JOHANNES KEPLER
UNIVERSITY LINZ

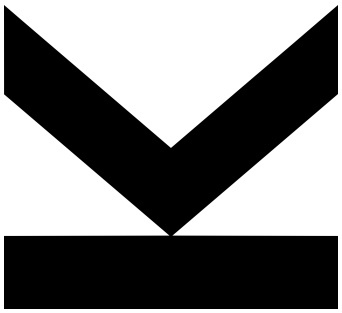
Author
Alexander Mayr, BSc

Submission
Institute for Business Informatics - Data & Knowledge Engineering

Thesis Supervisor
Assoz. Univ.-Prof. Mag. Dr. **Christoph Schütz**

September 2026

Automation of Schema Evolution in a Data Lake and Data Vault Architecture



Master Thesis

to confer the academic degree of

Master of Science

in the Master's Program

Business Informatics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
www.jku.at

Kurzfassung

In modernen Datenplattformen verändern sich Quellschemata kontinuierlich. In aufeinander aufbauenden Architekturen, die einen Data Lake (DL) mit einer Data Vault (DV) 2.0 Integrationsschicht kombinieren, können Schemaveränderungen in der Anlieferung Auswirkungen auf Persistenz, Transformation und nachgelagerte analytische Schnittstellen haben. Dadurch entsteht ein schichtübergreifendes Koordinationsproblem.

Diese Arbeit behandelt die Automatisierung von Schema Evolution und Daten-Pipeline-Management in kombinierten DL und DV Umgebungen mit dem Ziel, manuellen Aufwand zu reduzieren, Datenqualität zu sichern und einen verlässlichen Betrieb unter asynchroner Ausführung und partiellen Fehlern zu gewährleisten. Auf Basis von Wieringas Methodologie zur gestaltungsorientierten Forschung wird das *Schema Evolution Framework (SEF)* als übergeordnete Steuerungsinstanz für sichere Koordination von Schemaveränderungen entworfen, implementiert und evaluiert. Das Framework trennt das Erkennen einer Veränderung von der Vervollständigung und unterstützt die Steuerung, Bewertung der Anpassungen, Migrationsplanung, idempotente Durchführung und evidenzbasierte Verifizierung vor der Veröffentlichung der Ergebnisse.

Die Evaluierung zeigt, dass der vorgeschlagene Ansatz eine kontrollierte und überprüfbare strukturelle Schemaentwicklung innerhalb des definierten Umfangs ermöglicht und die Konvergenz unter der „At-Least-Once“-Semantik unterstützt. Die Governance-Evaluierung zeigt darüber hinaus eine messbare Asymmetrie zwischen blockierten und erlaubten Änderungspfaden sowie backend-abhängige Latenzeffekte bei Wiederholungsversuchen, die sich als Ausreißer im oberen Latenzbereich zeigen können. Detaillierte operative Benchmark-Ergebnisse sind im begleitenden technischen Online-Supplement dokumentiert.

Abstract

Modern data platforms operate in environments where source schemas change continuously. In layered architectures that combine a Data Lake (DL) with a Data Vault (DV) 2.0 integration layer, schema drift at the ingestion boundary can propagate across persistence, transformation and downstream analytical interfaces, creating a cross-layer coordination problem.

This thesis addresses the automation of schema evolution and pipeline management in combined DL and DV environments with the goals of reducing manual intervention, preserving data quality and ensuring reliable operation under asynchronous execution and partial failure. Following Wieringa's Design Science Methodology, the thesis designs, implements and evaluates the *Schema Evolution Framework (SEF)* as a control plane mechanism for replay-safe schema evolution coordination. The framework separates schema observation from schema completion and supports admission control, policy evaluation, migration planning, idempotent actuation and evidence-based verification before publishing outcomes.

The evaluation shows that the proposed approach enables controlled and verifiable structural schema evolution within the defined scope and supports convergence under at-least-once delivery semantics. The governance evaluation further demonstrates a measurable asymmetry between blocked and allowed change paths, and that retry-induced latency effects are backend-dependent and may manifest as tail amplification. Detailed operational benchmark findings are reported in the online technical supplement.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability. 88

API Application Programming Interface. 40

CDC Change Data Capture. 17–21, 50, 51, 53, 57, 60, 62, 67

CSV Comma-Separated Values. 55, 58, 67, 92, 93

DDL Data Definition Language. 60

DG Design Goal. 11, 24

DL Data Lake. 1–7, 10–14, 17, 18, 20, 24, 25, 28, 29, 46, 51, 56, 67, 75, 76, 78, 79, 81

DSM Design Science Methodology. 3–5, 7, 74, 78

DV Data Vault. 1–7, 10–15, 17–20, 24, 25, 28, 29, 40, 41, 46, 48, 50, 51, 61, 67, 75, 76, 78–81

DWH Data Warehouse. 12, 14, 81

E2E end-to-end. iv, vi, viii, 1, 6, 7, 9, 11, 12, 24, 27, 28, 35, 44, 54, 55, 63, 65–70, 74–76, 79–81, 97–99

EDW Enterprise Data Warehouse. 13, 15

ELT Extract–Load–Transform. 13, 82

ETL Extract–Transform–Load. 13, 94

gRPC Remote Procedure Calls. 20, 32, 46

IQR Interquartile Range. 72

JSON JavaScript Object Notation. 32, 38, 39, 43, 54

NoSQL Not Only SQL. 1, 7–10

OLAP Online Analytical Processing. 12

List of Abbreviations

OLTP Online Transaction Processing. 12

PII Personally Identifiable Information. 13

PIT point-in-time. 81

RDBMS Relational Database Management System. 9, 95, 97, 98

RPC Remote Procedure Call. 32, 40

RQ Research Question. 3, 5, 70, 73–76, 79

SEF Schema Evolution Framework. vi, viii, 1, 4–6, 10, 11, 17–27, 33, 35, 37, 38, 40, 41, 43–54, 56–58, 60, 62, 65, 67, 70, 72–76, 78–82, 92, 94, 98

SLA Service Level Agreement. 8, 9

UI User Interface. 80, 81

1. Introduction

In modern data platforms, schema evolution is a continuous operational task. As organizations integrate heterogeneous data sources, the structure of ingested data changes frequently, often without central coordination. Literature shows that schema evolution affects data-intensive applications using relational, Not Only SQL (NoSQL), and multi-model storage layers [1, 2, 3]. Structural drift in the sources not only affects the storage layer but also ingestion pipelines, metadata services, transformation models, and downstream consumption processes, making schema management a cross-layer challenge.

Modern data platforms often employ a combined Data Lake (DL) and Data Vault (DV) setting. Changes in the sources must be propagated from the DL to the DV schema. In this architecture, the DL serves as the ingestion and persistence layer, capturing raw datasets at scale with low up-front modeling cost, while DV 2.0 provides the integration backbone through regular constructs, namely Hubs, Links, and Satellites, that separate identity, relationships, and descriptive state to support auditable, change-tolerant enterprise integration [4, 5]. Although DV 2.0 modeling is designed to be additive and resilient to continuous change, structural drift originating at the ingestion boundary must traverse both layers in a coordinated manner. Under asynchronous, event-driven execution with at-least-once delivery semantics, schema changes can be observed, replayed, and processed out of phase with downstream model updates, introducing operational risk unless evolution is explicitly coordinated across the combined DL and DV setting [6].

This thesis addresses schema evolution across a combined DL and DV architecture by introducing the *Schema Evolution Framework* (SEF), a control plane mechanism for coordinating and verifying structural schema changes E2E. The SEF is embedded in a broader artifact that combines an ingestion-boundary observer (Filewatcher), layer handlers for lake and vault ingestion, and a message bus based on Apache Kafka. Together, these components form a complete pipeline from drift detection to verified evolution completion.

1.1. Motivation

In analytics environments, schema drift produces recurring manual effort across roles and architectural layers. Frequent change classes include attribute additions, deletions, type modifications, and integration of new source systems. These changes not only affect ingestion parsing and lake table schemas but also downstream transformations and warehouse-facing delivery contracts, forcing data teams to repeatedly refactor and coordinate across layers, risking silent data quality degradation when changes are applied incompletely or out of order.

The combination of a DL and DV is particularly suitable for this setting because the two layers provide complementary properties. The DL offers a flexible ingestion and persistence boundary for heterogeneous and changing source structures, while DV 2.0 is designed to absorb change incrementally through stable, historized integration structures [16, 4]. This creates a natural basis for automation: structural changes can be captured flexibly in the lake and propagated into the vault through deterministic and auditable adaptations, providing potential for a simple and reliable mechanism to keep both layers aligned.

Literature on schema evolution shows that schema-light environments do not eliminate the management burden but shift the management burden from the storage engine into integration tooling and metadata governance [1, 3]. Event-driven architectures operate under at-least-once semantics, exposing systems to duplicates and retries that must converge to stable outcomes [6]. Accordingly, schema evolution automation must be replay-safe and must formally distinguish between detecting a change and completing a verified evolution.

This thesis addresses *structural* schema evolution, meaning controlled modifications such as adding or removing attributes and changing attribute types, derivable from technical evidence including schema snapshots, inferred types, data lineage, and model dependencies [7, 2]. Semantic evolution, where the meaning of existing fields changes without a structural indicator, is excluded, as semantic evolution requires domain interpretation that cannot be robustly automated without additional context.

In a layered DL and DV architecture, structural drift at the ingestion boundary must propagate across lake persistence objects, derived change streams, vault loading logic, and downstream delivery models. DV 2.0 localizes change through construct separation but provides no automated mechanism for coordinating that change across layers [4]. Without explicit coordination, layers may evolve inconsistently. Persistence may admit new columns while downstream models remain unaware. Modeling may also update before persistence has applied the changed schema. Neither layer can resolve this problem in isolation. This thesis therefore treats schema

evolution as a cross-layer coordination problem that requires detection, planning, execution, and verification under realistic failure and replay conditions.

1.2. Methodology

This section formulates the design problem and corresponding research questions, outlines the proposed solution approach, positions the thesis within Design Science Methodology (DSM), and defines the scope of the work.

1.2.1. Design problem statement and research questions

The central design problem addressed in this thesis is the following:

> Reduce the manual effort and operational risk caused by structural drift in the sources by designing a control plane mechanism to automate schema evolution across a combined DL and DV environment under asynchronous, at-least-once execution.

This formulation narrows the broader challenge of schema evolution to the concrete setting considered in this thesis. The focus is not on schema change in isolation within a single storage layer, but on coordinated propagation across a layered architecture in which ingestion, persistence, integration, and downstream delivery must remain operationally aligned despite retries, duplicates, and partial failures.

The thesis investigates this design problem through the following Research Questions (RQs), structured to reflect the lifecycle from drift observation to verified completion:

RQ1: **Detection and representation.** How can structural schema drift at the ingestion boundary be detected and represented in a normalized form that supports deterministic downstream processing?

RQ2: **Cross-layer propagation.** How can schema evolution be propagated consistently across a layered DL and DV architecture while minimizing manual refactoring effort?

RQ3: **Reliability, governance and quality assurance.** Which policy and verification mechanisms are required to automate schema evolution safely under at-least-once delivery and partial failure, while preserving data quality and maintaining stable consumption contracts?

Together, these questions establish the requirements for a solution that treats schema evolution as a governed coordination problem rather than as an isolated schema update within one system component.

1.2.2. Approach

This thesis addresses the design problem by treating schema evolution as a control plane concern rather than as a side effect of ingestion or modeling logic. In an asynchronous platform, schema observation must not be conflated with schema modification. A detected drift is therefore treated as a *proposal* that requires admission control, explicit planning, idempotent execution, and evidence-based verification before the detected drift can be accepted as completed.

This separation between drift detection and completion is realized through a distinction between data plane and control plane, introduced in detail in Section 4.1. The data plane transports and persists record payloads, while the control plane interprets schema posture information and coordinates evolution across layers. At the ingestion boundary, the Filewatcher emits both row delta messages and schema notifications. The SEF consumes these notifications, derives change atoms, evaluates governance policies, synthesizes evolution plans, executes them through the corresponding layer handlers, and publishes a canonical completion signal only after verification.

The SEF design reflects literature findings that metadata and data lineage are key enablers of governed change in data-intensive systems [8, 9].The design is also a response to observability research that identifies schema drift as a recurring root cause of downstream incidents [10]. Accordingly, the proposed artifact is not merely an automation script for schema adaptation, but a coordination mechanism intended to ensure replay safety, convergence, and auditable completion under realistic operating conditions.

1.2.3. Design science methodology positioning

Because the thesis aims to design and evaluate an artifact that addresses a practical engineering problem, the thesis is positioned within the DSM proposed by Wieringa [11]. Table 1.1 organizes the thesis' activities across four DSM phases.

The Filewatcher, Data Lake Handler, and Data Vault Handler are fully realized prototypes available in the online technical supplement [12]. The corresponding thesis chapters present technology-agnostic conceptual designs, focusing on behavioral contracts, invariants, and SEF

Table 1.1.: Mapping of thesis structure to Wieringa's Design Science Methodology (DSM) phases [11].

DSM phase	Thesis realization (chapters and outputs)
Problem investigation	Chapter 1 (problem, objectives, RQs), Chapter 2 (gaps and derived design goals), Chapter 3 (technical foundations and relevance for schema evolution)
Treatment design	Chapter 4 (artifact boundary, assumptions), Chapter 5 (formalization, planning and verification semantics)
Treatment realization	Conceptual design chapters (Filewatcher, handlers, integration with SEF), and the corresponding reference implementation provided in the online technical supplement
Evaluation	Chapter 9 (test cases, scenarios, metrics, results)

integration semantics, because these design decisions determine the correctness and replay-safety properties examined in the evaluation.

This methodological positioning clarifies how the thesis progresses from problem analysis to artifact design, realization, and evaluation, and how the proposed solution is assessed as an engineering treatment for a concrete class of schema evolution problems.

1.2.4. Scope

To keep the design problem tractable and the evaluation interpretable, the thesis is bounded along three dimensions.

First, the work addresses *structural* schema evolution only. This includes controlled modifications such as adding or removing attributes and changing attribute types, derivable from technical evidence including schema snapshots, inferred types, data lineage, and model dependencies [7, 2]. Semantic evolution, in which the meaning of an existing field changes without a structural indicator, is excluded because semantic evolution requires domain interpretation that cannot be robustly automated without additional contextual knowledge.

Second, the work is situated in a layered architecture that combines a Data Lake persistence layer with Data Vault 2.0 modeling constructs. The contribution is therefore not intended as a universal solution for all forms of data integration architecture, but for a setting in which structural drift must propagate from ingestion-facing persistence into auditable integration models.

Third, the system assumes an event-driven execution model with asynchronous processing and at-least-once delivery. The artifact is therefore designed explicitly for replay safety, idempotent execution, and convergence under duplicate delivery and partial failure. Other execution assumptions, such as globally coordinated exactly-once processing, are outside the scope of this thesis.

1.3. Structure of the thesis

The remainder of the thesis is organized as follows:

- Chapter 2 surveys related work on schema evolution in event-driven data platforms and derives design goals that motivate the Schema Evolution Framework (SEF).
- Chapter 3 establishes the technical basis, covering data warehousing, DL and lakehouse paradigms, and DV 2.0 modeling, and clarifies their relevance for schema evolution in a DL–DV pipeline.
- Chapter 4 introduces the overall artifact architecture, including the separation into data plane and control plane, message families on the event bus, component responsibility boundaries, and failure assumptions.
- Chapter 5 specifies the SEF as the control plane mechanism for E2E schema evolution coordination, including formal objects, planning semantics, and verification gating.
- Chapter 6 and Chapter 7 present the conceptual design of the ingestion observer and the layer handlers, including their behavioral contracts and integration with the SEF. The reference implementations of these components are provided in [12].
- Chapter 8 presents a complete E2E example that instantiates the cross-chapter contracts with concrete artifacts, including files, event-bus messages, policy outcomes, operation plans, and verification evidence.
- Chapter 9 presents the evaluation results, discusses their implications, and states the validity boundaries of the work.
- Chapter 10 concludes the thesis by summarizing the artifact, answering the research questions, and outlining directions for further research and engineering.

