	0.1720
L4-B	RESOLVED	Root Cause	7	96.11	0.2874
L4-C	RESOLVED	Root Cause	6	103.69	0.2957
L5-A	RESOLVED	Root Cause	6	96.42	0.2456
L5-B	RESOLVED	Root Cause	4	81.88	0.1582
L6-B	RESOLVED	Root Cause	8	106.90	0.4969

A.2 Claude Haiku 4.5 Benchmark Results

Table A.3: Claude Haiku 4.5 — Baseline Version v1.0.

ID	Status	Classification	Turns	MTTR (s)	Cost ($)
L1-A	RESOLVED	Root Cause	6	54.09	0.0378
L1-B	RESOLVED	Root Cause	4	48.62	0.0222
L1-C	RESOLVED	Root Cause	6	52.20	0.0385
L2-A	RESOLVED	Root Cause	8	55.79	0.0367
L2-B	RESOLVED	Root Cause	9	75.90	0.0465
L3-A	RESOLVED	Root Cause	4	38.41	0.0197
L3-B	RESOLVED	Root Cause	6	58.13	0.0384
L3-C	RESOLVED	Workaround	6	56.57	0.0376
L3-D	RESOLVED	Workaround	4	26.73	0.0179
L4-A	RESOLVED	Workaround	14	131.91	0.0768
L4-B	RESOLVED	Root Cause	6	52.77	0.0367
L4-C	RESOLVED	Workaround	8	55.39	0.0434
L5-A	RESOLVED	Root Cause	6	58.94	0.0362
L5-B	RESOLVED	Root Cause	4	92.17	0.0345
L6-B	RESOLVED	Workaround	4	71.81	0.0431

Table A.4: Claude Haiku 4.5 — Guardrailed Version v1.1.

ID	Status	Classification	Turns	MTTR (s)	Cost ($)
L1-A	RESOLVED	Root Cause	6	88.12	0.0390
L1-B	RESOLVED	Root Cause	4	73.82	0.0234
L1-C	RESOLVED	Root Cause	6	76.38	0.0401
L2-A	RESOLVED	Root Cause	5	79.62	0.0267
L2-B	RESOLVED	Root Cause	7	98.51	0.0496
L3-A	RESOLVED	Root Cause	4	95.37	0.0249
L3-B	RESOLVED	Root Cause	6	76.99	0.0393
L3-C	RESOLVED	Root Cause	9	138.60	0.0735
L3-D	RESOLVED	Workaround	4	52.32	0.0196
L4-A	RESOLVED	Root Cause	5	76.89	0.0267
L4-B	RESOLVED	Root Cause	6	75.81	0.0405
L4-C	RESOLVED	Workaround	14	156.26	0.1806
L5-A	RESOLVED	Root Cause	6	74.18	0.0394
L5-B	RESOLVED	Root Cause	4	73.22	0.0285
L6-B	RESOLVED	Root Cause	8	85.46	0.0827

A.3 Qwen3.5 122B A10B Benchmark Results

Table A.5: Qwen3.5 122B A10B — Baseline Version v1.0.

ID	Status	Classification	Turns	MTTR (s)	Cost ($)
L1-A	RESOLVED	Root Cause	4	48.93	0.0060
L1-B	RESOLVED	Root Cause	4	50.88	0.0072
L1-C	RESOLVED	Root Cause	4	48.36	0.0070
L2-A	RESOLVED	Root Cause	4	47.97	0.0069
L2-B	RESOLVED	Root Cause	14	75.84	0.0220
L3-A	RESOLVED	Workaround	4	38.17	0.0059
L3-B	RESOLVED	Root Cause	4	50.87	0.0089
L3-C	RESOLVED	Workaround	6	58.90	0.0114
L3-D	RESOLVED	Workaround	4	29.00	0.0079
L4-A	RESOLVED	Workaround	8	86.96	0.0202
L4-B	RESOLVED	Root Cause	6	51.97	0.0114
L4-C	RESOLVED	Workaround	7	94.25	0.0193
L5-A	RESOLVED	Root Cause	4	53.67	0.0083
L5-B	RESOLVED	Root Cause	4	69.84	0.0134
L6-B	RESOLVED	Workaround	4	71.70	0.0144

Table A.6: Qwen3.5 122B A10B — Guardrailed Version v1.1.