2. Related work

In Wieringa's Design Science Methodology [11], the problem investigation establishes the environment, constraints and existing knowledge that inform the design of an artifact. This chapter positions schema evolution as a cross-layer concern in event-driven combined DL and DV pipelines and surveys research contributions that inform automation, governance and evaluation of evolution mechanisms, with particular focus on replay-safe coordination, explicit planning and actuation, and evidence-based verification.

The literature provides multiple building blocks for schema evolution automation, but most work focuses on a subset of the E2E problem. Classical and modern surveys discuss schema evolution in databases assuming a single storage boundary and centralized control [7, 1]. Recent work proposes generic evolution mechanisms across relational and NoSQL systems [2] and explores adaptive migration strategies under varying runtime constraints [13]. Benchmarking efforts motivate systematic evaluation under controlled workloads covering both additive and destructive transformations [14, 15]. For DL environments, surveys identify metadata, cataloging and data lineage tracking as central to safe evolution [16], while provenance-oriented approaches provide foundations for representing lineage across pipeline steps [8, 9]. Data quality observability work highlights drift detection and transparency mechanisms as critical for operational robustness [10].

Table 2.1 positions the reviewed literature along three analytical dimensions. The first dimension, *Schema Drift Detection Mechanism*, captures whether an approach actively detects schema changes or assumes that such changes are already known. The second dimension, *Metadata Integration*, describes how explicitly the approach relies on metadata, schema representations, provenance information or runtime context to manage evolution. The third dimension *Automation and Adaptation Impact* summarizes the extent to which the approach reduces minimal evolution effort, automates migration activities or adapts to changing system conditions

The comparison in Table 2.1 shows that the approaches address related but distinct aspects of schema evolution. Hernández et al. [2] focus on formalizing schema change operations and automating their execution across heterogeneous data models. Hillenbrand et al. [13] emphasize adaptive migration strategy selection based on workload, schema-change context

Table 2.1.: Literature review concept matrix.

Approach	Schema Drift Detection Mechanism	Metadata Integration	Automation & Adaptation Impact
Hernández et al. [2]	Drift detection is not a primary focus (assumes schema changes are known). Introduces a unified schema change operation taxonomy (SCO) to enable systematic handling of evolution operations.	Introduces the Orion language on the U-Schema model to unify relational and NoSQL schemas, using metadata about entities and relationships to ensure consistency of change operations across heterogeneous databases.	Automates multi-model schema evolution by generating database-specific migration scripts from high-level operations, validated across columnar, document, key-value, graph and relational stores.
Hillenbrand et al. [13]	Monitors runtime workload and schema change context to adapt migration strategy (eager, lazy, etc.), though not focused on detecting the schema change itself, because the change trigger is external.	Leverages system metadata (workload statistics and schema change types) and integrates SLAs as meta-constraints for adaptation.	Proposes a self-adapting data migration framework that automatically adjusts migration strategies and parameters with respect to the migration scenario and SLAs, optimizing trade-offs and supporting co-evolution with minimal manual tuning.
Pérez et al. [9]	No new detection technique. Focuses on tracking changes. Schema changes are assumed to be captured as events that require provenance tracking rather than being actively discovered.	Proposes PROV-IDEA to manage schema evolution with provenance metadata. Records schema versions and data lineage using the W3C PROV standard, enabling interoperability across evolution metadata tools.	Provides an automated provenance-integrated evolution management approach: as schemas evolve, the system maintains versioned provenance records for both schema and data, improving auditability and integration of schema changes in pipelines.
Yamanaka et al. [17]	Statistical schema change detection via data-distribution matching (Kolmogorov-Smirnov tests) to automatically identify corresponding columns across evolving datasets.	Minimal reliance on external metadata. Focuses on data content rather than schemas and treats each new dataset version independently (data-driven alignment).	Automates schema alignment in data integration pipelines, achieving approximately 90% accuracy in matching attributes across 12 years of evolving public datasets and reducing manual mapping effort.

and SLAs. Pérez et al. [9] concentrate on provenance based tracking and auditability of schema and data changes, whereas Yamanaka et al. [17] investigate data-driven detection of attribute correspondence across evolving datasets. Overall, the matrix indicates that existing work tends to emphasize detection, metadata management, provenance or adaptive migration as separate concerns. This motivates treating schema evolution in this thesis not as a local database migration task, but as an explicit control plane procedure that jointly coordinates detection, metadata integration and automated adaptation.

2.1. Schema evolution in event-driven data platforms

Modern analytical platforms are subject to persistent change driven by evolving business requirements, new source systems and iterative modernization of operational applications. Literature consistently characterizes schema evolution as a pervasive phenomenon across classical Relational Database Management System (RDBMS), NoSQL and multi-model settings [1, 3]. For the remainder of this thesis, schema evolution is treated as a first-class operational concern that must be addressed E2E rather than as an isolated database migration task.

This thesis focuses on *structural* evolution: changes observable as modifications to the structure of a dataset, such as added or removed attributes, changed attribute types or nullability constraints. Structural evolution can be detected using schema snapshots and type inference and represented as explicit change atoms. Semantic evolution, covering changes in meaning without structural indicators, is excluded, since semantic evolution requires domain interpretation that cannot be automated robustly without additional business context.

From an automation perspective, representing schema drift as explicit change operations is essential. Hernández et al. [2] propose a generic taxonomy of schema change operations for heterogeneous databases, highlighting that many practical changes can be decomposed into a small set of primitive operations such as add, delete or modify. Complex modifications such as renames, splits or merges require additional assumptions, because they cannot be derived from structural evidence alone. In contrast to additions, removals or type-level changes, such modifications usually depend on the intended meaning of attributes and on domain knowledge about how source fields relate to each other. They therefore represent semantic schema evolution rather than purely structural schema drift. This thesis adopts a conservative stance: semantic interpretation, including the automated detection of renames, splits and merges, remains outside the thesis scope. Such changes may still be handled by the proposed framework if they are made explicit through policy or manual modeling decisions, but they are not inferred automatically from ingestion-boundary observations.

Empirical studies and benchmarks underline that realistic evolution is not limited to purely additive changes. EvoBench [14] provided a benchmark for schema evolution in NoSQL infrastructures that included both additive and destructive transformations, supporting systematic evaluation of evolution mechanisms under controlled workloads. Time-related analyses of evolution histories indicate that changes often occur in bursts around release dates, followed by periods of stability [15]. These findings have two implications for the system design of this thesis: (i) automation should distinguish between non-breaking and potentially breaking changes, and (ii) the system must support repeatable and testable evolution procedures because change episodes recur.

When schema evolution is embedded into event-driven data pipelines, the execution model differs fundamentally from classical offline migration. Fragkoulis et al. [6] emphasize that at-least-once delivery and asynchronous orchestration are common assumptions, which shifts correctness requirements to consumer-side idempotency and replay-safe coordination mechanisms. From this, the system described in this thesis must therefore treat detected schema drift as an input proposal rather than a complete change. Safe automation requires explicit stages for admission, planning, execution and verification, which are realized in the SEF as a control plane procedure described in Section 4.1.

Across DL and lakehouse systems, metadata is consistently identified as the central enabler for governed evolution [16]. Transactional lakehouse designs such as Delta Lake emphasize that schema changes should be reflected in versioned metadata to make evolution explicit and auditable [18]. Provenance-centric approaches such as *PROV-IDEA* integrate schema versioning with lineage records to preserve interoperability and auditability across schema change episodes [9]. Work on data observability identifies schema drift as a recurring root cause of data incidents, motivating explicit monitoring and verification mechanisms [10]. In this thesis, these insights motivate a design in which evolution is not considered complete until verification has produced evidence that layer-specific artifacts match the intended post-change structure.

2.2. Design goals derived from the literature

The reviewed work provides strong building blocks for drift representation, migration strategy selection and provenance-aware metadata management. However, most approaches assume either centralized control within a single storage boundary or a migration procedure that can be executed as an offline task. The combined DL and DV setting studied in this thesis operates under asynchronous, at-least-once processing where duplicates and replays are expected and where multiple layer-specific artifacts must converge to a coherent post-change posture.

Based on these gaps, the following Design Goals (DGs) are derived for the SEF and the surrounding handlers:

- **DG-1 (Evidence-based completion):** Treat drift detection as a proposal and declare evolution complete only after layer-specific verification evidence has been produced.
- **DG-2 (Replay-safe coordination):** Ensure idempotent actuation and convergence under duplicates, retries and out-of-order processing.
- **DG-3 (Explicit planning semantics):** Represent evolution as explicit change operations and synthesize an ordered plan across affected layers.
- **DG-4 (Metadata as control enabler):** Maintain versioned schema posture, identifiers and traceability metadata to support auditing and dependency reasoning.
- **DG-5 (E2E propagation):** Propagate accepted evolution consistently across DL persistence, DV integration artifacts and presentation or consumption structures.
- **DG-6 (Conservative automation scope):** Prioritize deterministically derivable and safely executable operations. Exclude semantic evolution that requires domain interpretation.

These goals translate into functional and non-functional requirements (see requirements catalog in Appendix B) and guide the architectural decomposition and formal SEF specification in the subsequent chapters.

3. Foundation

This chapter establishes the conceptual and terminological basis required for the subsequent design and evaluation. The discussion follows the progression from classical data warehousing toward DL and lakehouse paradigms and introduces DV 2.0 as a modeling approach for change-resilient integration. The purpose is to establish where schemas are defined and drift is observed, which layer-specific artifacts must be updated when evolution occurs, and which architectural constraints influence replay-safe and verifiable automation.

3.1. Data Warehousing

Data warehousing emerged from a persistent mismatch between operational information systems and analytical decision support. Operational systems are optimized for Online Transaction Processing (OLTP) workloads: they process short, concurrent transactions, enforce strict consistency constraints and prioritize low-latency updates. By contrast, analytical systems are dominated by Online Analytical Processing (OLAP) workloads that scan and aggregate large volumes, integrate multiple sources and require stable historical context. The modern Data Warehouse (DWH) addresses this mismatch by providing a consolidated and historically consistent repository explicitly engineered for analytical consumption.

Analytical repositories must be logically distinct from operational databases, not only for performance reasons but also because analytical semantics require integration and consistent interpretation across multiple business domains. Inmon [19] describes an early DWH architecture in which heterogeneous operational sources are consolidated into an integrated store and then exposed through delivery structures designed for decision support. Even when compute and storage are elastic, analytical correctness depends on consistent conformance, historization and governance.

The E2E structure of a DWH is commonly described through layered architectures separating landing and cleansing, enterprise integration and information delivery [20]. In a `two-layer` architecture, data flows from a staging layer directly into consumption-oriented warehouse

structures. This increases coupling between upstream schema evolution and downstream consumption because integration and delivery are realized within the same layer. In a `three-layer` architecture, an additional integrated core (frequently implemented as an Enterprise Data Warehouse (EDW)) sits between staging and information layers, preserving history and providing auditable semantics intended to remain stable across multiple consumption use cases. This separation reduces coupling between source evolution and consumption contracts: delivery models can be adapted without forcing a redesign of the integration backbone, which is the property the DV layer exploits when absorbing structural drift from upstream sources.

The pipeline populating a warehouse is traditionally described as Extract–Transform–Load (ETL): data is extracted from sources, transformed into integrated representations and loaded into the warehouse. With the increasing prevalence of scalable analytical platforms, the Extract–Load–Transform (ELT) pattern has become more common, where data is first loaded and transformations are executed in-place. Independently of ordering, integration remains a modeling and engineering problem: transformation semantics, dependency structures and mappings must be explicit to be verifiable and maintainable under change [21].

3.2. Data Lake and Lakehouse

DLs were introduced to address limitations of classical warehousing in contexts characterized by high data volume, high data variety and rapidly changing sources [22]. The paradigm emphasizes early retention of raw or lightly processed datasets on scalable storage, postponing schema alignment and business interpretation to downstream processing, a shift from schema-on-write to schema-on-read. This flexibility supports exploratory analytics and accommodates semi-structured and unstructured data not naturally conformed to relational structures. In practice, DLs are frequently implemented on object store infrastructures and rely on open formats for interoperability, while compute engines such as Apache Spark provide scalable processing.

Flexibility does not remove the need for disciplined data management. Insufficient governance can cause lakes to degrade into data swamp, in which datasets are difficult to discover, trust and reuse [23]. The DL concept therefore implies an operating model for ingestion, cataloging, quality handling, security controls including the treatment of Personally Identifiable Information (PII), and lifecycle management.

DLs are commonly organized as layered refinement zones that progressively increase the semantic and quality guarantees of data. A typical structure distinguishes a raw zone for immutable landed data, an intermediate zone for validated and standardized representations

and curated zones for consumption-oriented datasets. The raw zone preserves source-aligned information with strong provenance as a reproducible baseline. Downstream zones progressively impose stronger constraints such as standardized typing, conforming reference values, quality checks and enrichment [24].

This zoned organization is a direct response to schema evolution. When sources evolve, raw ingestion can remain stable as long as ingestion captures files and minimal metadata consistently, while transformations in downstream zones can be adapted incrementally. Metadata management is the defining capability that transforms a storage-centric lake into a reusable analytical platform: metadata management encompasses technical metadata (formats, schemas, partitions), operational metadata (load times, validation outcomes, lineage), and business metadata (domain definitions, ownership, usage constraints). When datasets evolve structurally, consumers require mechanisms to identify what has changed, whether existing transformations remain valid and which downstream assets are affected [16].

A lakehouse combines the flexible and scalable storage characteristics of a DL with data-management capabilities traditionally associated with a DWH, including transactional guarantees, schema enforcement and reliable table abstractions. The approach aims to retain the openness and low-cost storage model of a DL while providing stronger consistency and governance for analytical workloads [25].

Delta Lake is a representative storage layer for lakehouse architectures, implementing transactional semantics and table management on top of decoupled storage via a transaction log that records table versions and enables controlled updates, deletes, merges and time travel queries [18]. Delta Lake manages the interplay between schema-on-write and schema-on-read: ingestion can remain flexible while Delta tables define explicit schemas that can be enforced or evolved under explicit rules, providing a pragmatic basis for managing change in shared analytical environments.

3.3. Data Vault 2.0

While DLs and lakehouse storage layers address flexible retention and reliable table management, enterprise analytics still requires an integration model that remains stable under continuous change. DV 2.0 is a DWH modeling methodology designed for scalable, auditable and change-tolerant integration of heterogeneous source systems into a long-lived analytical repository [4]. A central premise is that change is not an exception but a normal operating condition: the vault layer therefore prioritizes historization, provenance and auditability, while supporting incremental delivery.

In layered warehouse architectures, DV acts as the integrated core equivalent to an EDW. Downstream Information marts and semantic models are derived from the vault rather than loaded directly from source, reducing coupling between source volatility and consumption contracts [4]. Figure 3.1 provides an overview.

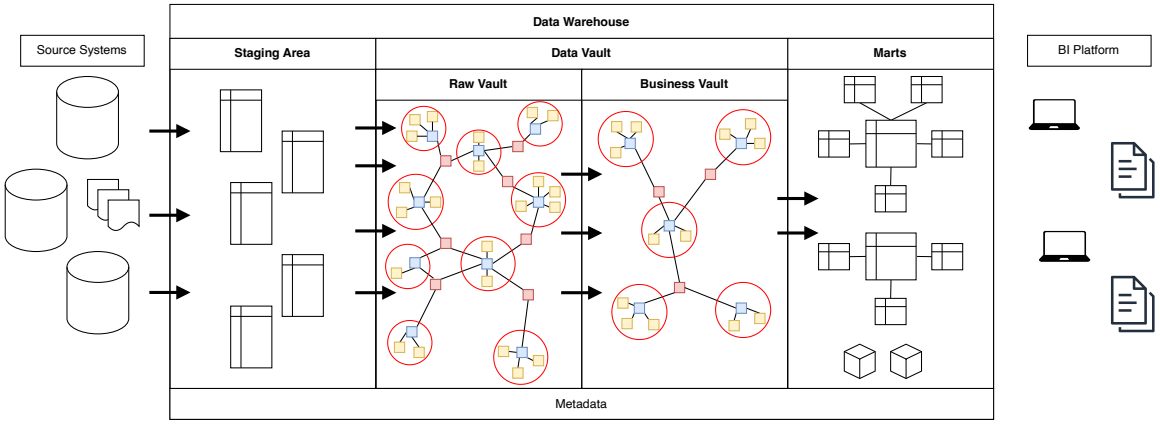


Figure 3.1.: Data warehouse architecture with Data Vault [26].

Data Vault organizes the integrated layer into three construct types: Hubs, Links and Satellites. The constructs are intentionally regular and repeatable. The methodology separates identity and relationships from descriptive attributes, since identity and structural associations tend to be more stable than descriptive attributes, localizing change and reducing the probability that source evolution triggers invasive redesign of the integration layer [4, 5].

Hubs Hubs store unique business keys for a business object type together with minimal technical metadata. A Hub is intentionally narrow, carrying no descriptive attributes beyond what is required to identify the entity. New descriptive fields can be added without altering hub structures. In DV 2.0, reconciliation of divergent identifiers is typically placed in the Business Vault, so that the Raw Vault remains an auditable record of what was observed [4].

Links Links represent associations between Hubs, modeling transactions, memberships, hierarchical relations or other relationship types. Descriptive relationship attributes are stored in a Satellite attached to the Link rather than in the Link table itself, keeping the Link structure stable even when relationship attributes evolve. Higher-arity relationships can be represented as a single Link referencing the participating Hubs [4].

Satellites Satellites store descriptive attributes and historization for a Hub or Link, loaded in an insert-only manner to preserve complete change history. Attributes that change at different rates or originate from different sources are commonly separated into distinct Satellites, localizing change and supporting incremental evolution. New attributes can be introduced by adding new Satellites rather than modifying existing historized structures [4].

The Raw Vault captures integrated, historized data with minimal business transformation, preserving source-aligned meaning together with explicit provenance. The Raw Vault is designed to be additive: new sources can be added as additional Hubs, Links and Satellites without restructuring existing objects. When sources add new descriptive attributes, new Satellites or additional columns in an existing Satellite can be introduced while preserving earlier history [4].

The Business Vault is an optional layer introducing reusable business derivation on top of the Raw Vault. Typical artifacts include standardized or reconciled attribute sets, enterprise-wide calculations, survivorship logic and navigation aids such as bridge tables. Since these artifacts can be re-derived from Raw Vault history, placing them outside the Raw Vault preserves the raw layer as a stable reference while allowing delivery logic to evolve [4].

The vault loading discipline emphasizes non-destructive historization. In a standard batch pipeline, business keys are identified and standardized, Hubs are loaded by inserting previously unseen business keys, Links are loaded by resolving participating Hub keys, and Satellites are loaded by appending new descriptive states when changes are detected. This sequencing supports parallelization because Hubs and Links can be loaded independently across domains. Change detection is commonly implemented through hashdiffs computed over descriptive attributes [4].

A characteristic technique is the use of hash keys as surrogate identifiers, derived deterministically from business key values. Multiple ingestion pipelines can derive identical surrogate keys without coordination, avoiding the bottlenecks of centralized surrogate key generators. Business key values must be normalized consistently across casing, whitespace, locale-specific formatting and null representation, so that logically equivalent keys yield identical hashes [4].

4. Overall system architecture

The Schema Evolution Framework constitutes the primary contribution of this thesis and is presented in detail in Chapter 5. This chapter introduces the surrounding system architecture: the principal runtime components, the separation between payload ingestion and schema governance, the event-bus contracts decoupling producer and consumer, and the handler interfaces through which the SEF applies and verifies schema evolution across the Data Lake and Data Vault layers. The chapter establishes terminology, responsibility boundaries and failure assumptions used as prerequisites for the formal SEF exposition.

4.1. System decomposition: data plane and control plane

The overall architecture separates two conceptual planes. The data plane transports and persists record payloads: the data plane ingests data into the lake backend, derives Change Data Capture (CDC) events and maintains staging and vault artifacts. The control plane transports and interprets schema posture information and coordinates schema evolution actions across layers. The control plane consumes normalized schema notifications, evaluates governance constraints, synthesizes evolution plans, executes them through handlers and emits a canonical outcome event only after verification.

This separation between control plane and data plane matters because the ingestion boundary and the platform layers do not evolve synchronously. The ingestion process may observe new attributes before the persistence layer has been altered to accept them, and downstream modeling may lag behind persistence. The architecture therefore treats schema notifications as proposals and treats the evolution outcome as the only authoritative indicator that a schema transition has been applied and verified.

Dataset identifiers are derived deterministically at the ingestion boundary and used as the routing and partitioning key throughout the platform. Within the SEF, each admitted evolution attempt is bound to a unique correlation identifier and each concrete migration plan receives a distinct plan identifier. Handler-side idempotency is expressed through explicit idempotency

keys, which allow retry semantics under at-least-once delivery to converge to exactly-once effects at the level of schema operations.

4.2. Event bus contracts and message families

An event bus is used to decouple producers and consumers and to enable buffering, replay and scale-out consumption. In this thesis, Kafka is used as an event bus. Kafka records are addressed by topic, partition and Kafka offset. This matters for restart safety because consumers can resume from committed positions or replay retained records. The architecture employs distinct message families reflecting the data plane/control plane separation.

Figure 4.1 illustrates the event-bus topology and message families. In the data plane, the ingestion boundary publishes row deltas to `delta_stream`. These deltas are consumed by the Data Lake Handler and persisted into the configured lake backend. The lake state then serves as the source for insert-only CDC messages on the CDC stream consumed by the Data Vault Handler. In the control plane, the ingestion boundary publishes schema notifications to `schema_stream`. These notifications are consumed by the SEF, which interprets them relative to the last accepted schema version, evaluates governance policies and, if admissible, executes a multi-layer evolution plan. Only after execution and evidence-based verification does the SEF publish a `schema.evolved` event.

Kafka ordering guarantees hold only within a single topic partition. Since notification and delta streams are separate families published to separate topics, the architecture does not assume that a notification is processed before the corresponding payload delta is persisted. Correctness is achieved through idempotent convergence and explicit verification at the control plane level.

Table 4.1 summarizes the canonical topic roles used by the implementation.

In addition to an event bus, a bulk mode has been implemented as fallback, where periodic bulk draining replaces the streaming entirely. Further information for the bulk mode can be found in the support material [12].

Table 4.1.: Canonical event bus topics and their roles in the architecture.

Topic role	Semantics
Row deltas	Append-only payload batches produced by the Filewatcher. Consumed by the Data Lake Handler for persistence.
Schema notifications	Normalized schema posture proposals produced by the Filewatcher. Consumed by the SEF as control plane input.
CDC events	Insert-only change events derived from lake state. Consumed by the DV ingestion subsystem for Bronze persistence and downstream modeling.
Schema evolved	Canonical SEF outcome events emitted after plan execution and verification. Enables downstream orchestration and compatibility-aware routing.

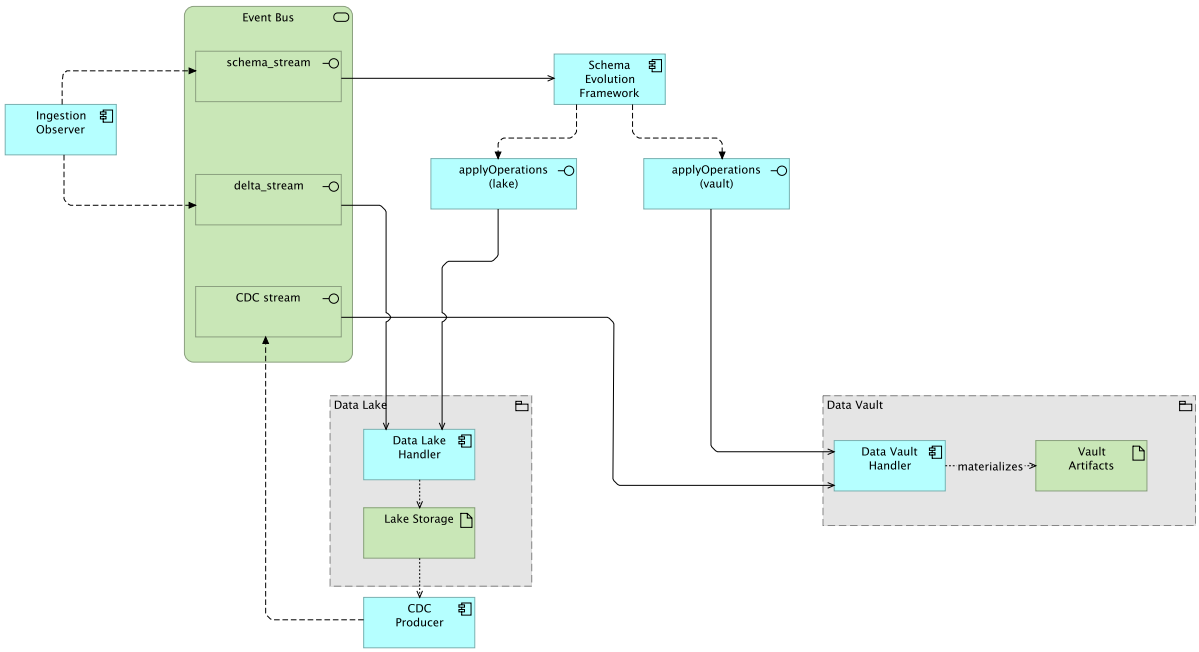


Figure 4.1.: Event bus topology and message families.

4.3. Runtime components and responsibility boundaries

Figure 4.2 shows the overall system architecture. The implemented system comprises four primary services: an ingestion-side observer (Filewatcher), two layer handlers (Data Lake Handler and Data Vault Handler) and the SEF core as a control plane orchestrator.

The Filewatcher monitors a directory for file creation and modification events. Under an apply-only assumption, the Filewatcher computes appended records since the last persisted offset and publishes those appended records as bounded delta envelopes to the data plane stream. The Filewatcher also derives a schema snapshot from the header posture and publishes the schema snapshot as a normalized schema notification to the control plane stream.

The Data Lake Handler ingests row deltas and persists them into the lake backend. The Data Lake Handler also provides a SEF-compatible Remote Procedure Calls (gRPC) boundary through which lake-level schema operations can be applied idempotently and the authoritative lake schema can be introspected as verification evidence. The Data Vault Handler integrates CDC derivation, Bronze ingestion and Raw Vault materialization, and exposes a SEF-compatible gRPC boundary for vault-level operations and evidence introspection.

The SEF consumes schema notification events and implements the control plane semantics of admission, diffing, impact analysis, policy evaluation, migration planning, execution through handlers and verification. The SEF is the only component authorized to publish schema-evolved outcomes, centralizing compatibility classification and verification gating.

Layer handlers implement a common gRPC contract. Each operation specifies the target layer, operation kind, target entity and parameter payload. Handler implementations must be deterministic and idempotent with respect to operation identity, returning a stable `ALREADY_APPLIED` status code when the same operation is repeated. Rather than trusting a handler's success code, the SEF requests an authoritative structural snapshot from the handler derived from the backend catalog and vault modeling artifacts.

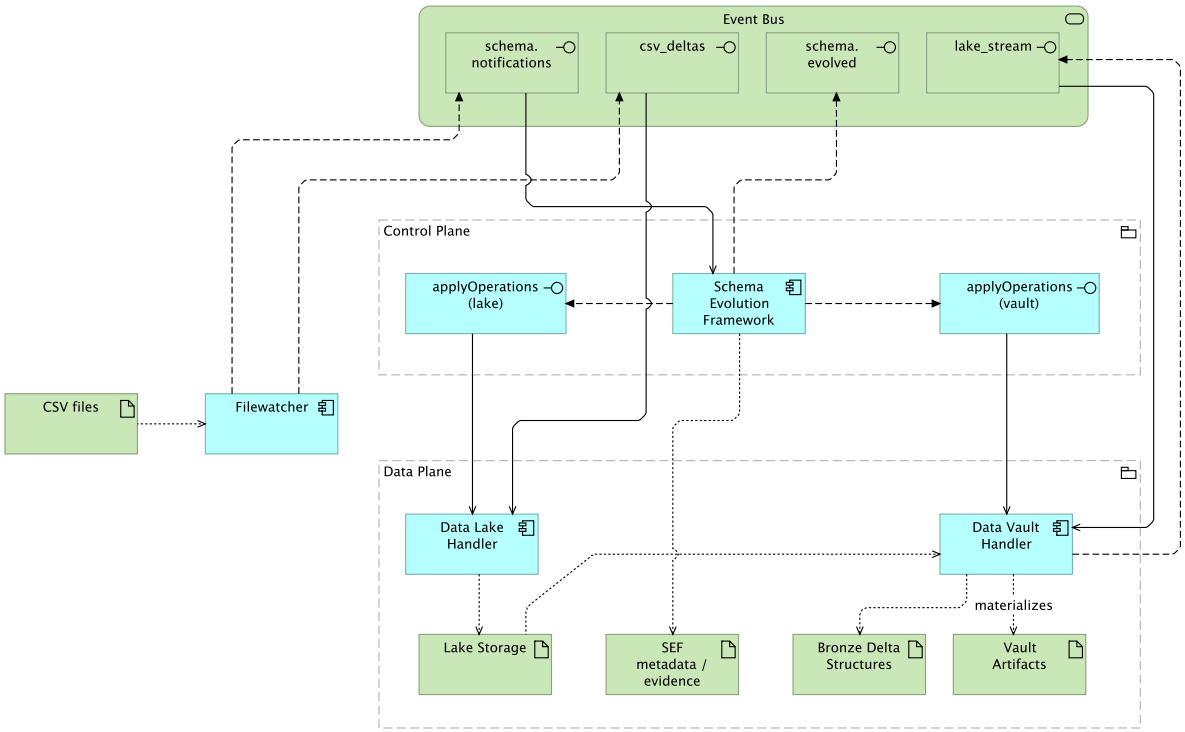


Figure 4.2.: System overview: separation of data plane ingestion and control plane schema evolution.

4.4. Execution views: ingestion and evolution

At runtime, ingestion and evolution proceed concurrently under potentially different scheduling regimes. In the ingestion view, row deltas from `csv_deltas` are consumed and persisted continuously, and CDC insert events are generated from lake state using a watermark discipline before publication to `lake_stream`. In the evolution view, `schema.notification` events trigger SEF planning and handler actuation, which may occur asynchronously with respect to payload ingestion. Figure 4.3 summarizes the ingestion path and Figure 4.4 the evolution path as sequence diagrams.

The concurrency between ingestion and evolution implies that ingestion and evolution can race, especially when the persistence backend is strictly schema-enforcing. If a payload batch includes a newly observed attribute not yet present in the target table schema, ingestion may fail. The architecture relies on a combination of early notification, idempotent handler-side convergence and SEF verification gating, treating stronger coordination barriers as an extension that can be added without changing the fundamental control plane design.

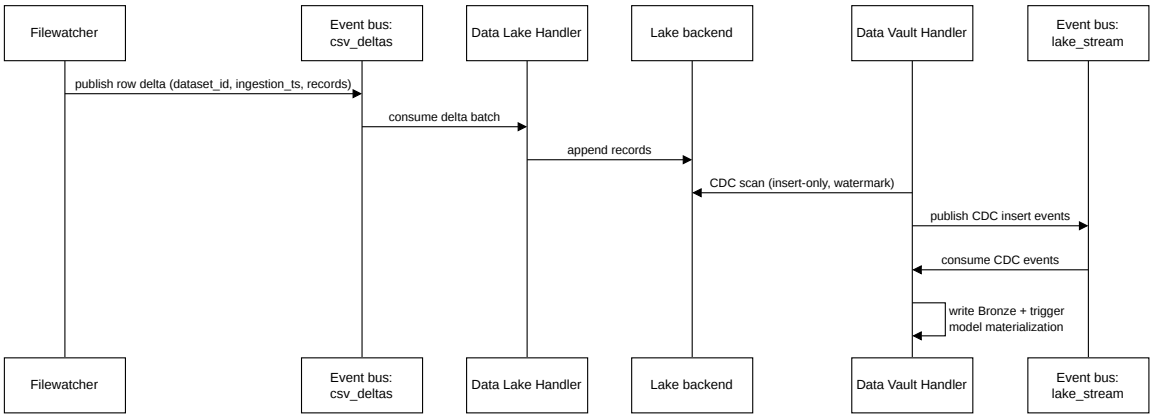


Figure 4.3.: Execution view of data plane ingestion: row deltas to lake persistence and insert-only CDC to vault ingestion.