ID	Status	Classification	Turns	MTTR (s)	Cost ($)
L1-A	RESOLVED	Root Cause	4	70.98	0.0065
L1-B	RESOLVED	Root Cause	4	73.45	0.0072
L1-C	RESOLVED	Root Cause	4	69.32	0.0071
L2-A	RESOLVED	Root Cause	9	115.27	0.0242
L2-B	FAILED	FAILED	15	177.15	0.0352
L3-A	RESOLVED	Workaround	4	70.27	0.0080
L3-B	RESOLVED	Root Cause	5	74.45	0.0100
L3-C	RESOLVED	Root Cause	14	143.07	0.0426
L3-D	RESOLVED	Workaround	5	29.91	0.0097
L4-A	RESOLVED	Root Cause	5	76.71	0.0124
L4-B	RESOLVED	Root Cause	7	72.36	0.0152
L4-C	RESOLVED	Root Cause	9	140.25	0.0310
L5-A	RESOLVED	Root Cause	5	75.51	0.0110
L5-B	RESOLVED	Root Cause	4	76.24	0.0110
L6-B	RESOLVED	Workaround	18	312.86	0.0770

Appendix B

Case Study: Multi-File Jinja Refactoring

This appendix provides a qualitative deep-dive into a representative benchmark scenario to illustrate the behavioral shift between framework versions v1.0 and v1.1.

Specifically, this case study analyzes **Scenario L6-B** (*Multi-File Jinja Macro Refactoring*), executed under **Claude Haiku 4.5**, that evaluates how architectural guardrails reduce agent workarounds and preserve modular software engineering principles during DataOps incident remediation.

B.1 Scenario Description and Pre-Remediation Baseline

Scenario `L6-B` introduces an interface signature drift within a reusable Jinja macro across the analytical transformation layer. The macro `macro_calc_goal_difference.sql`, located in the `macros/` directory, calculates team goal differentials:

```
{% macro macro_calc_goal_difference(scored_col, conceded_col) %}
    ({{ scored_col }} - {{ conceded_col }})
{% endmacro %}
```

Listing B.1: Original macro definition (`macros/macro_calc_goal_difference.sql`) accepting only positional parameters `scored_col` and `conceded_col`.

Downstream caller models, such as `gold_team_season_stats.sql`, invoke the macro passing keyword arguments (`goals_for`, `goals_against`). This invocation fails because the macro signature accepts only positional parameters (`scored_col`, `conceded_col`), not keyword arguments:

```sql
SELECT
    team_id,
    total_goals_scored,
    total_goals_conceded,
    {{ macro_calc_goal_difference(goals_for='total_goals_scored', goals_against='total_goals_conceded') }} AS goal_difference
FROM {{ ref('silver_team_stats') }}
```

Listing B.2: Caller model (`models/gold/gold_team_season_stats.sql`) invoking the macro with keyword arguments, triggering a Jinja compilation failure.

During `dbt build` compilation, this mismatch triggers a Jinja compilation error in `gold_team_season_stats.sql`: «Compilation Error in model `gold_team_season_stats`: macro `"macro_calc_goal_difference"` takes no keyword argument `"goals_for"`».

B.2 Baseline Framework (v1.0) Execution

Under baseline framework v1.0, Claude Haiku 4.5 operated without Root Cause Analysis (RCA) system prompting or multi-file inspection and modification tools and, as a result, the agent opted for a shortcut: «The simplest fix is to replace the macro call with a direct SQL calculation for goal difference, which is simply: `total_goals_scored - total_goals_conceded`».

To bypass the compilation failure, the agent modified `models/gold/gold_team_season_stats.sql` by replacing the Jinja macro call with an inline subtraction:

```sql
SELECT
    team_id,
    total_goals_scored,
    total_goals_conceded,
-   {{ macro_calc_goal_difference(goals_for='total_goals_scored', goals_against='total_goals_conceded') }} AS goal_difference
+   (total_goals_scored - total_goals_conceded) AS goal_difference
FROM {{ ref('silver_team_stats') }}
```

Listing B.3: Baseline v1.0 patch (`models/gold/gold_team_season_stats.sql`): inline SQL replacement bypassing the macro invocation.

However, the underlying macro file `macros/macro_calc_goal_difference.sql` was left untouched and uncorrected in the codebase:

```sql
-- UNTOUCHED BY AGENT IN V1.0 (Left broken in macros/ directory)
{% macro macro_calc_goal_difference(scored_col, conceded_col) %}
    ({{ scored_col }} - {{ conceded_col }})
{% endmacro %}
```

Listing B.4: Baseline v1.0 macro status (`macros/macro_calc_goal_difference.sql`): macro left uncorrected in the codebase.

Although `dbt build` returned a sandbox `SUCCESS` status (recorded as `RESOLVED` in the execution logs), this execution was qualitatively classified as a **Workaround**.

By deleting the macro invocation to bypass syntax errors, the agent violated software engineering modularity, leaving the macro broken for all other downstream analytical pipelines.

B.3 Guardrailed Framework (v1.1) Execution

Under guardrailed framework iteration v1.1, Claude Haiku 4.5 was governed by Root Cause Analysis (RCA) system prompting, Stderr Fallback Parsing, and multi-file inspection and modification tools. Driven by lineage-aware prompts, the agent recognized the need for a root-cause remediation: «The compilation error shows the macro `macro_calc_goal_difference` is failing... I need to update the macro parameter signature to accept keyword arguments and update the caller model».

To resolve the interface mismatch, the agent refactored the Jinja macro signature in `macros/macro_calc_goal_difference.sql` by adding backward-compatible keyword parameter handling:

```
-{% macro macro_calc_goal_difference(scored_col, conceded_col) %}
+{% macro macro_calc_goal_difference(scored_col=None, conceded_col=None, goals_for
    =None, goals_against=None) %}
+   {% set scored = scored_col if scored_col is not none else goals_for %}
+   {% set conceded = conceded_col if conceded_col is not none else goals_against
    %}
    (({{ scored }} - {{ conceded }})
 {% endmacro %}
```

Listing B.5: Guardrailed v1.1 patch (`macros/macro_calc_goal_difference.sql`): refactored signature adding backward-compatible keyword parameter handling.

With the macro interface repaired, the caller model `models/gold/gold_team_season_stats.sql` was retained with its modular macro invocation intact:

```
SELECT
    team_id,
    total_goals_scored,
    total_goals_conceded,
    {{ macro_calc_goal_difference(goals_for='total_goals_scored', goals_against='
    total_goals_conceded') }} AS goal_difference
FROM {{ ref('silver_team_stats') }}
```

Listing B.6: Guardrailed v1.1 downstream caller (`models/gold/gold_team_season_stats.sql`): existing macro invocation retained after interface refactoring.

This execution was qualitatively classified as a **Root Cause Fix**. Claude Haiku 4.5 preserved modular software architecture, repaired the Jinja macro interface to support downstream consumers, and submitted a validated multi-file Merge Request for human review in 8 turns (MTTR: 85.46s, estimated inference cost: $0.0827).

B.4 Comparative Execution Summary

Table B.1 summarizes the structural differences between framework iterations for Scenario `L6-B` under Claude Haiku 4.5.

Table B.1: Comparative Execution Metrics for Scenario `L6-B` under Claude Haiku 4.5.

Metric / Dimension	**Baseline (v1.0)**	**Guardrailed (v1.1)**
Execution Status	RESOLVED	RESOLVED
Qualitative Outcome	Workaround	Root Cause Fix
ReAct Turns	4 turns	8 turns
MTTR Latency	71.81 s	85.46 s
Estimated Inference Cost	$0.0431	$0.0827
Modified Files	1 file	2 files
DRY Principle	Violated	Preserved

This qualitative comparison highlights the fundamental role of architectural guardrails in production AI DataOps: while unconstrained agents may satisfy automated build checks via superficial workarounds, structured lineage prompts and multi-file validation support true root-cause remediation aligned with enterprise software engineering standards.