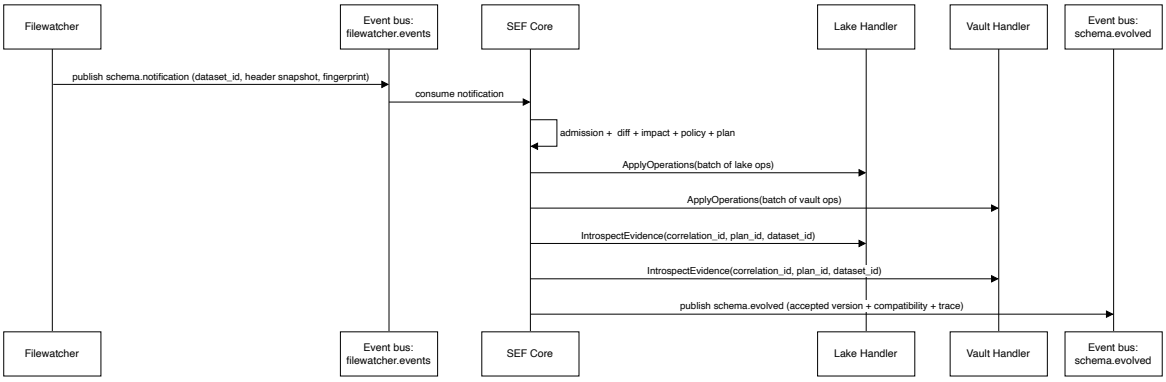


Figure 4.4.: Execution view of control plane evolution: schema notifications to SEF planning, handler actuation and verified publication.

4.5. Persistent state, failure model and convergence semantics

Every artifact whose loss would change operational semantics from restart-safe replay to best-effort reprocessing is persisted outside the application process. The event bus retains message streams as the primary replay mechanism. Ingestion components save offsets to durable storage and the SEF persists the framework metadata, plan and verification repository as the authoritative source of truth for schema versioning. Restarts are expected. Recovery proceeds through replay and idempotent convergence rather than manual intervention. The concrete container topology, volume assignments and service configuration are documented in the online technical supplement [12].

The failure model is consistent with at-least-once delivery and partial failure across independent services. Messages may be duplicated, consumers may restart mid-batch and handler calls may time out or be retried. At the data plane level, ingestion is restart-safe with respect to offsets and checkpoints, but may still exhibit at-least-once semantics at the record level. At the control plane level, the SEF converts at-least-once delivery into exactly-once effects at the level of schema operations by attaching idempotency keys, persisting checkpoints and verifying outcomes against authoritative evidence. A schema notification is an input proposal. Completion is represented only by a schema-evolved event emitted after successful execution and verification.

5. Schema evolution framework

This chapter specifies the SEF as the thesis' central contribution. Chapter 4 established the overall architecture, including the data plane/control plane separation, the event-bus contracts and the failure model in which at-least-once delivery and partial failure are expected rather than exceptional. Building on these prerequisites, this chapter defines how schema evolution is coordinated and verified as an explicit control plane.

The framework is structured around a single guiding premise: in an asynchronous platform, a schema notification is a proposal, not a completed change. The SEF determines whether a proposal is admissible under governance policy, what the cross-layer impact is, and which ordered operations are required to converge the DL and DV layers to a coherent new version. Completion is represented only by a published outcome after idempotent execution and evidence-based verification.

Section 5.1 introduces the SEF progressively. Section 5.2 details the component responsibilities. Section 5.3 specifies the E2E control flow and processing state machine, while Section 5.4 defines the operational records, invariants and verification conditions.

The six design goals from Chapter 2 directly shape this chapter. **DG-1** (evidence-based completion) is realized by the evidence-gated verification semantics in Section 5.4.7. **DG-2** (replay-safe coordination) is realized by the idempotency contract in Section 5.5.2. **DG-3** (explicit planning semantics) is realized in Section 5.4.6. **DG-4** (metadata as control enabler) is addressed by the versioned metadata repository in Section 5.4.3. **DG-5** (E2E propagation) is enforced by the cross-layer ordering constraints in Section 5.5.3. **DG-6** (conservative automation scope) is maintained by the change classification in Section 5.4.4.

5.1. Conceptual overview

The SEF operates as an event-driven control plane that detects schema changes at the ingestion boundary, classifies them into well-defined change classes, plans safe and policy-compliant evolutions across the DL and the DV, and executes those evolutions through dedicated handlers.

The SEF then verifies that the resulting structural state matches the intended post-change state before publishing a new accepted schema version and an authoritative outcome. Rather than introducing all components at once, this section develops the framework top-down, starting from a high-level concept, shown in Figure 5.1, and expanding each component in detail afterwards. Each step makes one additional responsibility explicit while preserving the relationships established in the previous one.

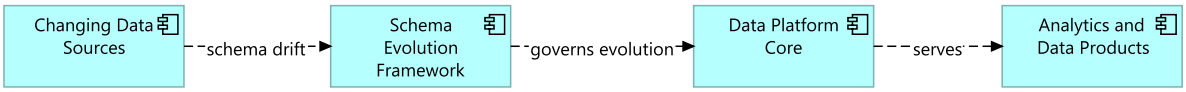


Figure 5.1.: High-level concept for the SEF.

5.1.1. Data platform core

Figure 5.2 shows the decomposition of the single platform core into the two layers this thesis targets: the Data Lake and the Data Vault 2.0 layer.

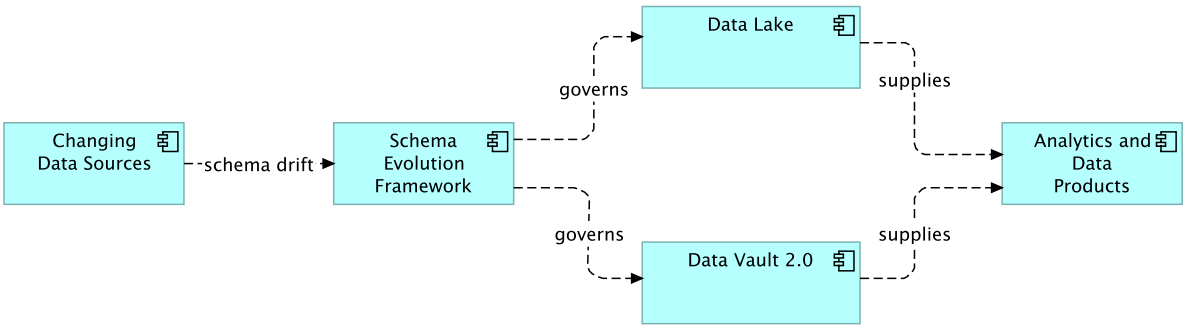


Figure 5.2.: Decomposition of the data platform into Data Lake and Data Vault.

The impact of schema evolution is asymmetric across these two layers. Adding an attribute may be a simple column addition in the lake, whereas the same additive change may require extending a Satellite in the vault. A type change, appearing harmless in the lake, may conflict with change-detection logic in the vault. Additional datasets appear as a simple file or table in the lake, but in the vault a new Hub, one or more Satellites or a new link between Hubs must be established. Every accepted schema version in the framework must correspond to a well-defined arrangement of lake files or tables and of vault objects.

5.1.2. Responsibilities of the SEF

Figure 5.3 reveals the SEF's two principal responsibilities. The framework distinguishes between an *Interpret and Plan* phase and an *Apply and Verify* phase.

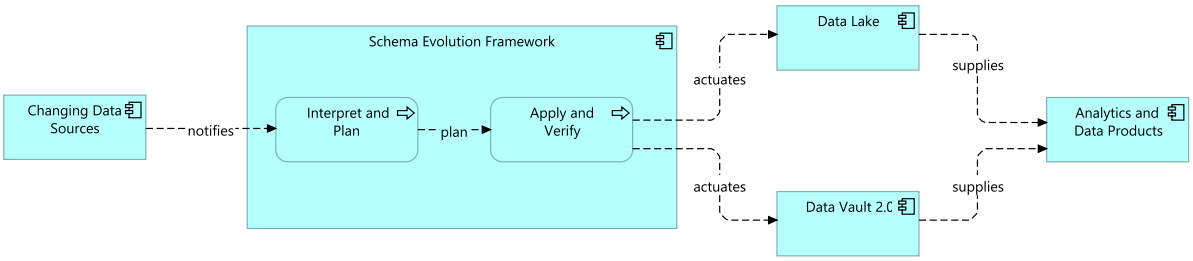


Figure 5.3.: Internal responsibilities of the SEF.

The *Interpret-and-plan* phase ingests schema change information, reconstructs the last accepted schema version from metadata, computes the difference between old and new schemas, classifies the resulting change atoms, assesses their impact via lineage and evaluates policies to determine admissible evolution strategies. The *Interpret-and-plan* phase describes *what* should be done without committing to *how* the planned evolution is implemented on concrete systems.

The *Apply-and-verify* phase takes the resulting plan, orchestrates layer-specific handlers, enforces the plan's operation ordering, manages idempotency keys and checkpoints to remain correct under retries, and performs structural and contract-based checks after operations have been applied. The phase records the new schema version together with the plan and the execution trace, then publishes a concise outcome to downstream orchestrators.

The separation also bounds failure handling: an error during *apply-and-verify* allows the *interpret-and-plan* function to compute compensating plans without reinterpreting the underlying change or re-evaluating policies. The separation also permits dry-run planning and policy simulations without touching production systems [27].

5.1.3. Ingestion

Figure 5.4 introduces how the SEF is notified about schema drift. Source systems deliver files into an ingestion area, where the Filewatcher monitors new and modified files and converts low-level file-system events into normalized schema change notifications consumed by the SEF.

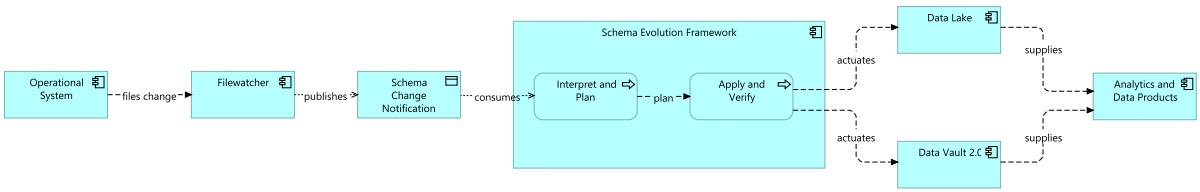


Figure 5.4.: Ingestion path and schema-change notifications.

The SEF depends only on the table contract carried by each notification, namely a dataset identifier, a set of attribute names, and contextual metadata such as a timestamp. Schema inference is performed by the Filewatcher. The SEF therefore receives already interpreted structural information and can focus on version comparison, change classification, and evolution planning. This separation improves portability because the framework can operate above different ingestion technologies as long as they emit the agreed notification format.

5.2. System architecture and component responsibilities

The E2E system comprises an ingestion service, a lightweight event bus, a core framework, specialized handlers for each target layer and repositories providing persistent metadata, lineage, auditability and metrics. A change director consumes notifications and serves as the entry point into a controlled processing state machine. An impact analyzer interrogates metadata and lineage to determine which tables, storage objects, Hubs, Links, Satellites, views and scheduled transformations are implicated by the candidate change. A policy engine expresses organizational rules and safety constraints governing whether a change may be executed automatically, whether an approval step is required and what compatibility measures must accompany potentially destructive modifications. A migration planner translates policy outcomes into a sequence of operations per layer, maintains ordering and preconditions and attaches idempotency keys to guarantee convergence under retries. An executor performs these operations through handler interfaces while enforcing transactional boundaries that minimize consumer-visible risk. A verifier inspects the resulting state by querying catalogs, enumerating storage objects and running targeted contract checks, then updates metadata and writes detailed audit records. Finally, the framework publishes the outcome as a schema-evolved event.

5.2.1. Change director: admission, correlation and flow control

The change director (Figure 5.5) is the authoritative ingress to the SEF. The change director admits `schema.notifications` only after validation and basic integrity checks, ensuring that

dataset identifiers are known, header snapshots are well-formed, timestamps are monotone within a tolerance window and payloads are not obviously duplicated. On admission, a notification receives a correlation identifier that threads through all subsequent stages and enables E2E tracing. The director enforces dataset-level serialization by acquiring an ephemeral lock keyed by dataset and current head version, preventing interleaving of concurrent evolution plans for the same dataset while allowing independent datasets to progress in parallel. The director also performs backpressure control via temporary queuing while the event bus retains the raw notifications for replay.

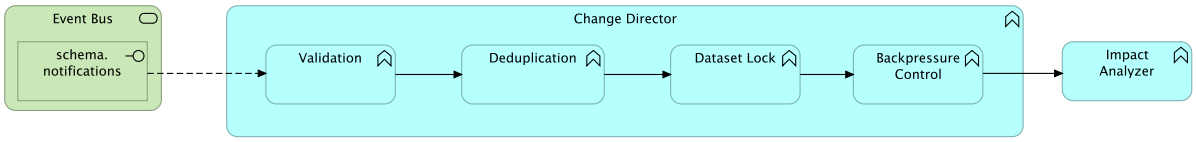


Figure 5.5.: Change director: admission and flow control

5.2.2. Impact analyzer: lineage-aware scope determination

The impact analyzer (Figure 5.6) determines the scope of a candidate change before any plan is constructed. The impact analyzer reconstructs the latest accepted schema version from the metadata repository and compares the accepted schema version with the observed header to derive a schema diff and tentative change atoms. The impact analyzer then performs lineage lookup to identify the affected artifacts across both layers. In the DL, this includes tables, registered schemas, and consumer-facing views. In the DV, this includes the relevant Satellites, affected Links, and possibly Hubs when key changes are indicated. Where lineage is available, the analyzer also inspects downstream transformations. This scope resolution informs the policy engine and reduces the risk of leaving dependent artifacts in an incoherent state.

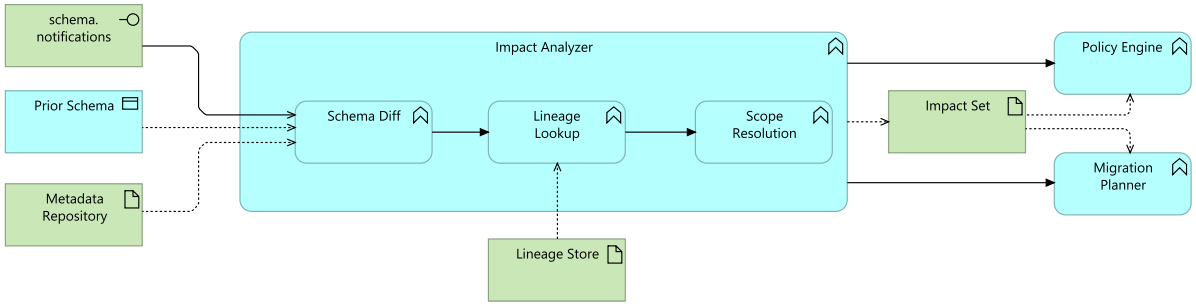


Figure 5.6.: Impact analyzer: diff, lineage and scope

5.2.3. Policy engine: governance as executable constraints

The policy engine (Figure 5.7) expresses the stance on acceptable automation. The engine evaluates rules deterministically from the candidate difference and the impact set and returns a policy outcome containing decisions, approval requirements and obligations. Additive changes and type widenings may be authorized for automatic execution, subject to environment and time-of-day constraints. Removals are treated as deprecations with retention windows and may require creation of compatibility views or aliases before any physical structure is changed. Type narrowings are flagged as potentially breaking and are either rejected outright or converted into forward-only changes that result in new Satellites while preserving historical interpretation. Rename, split and merge semantics are not inferred automatically. They require explicit operator- or governance-supplied mappings before they can enter the policy path. Because policies are pure functions of inputs, their evaluation can be simulated in dry runs.