Bibliography

[1] Joe Reis and Matt Housley. *Fundamentals of Data Engineering: Plan and Build Robust Data Systems*. O'Reilly Media, 2022. ISBN: 9781098108274. URL: https://books.google.it/books?id=3qd2EAAAQBAJ (cit. on pp. 1, 3, 36).

[2] William H. Inmon. *Building the Data Warehouse*. Wiley, 2005. ISBN: 9780471774235. URL: https://books.google.it/books?id=BT-IBUb-g68C (cit. on p. 1).

[3] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. «Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics». In: *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR)*. 2021. URL: https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf (cit. on pp. 1, 9, 18).

[4] AC3. *The Lakehouse Paradigm: Building Modern Enterprise Data Analytics Platforms with Databricks, AWS, and Azure*. Accessed: 2026-06-09. 2023. URL: https://www.ac3.com.au/resources/the-lakehouse-paradigm-building-modern-enterprise-data-analytics-platforms-with-databricks-aws-and-azure (cit. on p. 2).

[5] Barr Moses, Lior Gavish, and Molly Vorwerck. *Data Quality Fundamentals*. O'Reilly Media, 2022. ISBN: 9781098112011. URL: https://books.google.it/books?id=2q6GEAAAQBAJ (cit. on p. 1).

[6] Erhard Rahm and Hong Do. «Data Cleaning: Problems and Current Approaches». In: *IEEE Data Eng. Bull.* 23 (Jan. 2000), pp. 3–13 (cit. on p. 2).

[7] Carlo Batini and Monica Scannapieco. *Data and Information Quality: Dimensions, Principles and Techniques*. Data-Centric Systems and Applications. Springer International Publishing, 2016. ISBN: 9783319241067. URL: https://books.google.it/books?id=kJ_WCwAAQBAJ (cit. on p. 2).

[8] Aiswarya Raj Munappy, David Issa Mattos, Jan Bosch, Helena Holmstr"om Olsson, and Anas Dakkak. «From Ad-Hoc Data Analytics to DataOps». In: *Proceedings of the International Conference on Software and System Processes (ICSSP '20)*. New York, NY, USA: Association for Computing Machinery,

2020, pp. 165–174. ISBN: 9781450375122. DOI: 10.1145/3379177.3388909 (cit. on p. 3).

[9] Lakshmi Koralla. «DataOps: Revolutionizing Application Development through Data-Centric Methodologies». In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 11 (2025), pp. 2155–2167. DOI: 10.32628/CSEIT23112576 (cit. on pp. 5–8, 16).

[10] Statista. *Data generation volume worldwide 2010-2029*. Accessed on 09 May 2026. 2024. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (cit. on p. 5).

[11] Sarah Azzabi, Zakiya Alfughi, and Abdelkader Ouda. «Data Lakes: A Survey of Concepts and Architectures». In: *Computers* 13 (2024), p. 183. DOI: 10.3390/computers13070183 (cit. on pp. 5, 8, 9, 16–19).

[12] Hosne Mohna, Tonmoy Barua, Mohammad Mohiuddin, and Md Mostafizur Rahman. «AI-READY DATA ENGINEERING PIPELINES: A REVIEW OF MEDALLION ARCHITECTURE AND CLOUD-BASED INTEGRATION MODELS». In: *American Journal of Scholarly Research and Innovation* 01 (2022), pp. 319–350. DOI: 10.63125/51kxtf08 (cit. on pp. 5, 17, 18, 21).

[13] Leonard J. Bass, Ingo M. Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015. ISBN: 978-0-1340-4984-7 (cit. on p. 6).

[14] Julian Ereth. «DataOps – Towards a Definition». In: *LWDA* 2191.104–112 (2018), p. 106 (cit. on p. 6).

[15] Christopher Bergh, Gil Benghiat, and Eran Strod. *The DataOps Cookbook*. 2nd ed. DataKitchen, 2019 (cit. on pp. 6, 7).

[16] Harvinder Atwal. *Practical DataOps: Delivering Agile Data Science at Scale*. Apress, 2020. ISBN: 978-1-4842-5103-4. DOI: 10.1007/978-1-4842-5104-1 (cit. on p. 6).

[17] J. Dixon. *Pentaho, Hadoop, and Data Lakes*. 2010. URL: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/ (cit. on p. 8).

[18] Ruoran Liu, Haruna Isah, and Farhana Zulkernine. «A Big Data Lake for Multilevel Streaming Analytics». In: *2020 1st International Conference on Big Data Analytics and Practices (IBDAP)*. 2020, pp. 1–6. DOI: 10.1109/IBDAP50342.2020.9245460 (cit. on p. 8).

[19] Sarah Benjelloun, Mohamed El Aissi, Younes Lakhrissi, and Safae Ali. «Data Lake Architecture for Smart Fish Farming Data-Driven Strategy». In: *Applied System Innovation* 6 (2023), p. 8. DOI: 10.3390/asi6010008 (cit. on p. 8).

[20] Julia Couto, Olimar Borges, Duncan Ruiz, Sabrina Marczak, and Rafael Prikladnicki. «A Mapping Study about Data Lakes: An Improved Definition and Possible Architectures». In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE 2019)*. 2019, pp. 453–458. DOI: 10.18293/SEKE2019-129 (cit. on p. 8).

[21] E Zagan and M Danubianu. «From Data Warehouse to a New Trend in Data Architectures–Data Lake». In: *IJCSNS International Journal of Computer Science and Network Security* 19.3 (2019), pp. 30–35 (cit. on p. 8).

[22] Weiming Sheng, Jinlang Wang, Manuel Barros, Aldrin Montana, Jacopo Tagliabue, and Luca Bigon. *Building a Correct-by-Design Lakehouse. Data Contracts, Versioning, and Transactional Pipelines for Humans and Agents*. 2026. arXiv: 2602.02335 [cs.DC]. URL: https://arxiv.org/abs/2602.02335 (cit. on pp. 9, 21–23, 31).

[23] Z. Dehghani. *Data Mesh: Delivering Data-driven Value at Scale*. O'Reilly, 2022. ISBN: 9781492092391. URL: https://books.google.it/books?id=M5J5zgEACAAJ (cit. on pp. 10, 12, 14, 35, 37, 38).

[24] Abel Goedegebuure, Indika Kumara, Stefan Driessen, Willem-Jan Van Den Heuvel, Geert Monsieur, Damian Andrew Tamburri, and Dario Di Nucci. «Data Mesh: A Systematic Gray Literature Review». In: *ACM Computing Surveys* 57.1 (Oct. 2024), pp. 1–36. ISSN: 1557-7341. DOI: 10.1145/3687301. URL: http://dx.doi.org/10.1145/3687301 (cit. on pp. 10–12, 14, 35).

[25] Gema Correa. *Data Lake vs Data Mesh: understanding the differences and when to use each*. Medium – Artificial Intelligence in Plain English. Accessed: 2026-06-10. 2026. URL: https://ai.plainenglish.io/data-lake-vs-data-mesh-understanding-the-differences-and-when-to-use-each-12117ff66ff3 (cit. on p. 10).

[26] Raju Mandal. *Everything to know about Data Mesh*. Accessed: 2026-06-10. 2024. URL: https://hevoacademy.com/data-management/understanding-data-mesh/ (cit. on p. 12).

[27] Franck Ravat and Yan Zhao. «Data lakes: Trends and perspectives». In: *International Conference on Database and Expert Systems Applications*. Springer. 2019, pp. 304–313 (cit. on pp. 16, 17).

[28] Bismart. *Medallion Architecture: What is it and what is it for?* Accessed: 2026-06-10. 2024. URL: https://blog.bismart.com/en/medallion-architecture (cit. on p. 19).

[29] G.A. Thalpati. *Practical Lakehouse Architecture: Designing and Implementing Modern Data Platforms at Scale*. O'Reilly Media, Incorporated, 2024. ISBN: 9781098153014. URL: https://books.google.it/books?id=Ay2r0AEACAAJ (cit. on p. 19).