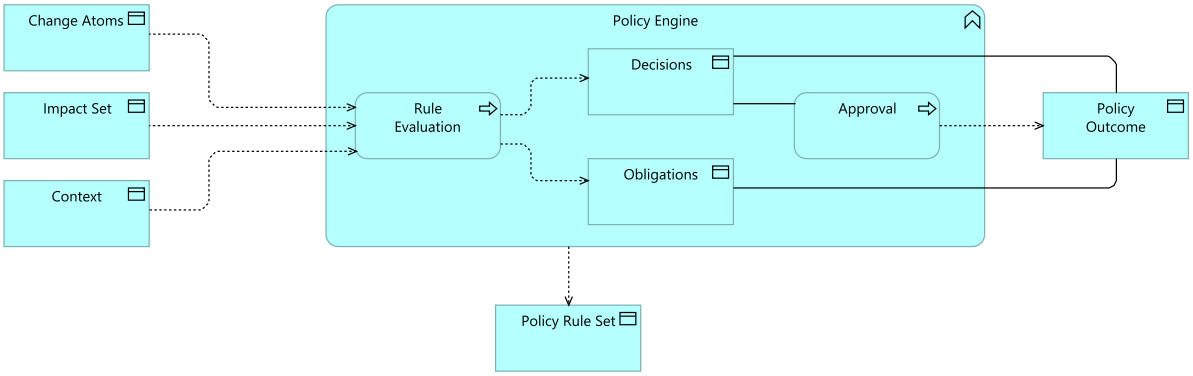


Figure 5.7.: Policy engine: rules, approvals and outcomes

5.2.4. Migration planner: from decision to ordered operation

The migration planner (Figure 5.8) translates policy outcomes into an ordered sequence of concrete operations. The migration planner builds a dependency graph whose nodes are operations and whose edges represent preconditions that must be satisfied before an operation may be executed. In the DL, the graph commonly begins with compatibility views or aliases masking impending changes from consumers, followed by metadata-only alterations where possible and finally by structural adjustments to storage objects. In the DV, the graph often introduces new Satellite versions when destructive or type-related changes would otherwise disrupt history, extends existing Satellites for purely additive changes and enforces the creation or extension of Hubs and Links when new entities or relationships appear. Each operation receives an idempotency key as a hash of target identity, operation kind and parameters,

permitting safe retries. The graph is topologically sorted, annotated with checkpoints and emitted as an ordered plan for the executor.

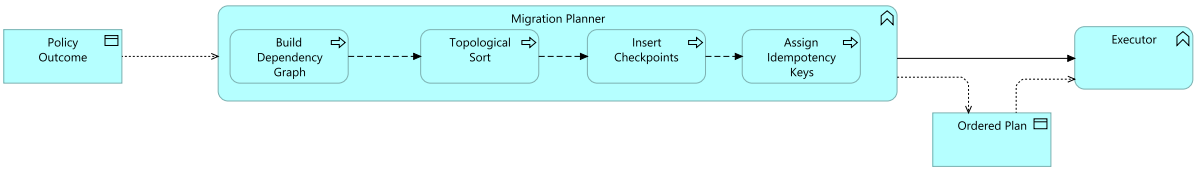


Figure 5.8.: Migration planner: dependency graph to ordered operation

5.2.5. Executor: idempotent actuation with checkpoints

The executor (Figure 5.9) advances ordered plans by fetching the next operation and invoking handler operations while preserving exactly-once effects at the level of platform state. The executor respects the plan order and pauses at checkpoints to persist progress in the execution log. On each operation, the executor supplies the idempotency key and expects the handler either to perform the change or to acknowledge that the requested change has already been performed. Transient failures (timeouts, temporary unavailability) trigger bounded retries. Permanent failures (policy violations detected late, out-of-band schema conflicts) cause the executor to halt and return control to the planner for compensating action. Compatibility views are put in place before destructive changes. Where a change cannot be performed atomically without exposing consumers to inconsistent intermediate state, the executor may also create temporary scaffolding, such as shadow columns, staging tables, aliases, or transitional compatibility views. These structures are intentionally provisional: they remain in place until downstream publication confirms consumer readiness, and they are removed only after that readiness signal where policy requires removal of the temporary scaffolding.

5.2.6. Verifier: structural and contract-based assurance

Verification confirms that the observed platform state refines the intended state (Figure 5.10). Structural verification inspects catalogs and storage to confirm that newly added columns exist with the correct logical types and nullability, that widened types have been propagated, that new Satellites are registered and reachable through views, and that compatibility layers resolve to expected shapes. Semantic probes evaluate nullability exceptions on a sample of recent records, checks that default values or backfills were applied where mandated by policy, and the lineage recheck verifies that dependent transformations remain parseable or that compatibility layers are present where policies require them. The verifier does not infer domain-semantic

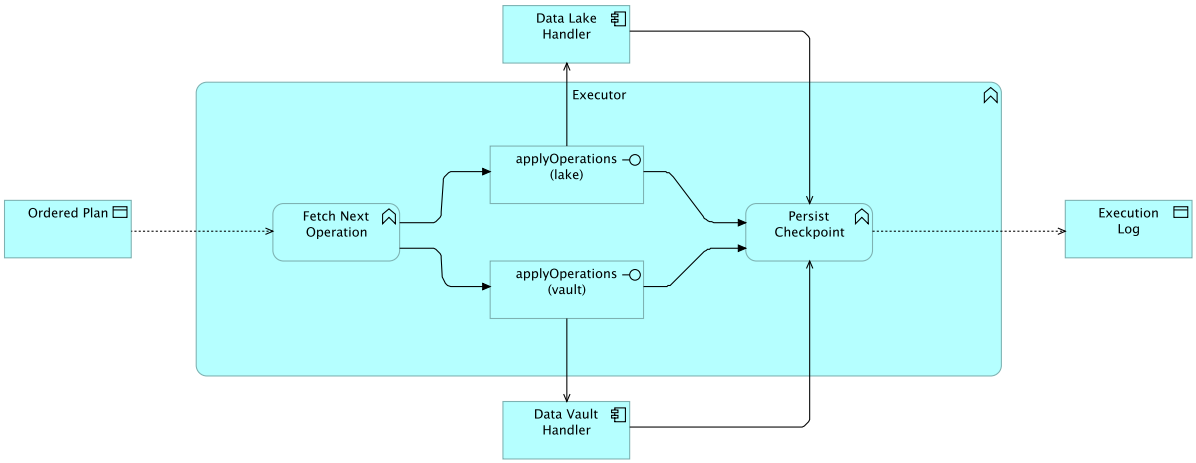


Figure 5.9.: Executor: handlers, checkpoints and retries

equivalence, renames, splits or merges. If an explicit mapping is supplied by an operator or upstream governance process, it can be checked as an input contract, but such mappings are outside the evaluated automatic inference scope. Only if all checks pass does the verifier permit publication, at which point the verifier records the new version and the verification evidence in the metadata repository.

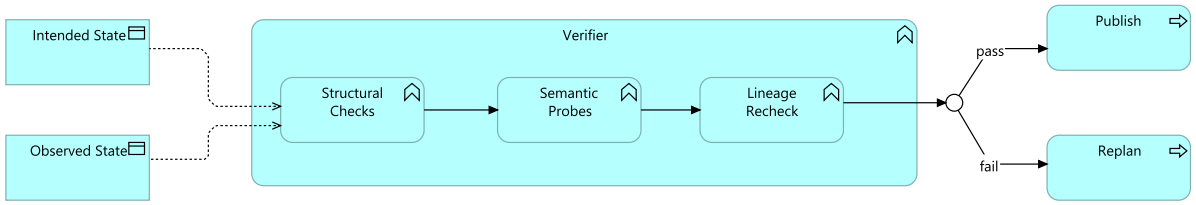


Figure 5.10.: Verifier: pipeline validation stages

The evidence snapshot pattern in Figure 5.11 makes the verifier's two evidence channels explicit. The verifier relies on two complementary channels. The first is a command-response path: when the executor completes a checkpointed slice of the plan, each handler persists an evidence snapshot that canonically describes the objects modified by that handler operation. The handler writes the evidence snapshot to the metadata repository under the same correlation and idempotency keys used during execution and returns a reference. The verifier reads those snapshots directly and treats them as the initial truth for structural checks. The second channel is live introspection, in which the verifier queries the authoritative catalogs and storage systems to cross-check that external reality matches the evidence. This dual-channel design avoids coupling while still enabling fresh reads when necessary.

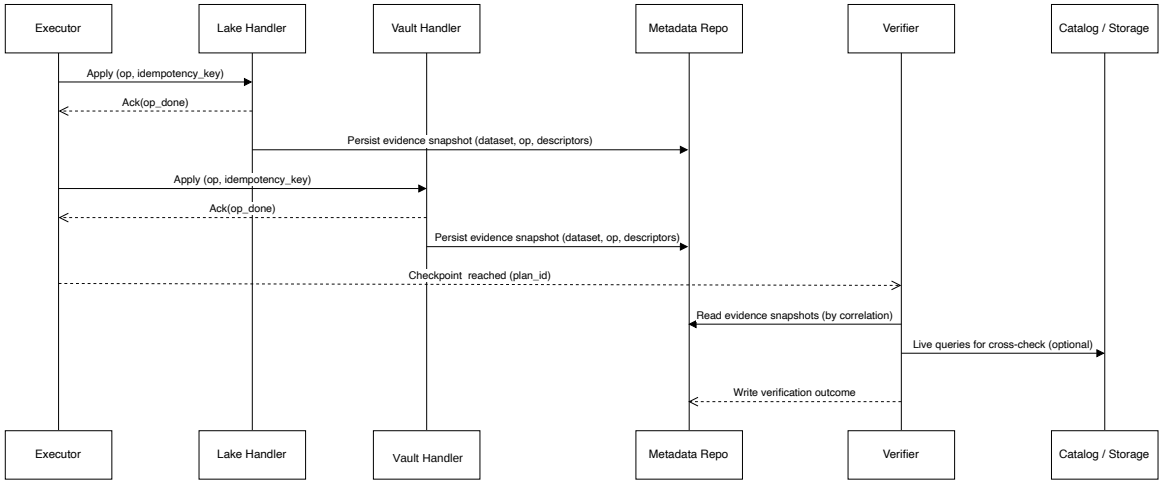


Figure 5.11.: Verifier: evidence snapshot pattern and live introspection

The command-response interaction can be realized using Representational State Transfer (REST) with JavaScript Object Notation (JSON) or via a typed Remote Procedure Call (RPC) such as gRPC. Evidence snapshots published live to a repository already trusted by the verifier eliminate any time-of-check or time-of-use race against mutable handler states.

5.2.7. Handlers, metadata repository and event bus

Figure 5.12 summarizes how `applyOperations` tasks are routed to the lake and vault boundaries. Handlers encapsulate product-specific logic behind stable interfaces so that the core framework remains technology-agnostic. The lake handler distinguishes between metadata-only operations and physical operations. The lake handler applies changes while keeping the logical catalog and storage layer aligned, and the lake handler manages compatibility views that protect consumers during transitions. The vault handler discovers Hubs, Links, and Satellites, maintains change-tracking structures such as load and end dates, and introduces new Satellite versions when destructive or compatibility-relevant changes would otherwise compromise history. Outcome publication remains reserved to the core framework so that completion guarantees stay centralized.

The metadata repository (Figure 5.13) stores immutable schema versions, plans, execution checkpoints, verification results and publications. The metadata repository supports compare-and-swap admission of new versions to enforce per-dataset monotonicity and maintains indices for rapid reconstruction of the latest accepted schema. The lineage store records dependencies between datasets, views, transformations and vault constructs, allowing the impact analyzer

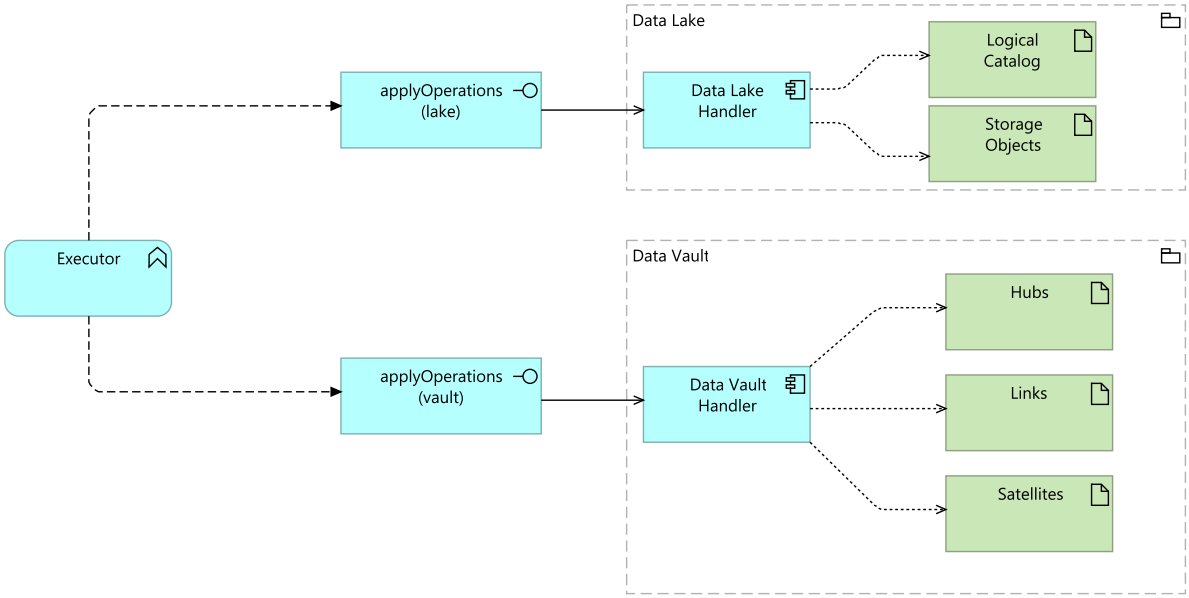


Figure 5.12.: Handlers: routing tasks to Data Vault and Data Lake.

and verifier to reason about consequences of change. The SEF refuses to accept new events when persistence is not possible, thereby preventing orphaned mutations.

The event bus (Figure 5.14) transports `schema.notifications` from the ingestion observer to the SEF and delivers `schema.evolved` publications to downstream consumers. Topics are keyed by dataset to permit scale-out consumption and retention is configured to enable replay during recovery. When an operation fails after compatibility layers have been established, consumers continue to interact with a stable contract while the executor retries or re-plans. The system prefers roll-forward (completing incomplete changes safely) over rollback, which risks reintroducing inconsistency.

Failure handling follows a retry-and-compensate discipline rather than a destructive rollback discipline. Transient faults keep the current plan in the execution state, bounded retries reuse the same idempotency keys, and completed checkpoints remain durable evidence for subsequent verification. Permanent faults stop further actuation and return the failed change to planning or policy handling, while already established compatibility surfaces remain in place until a verified forward path is available.

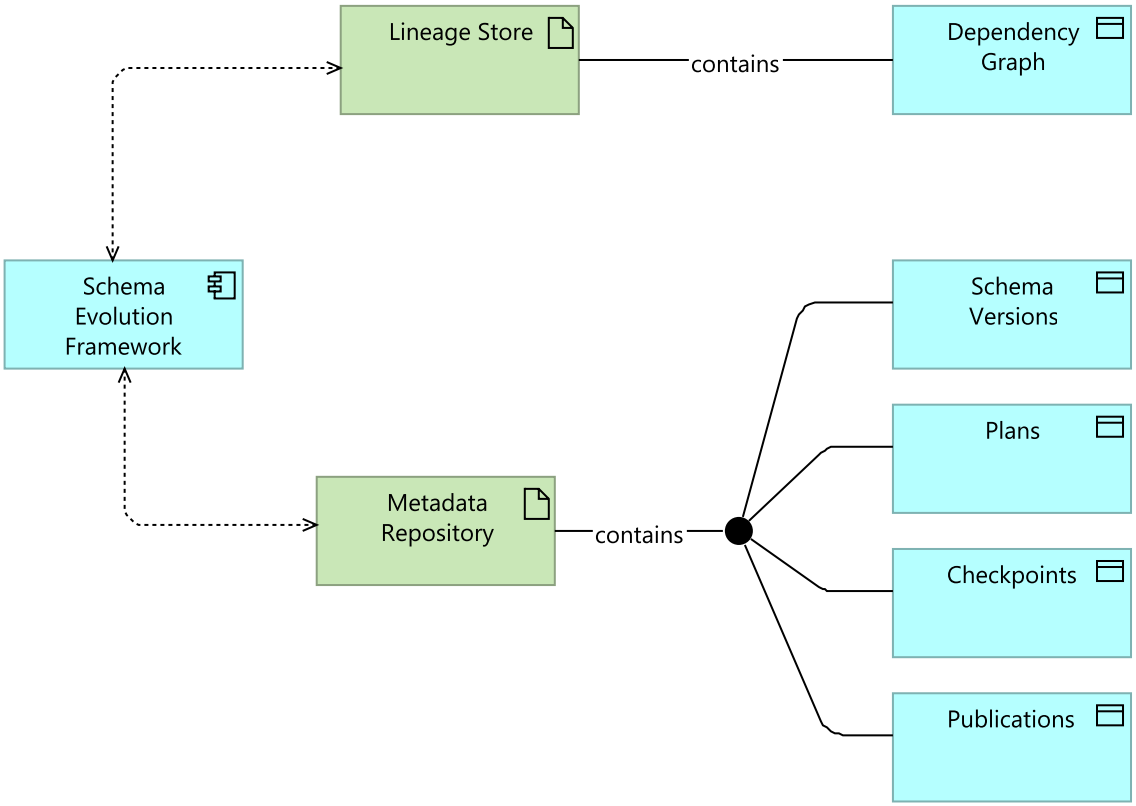


Figure 5.13.: Metadata and lineage repository.

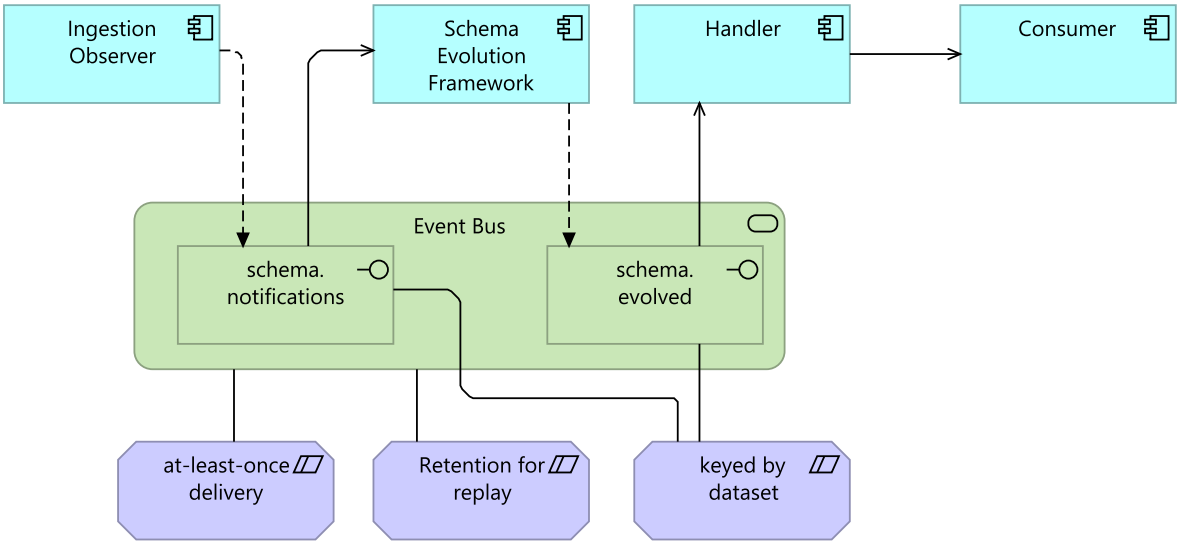


Figure 5.14.: Event bus: topics, keys, delivery and replay.

5.3. E2E control flow and processing state

This section collects the dynamic aspects of the SEF into a single narrative, complementing the component-oriented view of Section 5.2 by defining the states of processing together with their transitions, guards and actions.

At runtime, each schema change traverses a fixed sequence of interactions, intentionally linear at the dataset level to preserve serialization, while permitting parallelism across datasets. Figure 5.15 shows the path from schema snapshot or delta envelope observation to handler execution and optional publication status.

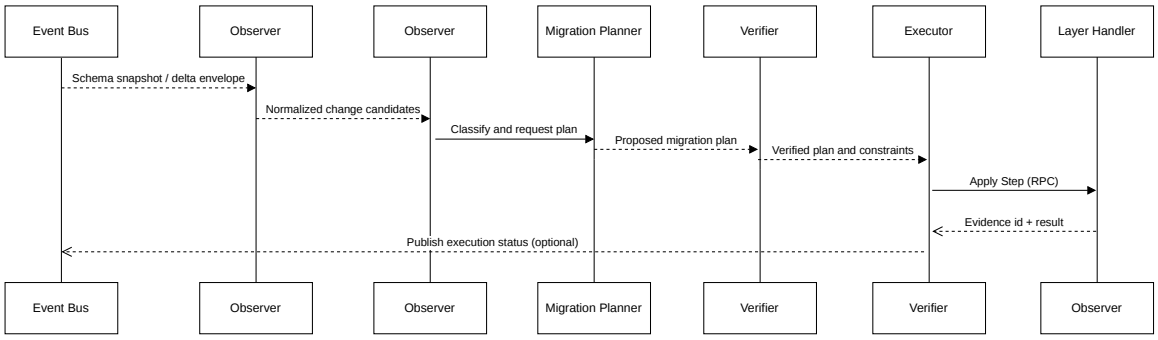


Figure 5.15.: Sequence-diagram view of the dataset-scoped notification-to-publication pipeline, emphasizing handler calls and evidence-based verification.

While Figure 5.15 describes *what* interactions occur, the state machine in Figure 5.16 specifies *when* a dataset-scoped evolution attempt progresses from observed to classified, planned, verified and executing states and which transitions are permissible under failures. The model is intentionally monotone: once an attempt has advanced to planning, verification or execution, the attempt does not regress to earlier stages. Every transition corresponds to a durable update in the metadata repository and can therefore be replayed deterministically under at-least-once delivery.

The Filewatcher transforms low-level file events into a normalized notification delivered by the event bus to the change director. The director performs validation, deduplication, dataset locking and backpressure control and delegates to the impact analyzer to reconstruct the prior schema and compute a schema diff. The analyzer performs lineage lookup and scope resolution to determine implicated artifacts and passes the impact set to the policy engine. The engine evaluates rules deterministically and returns a policy outcome with decisions (blocked or allowed), approval requirements and obligations (create compatibility views). The migration planner translates these into an ordered, checkpointed operation plan annotated with idempotency keys. The executor advances the plan by calling handlers, where each operation either performs the

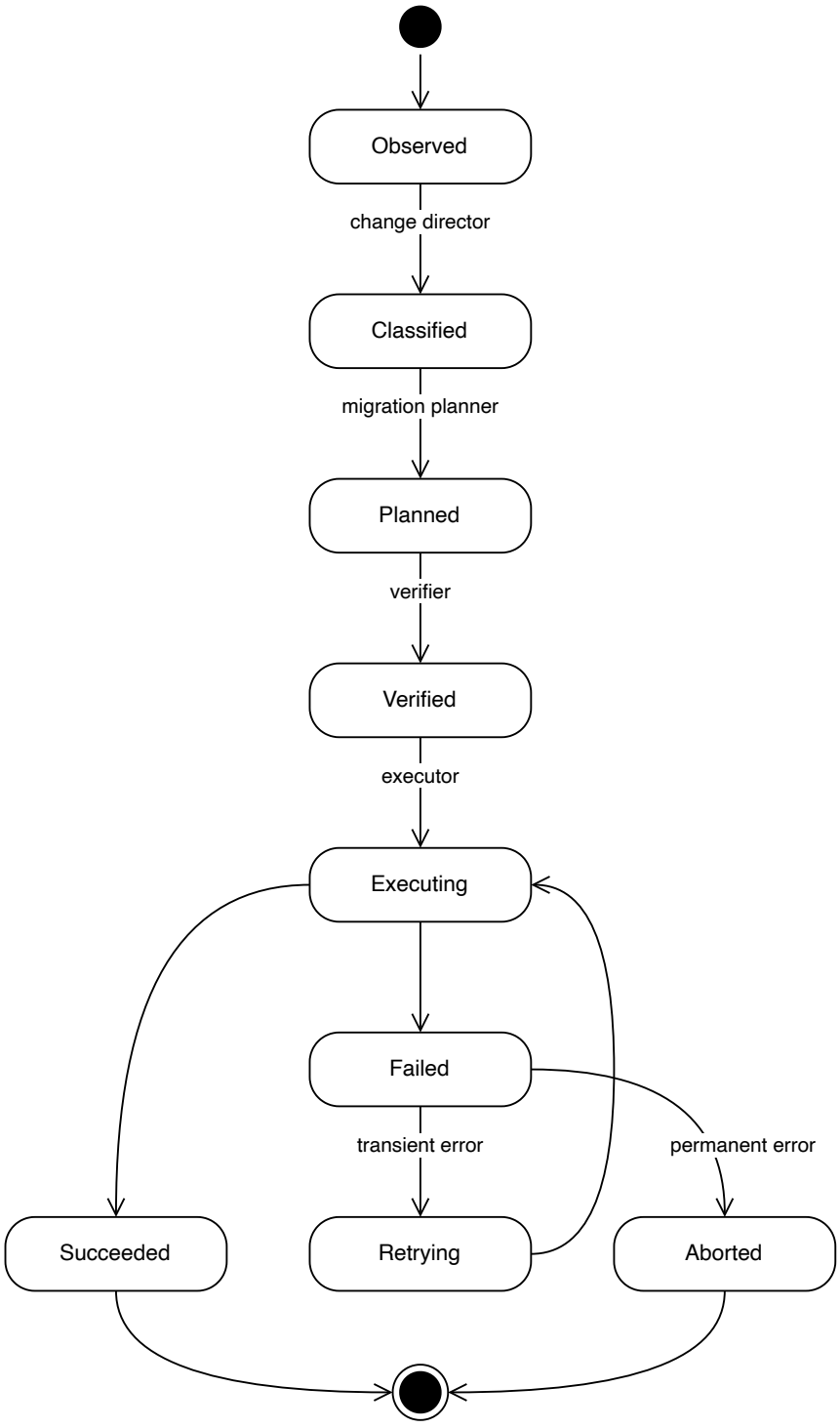


Figure 5.16.: Processing-state machine for dataset-scoped evolution in the SEF, including retry and abort transitions under transient and permanent failures.

change or returns idempotent success. After a checkpoint boundary is reached, the verifier compares intended and observed states using structural checks, semantic probes and a lineage recheck. On success, the publisher records the new version and emits a compact *schema.evolved* event for downstream orchestration.

5.4. Operational model and contracts of the SEF

This section specifies the SEF in terms of the framework operational records, transition discipline and verification conditions, providing an unambiguous contract between the core, the metadata repositories and the layer handlers. The specification remains technology-agnostic at the level of concepts while aligning terminology and field names with the reference implementation.

5.4.1. Core records and message families

A dataset is identified by a stable logical identifier (`id`) and may carry contextual qualifiers such as a `source` system label and an ingestion `zone`. The tuple is treated as the routing key for both persistence and governance scope.

A header snapshot is the schema posture observed at the ingestion boundary and consists of an ordered list of attribute descriptors (`attributes`), an optional primary key list (`primary_key`) and an optional fingerprint (`schema_fingerprint`) computed by the producer. Each attribute descriptor contains at minimum `name`, `logical_type`, `physical_type` and `nullable`. The SEF treats `logical_type` as the effective type for change classification when the logical type is present.

The ingestion layer emits `schema.notification` events. Each notification contains the dataset descriptor, the header snapshot, observation metadata and an optional hint (`previous_schema_version`). Notifications are treated as *proposals*: a proposal becomes effective only after planning, idempotent execution and evidence-based verification.

5.4.2. Admission and correlation

Before change processing, the change director performs admission checks: `event_type` correctness, presence of `dataset.id`, presence of `header.attributes` and presence of `ingestion.observed_at`. On admission, the SEF generates a fresh correlation id and binds the correlation id to all subsequent artifacts for auditability. Platform-managed technical columns (by

default `ingestion_timestamp`) are excluded from both the previous and new header snapshots, preventing internal technical fields from inducing false-positive drift.

5.4.3. Versioned metadata repository

For each dataset, the SEF maintains a monotonically increasing sequence of schema versions. Each stored version records the dataset identifier, version number, header snapshot, correlation identifier, and storage timestamp. Two invariants define the repository contract. Versions increase strictly per dataset and are never reused. Once recorded as published, a version's header snapshot is not mutated. The artifact supports both a relational store and a filesystem-backed store with append-only JSON records. Both backends are constrained by the same logical contract. The concrete schema is documented in the online technical supplement [12].

5.4.4. Change classification into atomic change statements

Given the last accepted header snapshot and an observed snapshot, the impact analyzer computes a name-based difference and reduces drift into a finite set of atomic change statements: `ADD_COLUMN(name, to_type)`, `DROP_COLUMN(name, from_type)` and `CHANGE_TYPE(name, from_type, to_type)`. Type tokens are normalized into a small alias set to avoid superficial differences from creating spurious type-change events. The SEF does not infer renames, splits or semantic mappings automatically: these are treated as governed, explicit interventions rather than implicit drift repair.

5.4.5. Policy evaluation and obligations

Policy evaluation is deterministic: for a fixed set of change statements, a fixed impact scope and a fixed policy configuration, the policy engine produces the same outcome. An outcome includes a decision (`allow` or `block`), a compatibility classification such as `backward` or `breaking` and a set of obligations that refine how the planner must act. The default policy is safe-by-default: the default policy allows purely additive changes and blocks destructive drops and type changes. A sandbox policy may permit these evolutions. For type changes, the policy engine can optionally enforce `widen_only` semantics, allowing only widening conversions such as `integer` to `float` and blocking non-widening changes.

The default rule set is summarized in Table 5.1 for reference.

Table 5.1.: Default policy rule set: decision and obligations by change class under safe-by-default (`production`) operation.

Change class	Default decision	Default obligations
`ADD_COLUMN`	`allow`, `backward`	`NEW_HUB`, `NEW_LINK` (if candidates present), `ADD_COLUMN` on lake and vault
`CHANGE_TYPE` (widening)	`allow`, `backward` only if widening policy is configured, otherwise `block` under safe-by-default	`CHANGE_TYPE` on lake and vault (if allowed)
`CHANGE_TYPE` (narrowing / breaking)	`block`, `breaking`	none
`DROP_COLUMN`	`block`, `breaking`	none

Policy sets are defined in a configuration file and selected at startup via an environment variable, or per-notification via `metadata.policy` in the incoming schema notification. The configurable fields and the concrete `policies.yaml` format are documented in the online technical supplement [12].

5.4.6. Planning, ordering, checkpoints and idempotency keys

Planning translates admitted drift into an ordered operation plan spanning the lake and vault layers. A plan contains: `plan_id`, `dataset_id`, `correlation_id`, an ordered list of operations and a checkpoint list. Each operation specifies: `layer` (`lake` or `vault`), `kind`, `target`, `params` and a deterministic `idempotency_key`.

The idempotency key is computed as a stable hash over a canonical JSON serialization of the operation identity tuple (dataset, layer, kind, target, parameters). This ensures that repeated deliveries and retries converge: handlers can detect that the requested effect is already durable and respond with `ALREADY_APPLIED`. Checkpoints encode coarse-grained transaction boundaries, allowing the executor to persist progress and enabling the verifier to reason about partial completion in a controlled manner.

5.4.7. Execution and verification semantics

The executor applies operations sequentially through the corresponding handler interfaces. `OK` denotes that the effect has been committed and is observable. `ALREADY_APPLIED` denotes idempotent success because the requested effect already holds. `TRANSIENT_ERROR` denotes a

retryable failure and `PERMANENT_ERROR` denotes a non-retryable failure. The executor performs bounded retries with backoff for transient failures and aborts on permanent errors.

Verification is evidence-based and combines two steps. First, execution trace checks ensure that all operations reached `OK` or `ALREADY_APPLIED`. Second, live introspection queries handlers for authoritative descriptors of the current lake and vault state and compares them against plan intent. Only after verification passes does the framework publish a `schema.evolved` event and record the new schema version.