[30] Michael Armbrust et al. «Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores». In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3411–3424. ISSN: 2150-8097. DOI: 10.14778/3415478.3415560 (cit. on p. 20).

[31] Harald Foidl, Valentina Golendukhina, Rudolf Ramler, and Michael Felderer. *Data Pipeline Quality: Influencing Factors, Root Causes of Data-related Issues, and Processing Problem Areas for Developers*. 2023. arXiv: 2309.07067 [cs.SE]. URL: https://arxiv.org/abs/2309.07067 (cit. on p. 21).

[32] dbt Labs. *What is dbt?* https://docs.getdbt.com/docs/introduction. Accessed: 2026-06-10. 2024 (cit. on pp. 23–25, 40).

[33] Fivetran. *Data Build Tool (dbt) Explained: Role, Use Case, and Stack Fit*. Accessed: 2026-06-10. 2025. URL: https://www.fivetran.com/blog/dbt-explained (cit. on pp. 23–25).

[34] Analytics8. *DBT Overview: What is dbt and What Can It Do for My Data Pipeline?* Accessed: 2026-06-10. 2026. URL: https://www.analytics8.com/blog/dbt-overview-what-is-dbt-and-what-can-it-do-for-my-data-pipeline/ (cit. on pp. 24, 25).

[35] Bethan Donovan. *DBT - Day 1*. Accessed: 2026-06-10. URL: https://www.thedataschool.co.uk/bethan-donovan/dbt-day-1/ (cit. on p. 24).

[36] Atharva Shrivas and Ashutosh Yesekar. *Streamlining Databricks Deployments with Databricks Asset Bundles (DABs) and GitLab CI/CD*. https://www.freshgravity.com/insights-blogs/streamlining-databricks-deployments/. Accessed: 2026-06-11. 2025 (cit. on pp. 25, 26, 39).

[37] Avin Kohale. *Databricks Asset Bundles – Overview*. Accessed: 2026-06-11. 2024. URL: https://towardsdev.com/databricks-asset-bundles-overview-9cc44a722622 (cit. on pp. 25, 26).

[38] Databricks. *What are Declarative Automation Bundles?* https://docs.databricks.com/en/dev-tools/bundles/index.html. Accessed: 2026-06-11 (cit. on pp. 26, 27, 39, 48).

[39] Soumen Chakraborty. «From DataOps to AIOps: How autonomous agents are revolutionizing data engineering». In: *World Journal of Advanced Engineering Technology and Sciences* 15.2 (2025), pp. 1403–1414. DOI: 10.30574/wjaets.2025.15.2.0650 (cit. on pp. 27, 28).

[40] Deepak Vijaykeerthy et al. «Root-Cause Diagnosis with Remediation Recommendations for ETL Pipelines via LLM-Based Reasoning». In: *Companion of the International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, May 2026, pp. 126–129. ISBN: 9798400724503. DOI: 10.1145/3788853.3801613. URL: https://doi.org/10.1145/3788853.3801613 (cit. on pp. 27–31, 43, 44, 55).

[41] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629 (cit. on pp. 27, 29, 30, 43, 60).

[42] Lei Wang et al. «A survey on large language model based autonomous agents». In: *Frontiers of Computer Science* 18.6 (Mar. 2024). ISSN: 2095-2236. DOI: 10.1007/s11704-024-40231-1. URL: http://dx.doi.org/10.1007/s11704-024-40231-1 (cit. on pp. 27, 30, 41–43, 45).

[43] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL]. URL: https://arxiv.org/abs/2310.06770 (cit. on pp. 31, 55, 58, 60).

[44] Hugo Mathien. *European Soccer Database*. Kaggle Dataset. 2016. URL: https://www.kaggle.com/datasets/hugomathien/soccer (cit. on pp. 32, 34).

[45] Werner Dubitzky, Philippe Lopes, Jesse Davis, and Daniel Berrar. «The Open International Soccer Database for machine learning». In: *Mach. Learn.* 108.1 (Jan. 2019), pp. 9–28. ISSN: 0885-6125. DOI: 10.1007/s10994-018-5726-0. URL: https://doi.org/10.1007/s10994-018-5726-0 (cit. on p. 32).