5.4.8. Safety properties

The operational model establishes the following safety properties:

1. **Decision-before-actuation:** no mutation is executed unless policy admits the change and a plan is persisted.
2. **Deterministic planning:** operation identities and idempotency keys are stable for equal inputs.
3. **At-least-once delivery, exactly-once effect:** transport may replay notifications, but durable platform state converges because operations are idempotent and verified.
4. **Evidence-gated publication:** success is announced only after external state matches intended post-state.

5.5. Layer handlers and behavioral contracts

Handlers are the adaptation surface turning the SEF into layer-specific actuation, isolating the core framework from product Application Programming Interfaces (APIs), physical storage formats and modeling conventions while providing strong behavioral guarantees enabling retries, verification and auditability.

5.5.1. Unified handler interface and operation model

Both the lake handler and the vault handler expose two capabilities: they apply a batch of operations and they provide authoritative evidence via introspection. The shared RPC interface is defined as a stable contract independent of specific lakehouse or DV implementations. The operation envelope provides identity and traceability, routing (`layer`), intent (`kind`), target

identifiers and parameters. This structure allows the executor and verifier to treat handlers uniformly while still enabling layer-specific implementations.

5.5.2. Behavioral contract: idempotency and status semantics

The core assumes the following handler obligations:

1. **Idempotency by key:** For a fixed `idempotency_key`, repeated invocations must converge to the same durable platform state. If the effect already exists, the handler must return `ALREADY_APPLIED` rather than performing a duplicate mutation.
2. **Status is authoritative:** The returned status must reflect whether the requested effect is durable and visible to introspection: `OK` indicates committed and observable state, `ALREADY_APPLIED` indicates the effect already holds under the same idempotency key, `TRANSIENT_ERROR` indicates a retryable failure and `PERMANENT_ERROR` indicates a non-retryable failure.
3. **Evidence snapshot emission:** On `OK` or `ALREADY_APPLIED`, the handler must return an `evidence_snapshot_id` that can be used by the verifier to retrieve canonical post-state descriptions.

These contracts enable the exactly-once effects property: delivery may be at-least-once, but actuation converges.

The introspection endpoint returns structured descriptors of current state: tables for the lake and Hubs, Links and Satellites for the vault, plus an optional raw evidence payload. The verifier uses this to cross-check that execution results match externally visible state without binding the SEF to a specific catalog technology.

Although the interface is uniform, operation semantics are layer-specific. The lake handler mediates between logical schema intent and physical state, distinguishes metadata-only alterations from operations requiring physical rewrites and ensures that introspection reflects the authoritative catalog view. Compatibility layers such as views are handled at this boundary so that consumer shielding remains a layer concern while decision logic stays centralized. The vault handler preserves DV invariants such as auditable, time-variant history, correct hub, link and Satellite structure and safe evolution strategies when the source contract changes. The policy layer signals structure alignment intent through obligations such as `NEW_HUB` and `NEW_LINK` and the planner emits corresponding vault operations while delegating the already-satisfied versus create versus side-by-side choice to the handler.

5.5.3. Cross-layer ordering constraints and publication centralization

The plan order is authoritative across layers. Handlers must apply operations according to the order they receive within a batch and must not publish outcomes directly. Publication is centralized in the core, so that compatibility classification and verification gating remain single-sourced.

6. Filewatcher service

Figure 6.1 shows where the Filewatcher sits between CSV files and the event bus. The Filewatcher forms the ingestion-side observation boundary of the overall data pipeline. The Filewatcher monitors a directory for file changes, derives record deltas under an append-only producer assumption and publishes those deltas as bounded JSON envelopes to an event bus. In addition, the Filewatcher emits normalized `schema.notifications` conforming to the SEF ingestion contract, enabling the SEF to detect and govern schema drift without coupling to file-system specifics. This chapter presents the technology-agnostic conceptual design. The concrete reference implementation is provided in the online technical supplement [12].

6.1. Conceptual overview

At the producer boundary, the system needs two signals: the arrival of new records and the current structural shape of those records. A file-based producer does not emit either signal explicitly. Instead, the observer sees file-system events such as *created* or *modified*. The Filewatcher treats these events as hints and turns them into explicit messages on an event bus. The service is restricted to observation, normalization and publication.

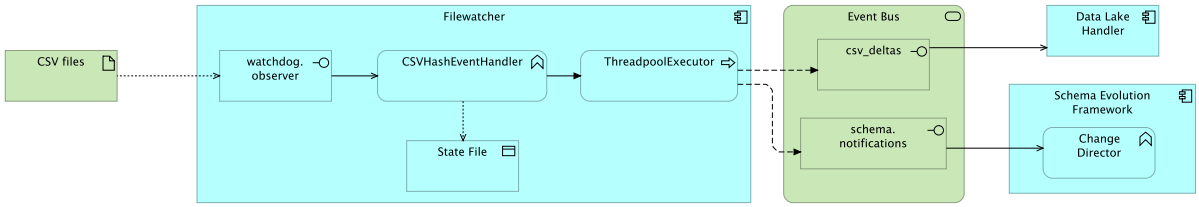


Figure 6.1.: Filewatcher placement and publication surfaces.

6.1.1. Dataset identity and deterministic routing

The Filewatcher publishes two independent streams: record deltas representing newly observed rows and schema notifications representing snapshot proposals of the current schema posture.

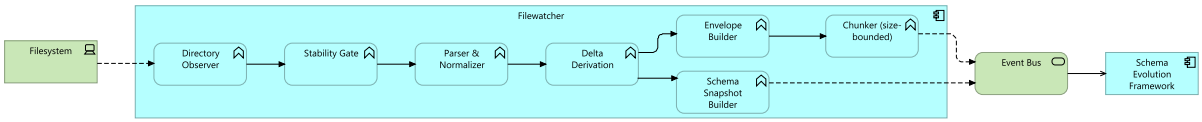


Figure 6.2.: Conceptual processing flow of the Filewatcher: observation, normalization, schema notification and delta publication.

Both streams are keyed by a stable dataset identifier derived from the file name. The identifier is complemented by a small descriptor covering source system and zone so that downstream systems can route the message deterministically without relying on absolute paths or host-specific directory layouts. Consumers can map the dataset id to a table or object of the same name and maintain per-dataset isolation.

6.1.2. Delta semantics, restart safety and bounded publication

The central observation task is to translate file modifications into new rows. For each observed file the service maintains a persistent high watermark representing how many data rows have already been published. On a new event, the service reads the current file contents and emits only the rows beyond the high watermark. After successful publishing, the high watermark is advanced to the new end-of-file position. This deliberately assumes that producers append rows: if a producer truncates files, rewrites earlier rows or performs in-place edits, an offset-based delta cannot reliably reconstruct intent. The system therefore targets ingestion pipelines in which append-only export semantics are acceptable.

Offsets are persisted to local state, providing continuity across restarts. The resulting E2E behavior is at-least-once at the record level: a failure can occur after an event-bus publish but before state persistence, allowing a subsequent restart to re-emit the last delta segment. Downstream consumers are therefore expected to tolerate duplicates via checkpointing or deduplication.

Event buses typically impose message size limits. The Filewatcher enforces a maximum payload size and segments large deltas into multiple envelopes. Segmentation is driven by a row-based chunk size but is guarded by a byte-size check after serialization. If a chunk exceeds the bound, the Filewatcher subdivides the chunk until the chunk fits. If even a single-row envelope exceeds the bound, the Filewatcher skips and logs that row to avoid deadlocks.

In addition to record deltas, the Filewatcher emits a schema snapshot as a SEF input proposal. A snapshot is derived from the header and a bounded sample of rows, carrying per-attribute descriptors (name, inferred logical type, physical type and nullability), an optional primary-key

list and a stable fingerprint over the descriptor list. This allows the SEF to detect schema drift by comparing fingerprints and attribute sets while keeping the framework decoupled from the file format and monitoring mechanism.

6.2. SEF integration contract

The properties described in Section 6.1 are not independent design choices. Together they define a producer-side contract that the SEF depends on for correct schema drift detection and governed actuation.

The Filewatcher publishes two independent message families on the event bus, both keyed by the same stable dataset identifier. Row-delta envelopes are data plane messages consumed directly by the lake handler. Schema notifications are control plane messages consumed exclusively by the SEF. Each notification carries per-attribute descriptors, an optional primary-key hint and a stable fingerprint. The two families are published independently and are not causally coupled at the transport level, so the SEF cannot assume that a schema notification and the row-delta envelopes described by the schema notification have been delivered in any particular order relative to each other.

Figure 6.2 summarizes the producer-side flow from directory observation through schema snapshot and record-delta publication. The producer-side contract rests on four guarantees. The dataset identifier is derived deterministically from file metadata and remains stable across restarts and re-deliveries, so the SEF can treat the dataset identifier as a durable routing and versioning key. The schema snapshot reflects the structural posture of the file at observation time. The SEF must not assume that the snapshot is consistent with any specific subset of delta envelopes already in transit. If the file structure has not changed since the last observation, the fingerprint will be identical, allowing the SEF to skip re-classification without inspecting attribute lists. Finally, schema notifications carry at-least-once delivery semantics and may be re-delivered following restarts. The SEF admission check handles this by comparing the observed fingerprint against the last accepted version, so an unchanged fingerprint results in a no-op rather than a spurious plan.

The SEF has no visibility into file-system layout, directory structure, file format specifics or producer clocks, and does not receive any signal when a file is deleted or when a producer stops emitting. The scope of automated schema evolution is therefore strictly bounded to changes observable from the schema notification surface: additions, removals and type changes of named attributes.

7. Layer handlers

This chapter documents the two layer-specific handler services introduced in subsection 5.2.7 for the DL and DV. Both handlers implement the unified behavioral contract defined in Section 5.5 and share the same status semantics (`OK`, `ALREADY_APPLIED`, `TRANSIENT_ERROR`, `PERMANENT_ERROR`). Reference implementations for both services are provided in the online technical supplement [12].

7.1. Data Lake Handler

The DL handler acts as the lake-side ingestion and actuation endpoint: the handler consumes record-delta envelopes from the event bus produced by the Filewatcher and persists them into a durable, queryable lake backend. The lake handler's gRPC control plane surface implements the Lake Handler contract through which schema operations are applied idempotently and authoritative schema state is returned as verification evidence.

7.1.1. Conceptual overview and abstract lake state

Figure 7.1 positions the Data Lake Handler as a colocated ingestion and actuation boundary. The Data Lake Handler provides a stable persistence surface for downstream consumers by translating `csv_deltas` into table-scoped lake objects. Conceptually, the Data Lake Handler decomposes into a data plane and a control plane. The data plane consumes delta envelopes, routes them deterministically to a logical table and appends them to a durable table object. The control plane exposes `applyOperations (lake)` for schema actuation and `introspectEvidence (lake)` for introspecting the authoritative lake schema as verification evidence. The separation is a correctness requirement: the data plane operates under a continuous, low-latency ingestion contract and must not be blocked by governance decisions, so schema operations are applied between micro-batches rather than within them. Readiness races, where a schema evolution operation arrives before the handler has materialized the target table, are treated as transient errors, allowing the SEF executor's bounded retry with backoff to resolve them.

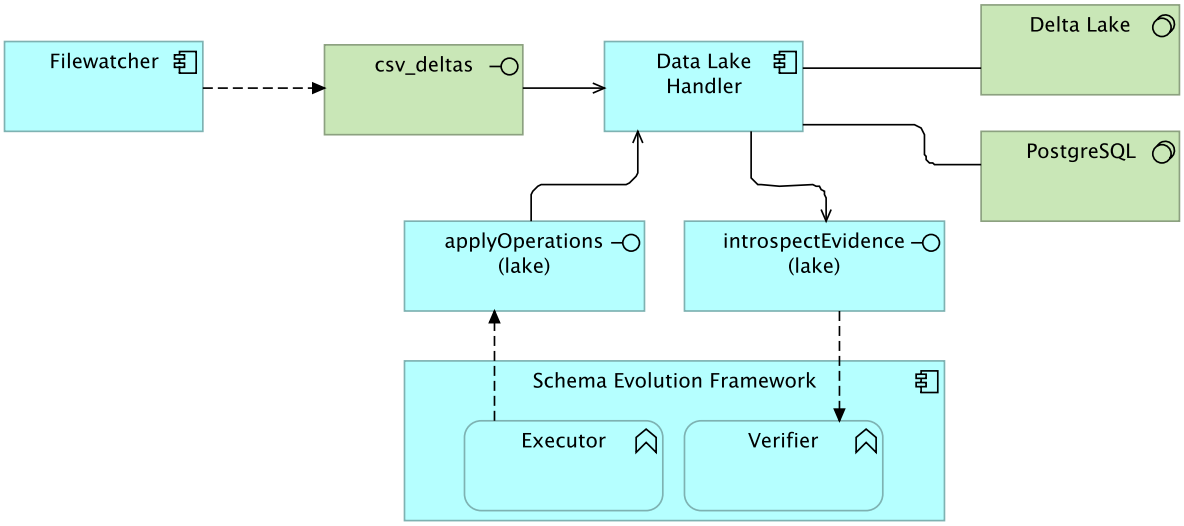


Figure 7.1.: Data Lake Handler architecture and colocated SEF boundary.

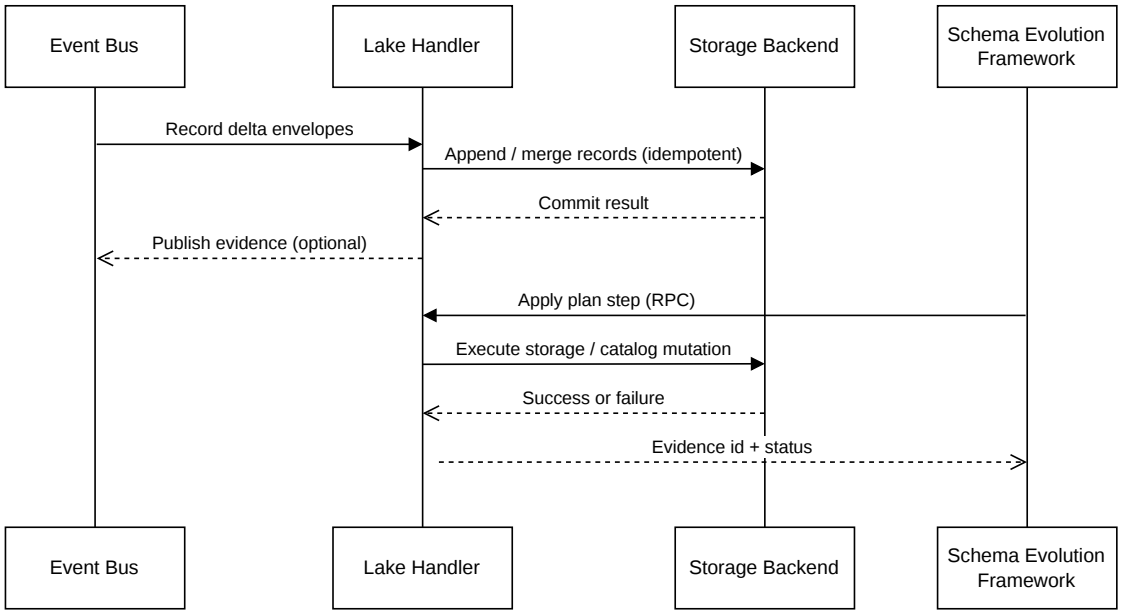


Figure 7.2.: Control plane actuation and evidence exchange of the Data Lake Handler.

Figure 7.2 focuses on the handler's control-plane actuation path: the SEF sends an apply-plan step by RPC, the handler applies the change idempotently and the handler returns an evidence id and status for verification. The lake can be viewed as a set of logical tables, each maintaining a durable schema and a set of persisted rows. Under append-first persistence, incoming rows are appended to the target table without primary-key enforcement, in-place update reconciliation or business-key deduplication, so the lake acts as a raw persistence layer. Restart safety is provided via checkpointed processing positions in streaming mode and explicit position commits in bulk mode, but failures and retries can still cause duplicates, making at-least-once the correct characterization of the record-level delivery guarantee.

Every persisted row carries an `ingestion_timestamp` added by the handler at write time. This field is not a business attribute and is excluded from schema change classification by the SEF. The timestamp field serves as an audit anchor allowing any row to be correlated with the ingestion episode that produced the persisted row and with the schema version active at that ingestion time.