[46] Maurizio Carpita and Silvia Golia. «Exploring the Kaggle European Soccer database with Bayesian Networks: the case of the Italian League Serie A». In: *Book of Short Papers SIS 2018*. Palermo: Pearson, June 2018, pp. 1–8. ISBN: 9788891910233 (cit. on p. 33).

[47] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. ISBN: 9780321125217 (cit. on p. 35).

[48] Simon Späti. *One Big Table (OBT)*. https://www.ssp.sh/brain/one-big-table/. Accessed: 2026-08-05. 2022 (cit. on p. 39).

[49] Michael Kaminsky. *Star Schema vs. OBT for Data Warehouse Performance*. Accessed: 2026-08-05. 2022. URL: https://www.fivetran.com/blog/star-schema-vs-obt (cit. on p. 39).

[50] SWE-agent Team. *SWE-ReX*. Accessed: 2026-08-05. URL: https://github.com/SWE-agent/SWE-ReX (cit. on p. 42).

[51] Linghao Zhang et al. *SWE-bench Goes Live!* 2025. arXiv: 2505.23419 [cs.SE]. URL: https://arxiv.org/abs/2505.23419 (cit. on p. 42).

[52] GitHub. *GitHub Agentic Workflows: Frequently Asked Questions*. Accessed: 2026-08-05. 2026. URL: https://github.github.com/gh-aw/reference/faq/ (cit. on p. 42).

[53] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. 2023. arXiv: 2308.08155 [cs.AI]. URL: https://arxiv.org/abs/2308.08155 (cit. on p. 42).

[54] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. «SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 37. 2024. arXiv: 2405.15793. URL: https://arxiv.org/abs/2405.15793 (cit. on pp. 45, 60).

[55] Databricks. *What is Unity Catalog? Databricks Documentation*. https://docs.databricks.com/aws/en/data-governance/unity-catalog/. Accessed: 2026-08-25. 2026 (cit. on p. 46).

[56] Astral Software Inc. *uv: An extremely fast Python package and project manager*. https://docs.astral.sh/uv/. Accessed: 2026-08-25. 2026 (cit. on p. 47).

[57] Databricks. *Point-in-Time Feature Joins and Time Series Tables*. https://docs.databricks.com/aws/en/machine-learning/feature-store/time-series. Accessed: 2026-08-25. 2026 (cit. on p. 47).

[58] Microsoft Azure Architecture Center. *Circuit Breaker Pattern - Cloud Architecture Patterns*. https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker. Accessed: 2026-08-25. 2026 (cit. on p. 51).

[59] dbt Labs. *What Data Infrastructure Do Agents Need?* https://www.getdbt.com/blog/what-data-infrastructure-do-agents-need. Accessed: 2026-08-25. 2024 (cit. on p. 51).

[60] LangChain Inc. *LangSmith: Production-Grade LLM Observability and Evaluation Platform*. https://docs.smith.langchain.com/. Accessed: 2026-08-25. 2026 (cit. on p. 52).

[61] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374 (cit. on p. 58).

[62] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. «How to design a program repair bot? Insights from the Repairnator project». In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 2018, pp. 95–104 (cit. on pp. 60, 61).

[63] LangChain, Inc. *LangSmith: Observability and Evaluation Platform for LLM Applications*. `https://smith.langchain.com`. Accessed: 2024. 2024 (cit. on p. 60).

[64] Anthropic, PBC. *Anthropic API Pricing Documentation*. `https://www.anthropic.com/pricing`. Accessed: 2026-08-28. 2024 (cit. on p. 60).

[65] OpenRouter. *OpenRouter LLM Model Pricing Documentation*. `https://openrouter.ai/models`. Accessed: 2026-08-28. 2024 (cit. on pp. 60, 62).

[66] John Yang, Akiba Liu, Shunyu Wu, Carlos E Jimenez, Karthik Narasimhan, and Shunyu Yao. «InterCode: Standardizing Interactive Execution Environments for Interactive Reasoning». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023 (cit. on p. 61).