7.1.2. Correctness goals and invariants

The handler is guided by three invariants that are independent of the backend technology. The first invariant is **deterministic routing**. Each message is mapped to exactly one logical table using stable metadata, such as the dataset identifier. This ensures that retries always route to the same target. The second invariant is **table isolation**. Schema drift must be contained to the originating table. Attributes introduced by records for one table must therefore not leak into another table merely because the records co-occur in a batch. The third invariant is **monotone additive evolution by default**. In the absence of explicit SEF-driven schema operations, the handler avoids destructive changes. The handler accommodates additive columns while deferring removals and breaking changes to governed actuation.

Table isolation is the key constraint: streaming ingestion frequently processes mixed-entity micro-batches and naive schema merging can turn incidental drift into global structural change. The monotone additive default follows from the same logic applied to time: removing a column destroys the ability to query historical rows that carried the removed column.

The lake handler is the first dependency in the DV handler's startup sequence. The vault handler blocks on lake readiness before spawning any worker threads and the vault handler's initial table discovery reads the lake catalog to determine which Bronze tables already exist and which still need bootstrapping. Schema operations applied to the lake via the SEF control plane automatically become visible to the next vault modeling cycle without requiring a separate notification. Evidence-based verification closes the loop: the SEF verifier confirms that what

was applied in the lake matches what the vault handler will observe during the next modeling cycle.

The handler supports two operational modes. Streaming mode provides continuous ingestion with low latency by processing micro-batches as they arrive. Bulk mode drains messages in controlled chunks with an emphasis on robustness and recovery. The concrete state machine governing these transitions is documented in the online technical supplement [12].

7.1.3. Handler contract and SEF integration

The lake handler exposes an operation surface through which the SEF planner applies schema changes. Three operation kinds are relevant. `ADD_COLUMN` introduces a new named attribute with a specified logical type. For existing rows the new column is implicitly `NULL`. The operation is safe to replay because the handler returns `ALREADY_APPLIED` if the column is already present with the correct type. `CHANGE_TYPE` alters the logical type of an existing attribute. Under the default policy, only type-widening conversions are executed. Non-widening conversions are blocked at the policy layer before reaching the handler. `DROP_COLUMN` removes an attribute from the governed schema. The `DROP_COLUMN` operation is blocked under the safe-by-default policy and is included to define the handler's expected behavior when a sandbox policy explicitly permits column removal.

For all operation kinds the handler must conform to the status semantics defined in Section 5.5.

After applying an operation, the SEF verifier calls the handler's introspection endpoint and compares the live catalog state against the plan's intended post-state. Two properties are essential for verification correctness. First, **post-write visibility**: the introspection response must reflect the effect of operations already returned as `OK`. A backend that buffers schema changes asynchronously without making them visible in introspection before acknowledging success will cause verification failures. Second, **consistency with ingestion**: the schema returned by introspection must be the same schema the data plane ingestion path uses for routing and column-set determination. If the catalog view diverges from the ingestion view, then verification evidence and ingested data will be inconsistent. This is precisely the failure mode the evidence-based completion model is designed to prevent.

The SEF planner determines which operations to issue. The lake handler applies them. The handler makes no governance decisions. The handler does not inspect policy, compute diffs or emit schema notifications. The handler's sole control plane obligations are to apply operations idempotently, report status truthfully and return authoritative introspection on demand.

7.2. Data Vault Handler

The DV handler is a prototype component realizing the vault-side actuation and evidence surface of the SEF architecture. In contrast to the lake handler, which focuses on durable persistence, the DV handler is an integrative orchestrator: the DV handler binds ingestion, modeling and lineage preservation into a reproducible system of record. The concrete reference implementation is provided in the online technical supplement [12].

7.2.1. Conceptual overview and scope

The Data Vault Handler constitutes the modeling boundary between raw, schema-tolerant persistence and a historized analytical representation. Schema evolution at this layer cannot be treated as a purely syntactic concern: structural drift directly affects historization behavior, key stability and the interpretability of derived entities over time.

The handler consumes insert-only change events and materializes two categories of outputs: an immutable Bronze persistence layer used for reproducible reprocessing, and derived DV artifacts that expose stable business entities and relationships with historized descriptive attributes. The DV handler additionally exposes control plane interfaces required by the SEF to apply structural operations and to produce verification evidence.

On the data plane side, the handler ingests insert-only CDC events, validates their conformance to the boundary contract and maintains a monotonic processing model under at-least-once delivery, persisting those payloads into an immutable Bronze staging layer that enables deterministic reprocessing. On the control plane side, the DV handler derives DV structures from Bronze according to deterministic modeling rules and schema evolution constraints, and receives and applies SEF operations that modify structural aspects of the vault.

Out of scope are semantic transformations requiring domain interpretation, such as inferring that an attribute rename corresponds to an unchanged business concept, and irreversible destructive changes that would invalidate historized interpretation. Such changes must be expressed explicitly through non-destructive operations or implemented manually.

The primary input is an insert-only CDC stream. Each CDC event comprises a dataset identifier, a change indicator, a modification timestamp and a payload representing the post-change state. The stream is append-only and replayable, meaning reprocessing the same events should converge to an equivalent Bronze and Vault state. The handler's processing pipeline has three conceptual stages. In **event intake**, CDC events are consumed from the event bus, validated and associated with processing-time metadata required for replay and observability. In **schema**

reconciliation and Bronze persistence, the payload is reconciled with the target Bronze schema under deterministic rules: missing attributes become `NULL` while unexpected attributes may be preserved in a generic payload representation. In **vault artifact derivation**, vault structures are derived from Bronze using deterministic modeling rules, with the critical requirement that this derivation remains stable under reprocessing and at-least-once delivery so that duplicates and retries do not lead to non-convergent historization.

7.2.2. Modeling invariants and schema evolution semantics

The DV layer imposes invariants that differ qualitatively from those of the DL. **Insert-only historization** requires that time-variant descriptive information be represented through append-only records, enabling point-in-time reconstruction without overwriting historical states. **Stable identity** requires that the criteria used to identify business entities and relationships remain stable across schema versions. **Control plane separation** requires that operational metadata such as ingestion timestamps be excluded from descriptive attribute sets rather than conflated with business attributes. **Idempotent processing** requires that under at-least-once delivery repeated processing of the same CDC event stream converges, demanding deterministic keying and duplicate-tolerant materialization strategies.

The handler adopts a non-destructive evolution stance expressed through three principles. Additive evolution is supported by extending existing descriptive structures or introducing new ones. Additions preserve historical interpretability because prior records remain valid and new attributes can be `NULL` for older states. Widening conversions are considered safe where they preserve all previously representable values, while narrowing conversions and lossy coercions are rejected or require explicit externally curated transformations. Semantic drift such as renames, splits, merges and semantic re-interpretations is not inferred automatically. Instead, semantic drift must be expressed as explicit additions and deprecations to avoid silently corrupting historized meaning.

A conceptual mapping between SEF operations and their vault-level intent is shown in Table 7.1.

To support SEF verification, the handler must provide observable evidence that an operation has been applied. This evidence includes structural introspection covering the existence of expected entities and relationships as well as attribute presence and types. Verification evidence also includes schema snapshots as immutable point-in-time captures comparable against expected post-operation states, and operational metadata tracing the link between a control plane request and the observed structural changes produced by that control plane request. The concrete evidence channels are specified in the online technical supplement [12].

Table 7.1.: Conceptual intent of SEF operations when applied to the Data Vault layer.

Operation Category	Vault-Level Intent (Conceptual)
Introduce entity structure	Create a new stable representation for a business entity (Hub) with explicit identity criteria.
Introduce relationship structure	Create a new stable representation for an association (Link) between entities, preserving relationship identity.
Extend descriptive attributes	Add new non-key attributes to historized descriptive structures (Satellites), preserving prior states.
Widen attribute type	Permit a broader representation while preserving all historical values (non-lossy widening).
Deprecate attribute/structure	Mark an element as no longer used for future loads while keeping historical data intact.

7.2.3. Handler contract and SEF integration

The vault handler exposes the same dual-capability interface as the lake handler. The vault handler provides an operation application endpoint and an introspection endpoint. The SEF planner issues vault operations as part of a cross-layer plan in which lake operations are ordered before their vault counterparts, ensuring that Bronze data is available before vault structures are materialized.

Three properties are required regardless of which specific operation is being applied. First, **non-destructive by default**: the handler must not silently remove or overwrite historized data. Additive operations extend existing structures while destructive changes are only executed when explicitly permitted by policy and must leave prior historical records intact. Second, **idempotency by operation key**: repeated application of an operation with the same idempotency key must converge to the same vault state. The handler must return `ALREADY_APPLIED` rather than `OK` when the requested structural effect already holds. This matters especially because the modeling rules that derive Satellites and Links may inspect existing structures before deciding whether to create or extend them. Third, **stable identity preservation**: operations that introduce new entity or relationship structures must use the business-key specification provided by the SEF planner. The handler must not silently substitute or reorder key attributes, as doing so would produce a structurally distinct entity that breaks the identity contract with prior and future loads.

Following plan execution, the SEF verifier calls the handler's introspection endpoint and compares the returned structural descriptors against the plan's intended post-state. The vault handler's introspection must enumerate existing Hub, Link and Satellite entities by name with their key attribute specifications, report the current set of descriptive attributes with their names and logical types for each Satellite, and indicate when each structure was last modified so that the verifier can distinguish pre-existing structures from newly materialized ones.

The vault handler has a structural dependency on the lake handler: the vault handler reads the lake catalog during bootstrapping to determine which Bronze tables exist and which still require initialization. Lake-layer operations in a plan must therefore be executed and verified before the SEF issues vault-layer operations for the same dataset. The vault handler can assume that a Bronze table for a given dataset already exists when a vault operation targeting that dataset arrives.

The presented approach assumes that the CDC input is insert-only and replayable and that identity criteria for entities and relationships are either stable or explicitly configured. Semantic mappings for ambiguous drift classes such as renames, splits and merges are not inferred, as incorporating such mappings would require domain knowledge and risk silently incorrect historization. Non-destructive evolution is prioritized over aggressive consolidation.

8. End-to-end example

This chapter provides a concrete, E2E example that ties together the runtime architecture (Chapter 4), the SEF core (Chapter 5) and the layer-specific ingestion handlers (Chapter 6, Chapter 7). The intent is to make the cross-chapter contracts and invariants operationally tangible by tracing a scenario through detection, policy evaluation, planning, execution and retry semantics. The example reuses the conceptual objects introduced earlier, covering dataset identity, schema notification, atomic change statements, policy outcomes and ordered operation plans, and shows how they interact as a single trace.

Full concrete implementation artifacts for this scenario, including delta envelopes, schema notification JSON, the migration plan with deterministic idempotency keys, the link-probe response and handler introspection payloads, are provided in the online technical supplement [12]. The conceptual trace in this chapter is designed to be self-contained without those listings.

Before turning to the scenario trace, Figure 8.1 presents a sequence-diagram view spanning the Producer, Filewatcher, Event Bus, SEF, Registry / State Store, Lake Handler, Vault Handler, Lake Storage and Vault Storage. The diagram is intentionally technology-agnostic: the diagram characterizes ordering constraints, idempotent execution through handler interfaces and evidence-gated publication, without relying on any product-specific transport or processing semantics.

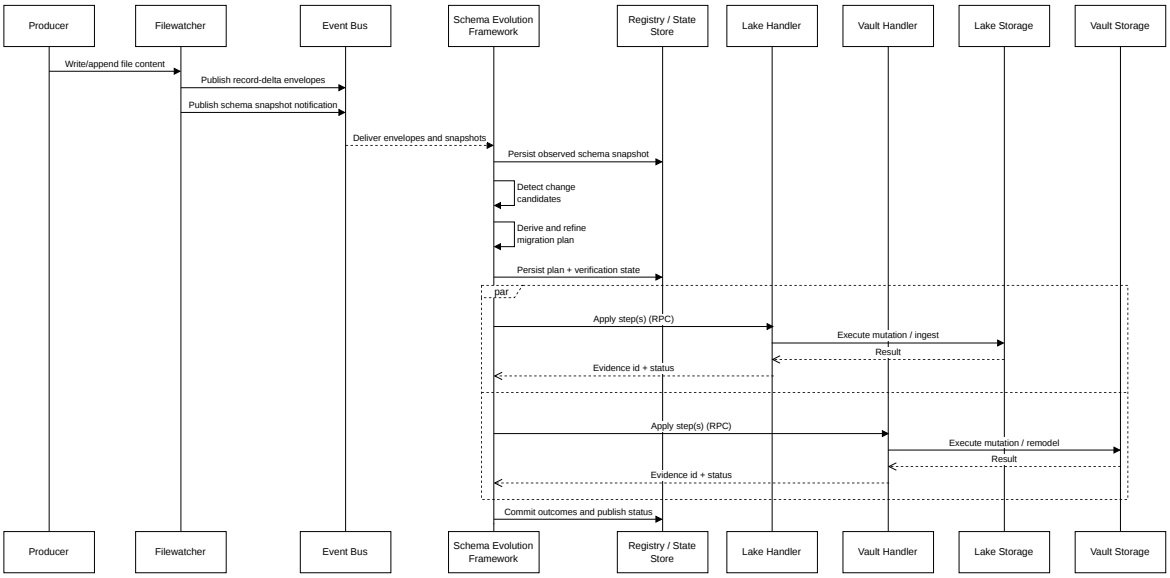


Figure 8.1.: E2E sequence view of the example, from file observation to evidence-based schema publication.

8.1. Scenario and dataset

We consider a single Comma-Separated Values (CSV) file `order.csv` in the monitored directory of the Filewatcher service. Following the dataset identity rule in Subsection 6.1.1, the logical dataset id is derived deterministically from the file name by stripping the extension, yielding the stable dataset id `order`. This identifier is used as the routing key across the data plane and control plane as described in Section 4.1 and Section 4.2.

Baseline file at t_0 At time t_0, the file contains the header

$$A_1 = [\texttt{id}, \texttt{customer\_id}, \texttt{amount}],$$

and a first set of data rows. We will use the following concrete content:

```
id,customer_id,amount
1001,C-17,12
1002,C-17,18
1003,C-08,7
```

Listing 8.1: `order.csv` at t_0 (header + three rows)

Timeline used in this chapter We use four time points ($t_0 < t_1 < t_2 < t_3$) to highlight both allowed and blocked drift. Policy behavior in this chapter follows the default SEF policy configuration (Subsection 5.2.3). Additive changes (`ADD_COLUMN`) are allowed and trigger vault structural obligations `NEW_HUB` and `NEW_LINK`. Type changes (`CHANGE_TYPE`) are blocked unless a sandbox or widening policy is configured. Destructive drops (`DROP_COLUMN`) are blocked under safe-by-default operation.

Table 8.1.: Timeline of schema drift and the SEF interpretation of the drift in this chapter.

Time	Producer change	Atomic change statement(s)	Expected governance outcome
t_0	Initial publish: `id, customer_id, amount`	none (baseline)	`allow`, baseline schema version recorded in the metadata repository
t_1	Add `discount` column	`ADD_COLUMN(discount)`	`allow`, `backward`, emits `NEW_HUB`, `NEW_LINK`, and lake/vault `ADD_COLUMN`
t_2	Widen `amount`: integer to float	`CHANGE_TYPE(amount)`	`block` by default, `allow` only under a widening or sandbox policy
t_3	Drop `customer_id` (breaking)	`DROP_COLUMN(customer_id)`	`block` under safe-by-default policy (Subsection 5.4.5)

8.2. Initial ingestion as a data-plane trace

A filesystem change triggers delta extraction in the Filewatcher. The schema notification is published before row-delta envelopes, giving the SEF control plane an early signal before new rows arrive at the lake handler. Since transport is bounded, rows are segmented into chunks so no single message exceeds the configured payload size, making at-least-once delivery tolerance visible from the first emission. Full offset-management and restart-safety details are in the online technical supplement [12].

Lake persistence and table isolation The DL handler persists the delta to the `order` table, enforcing table isolation so that drift in other concurrent datasets cannot pollute `order`'s schema.

After t_0, the lake holds three appended rows whose schema matches header A_1. The table may contain duplicates if the envelope is replayed under at-least-once delivery.

Vault handler: bronze persistence and catalog discovery The vault handler participates in the t_0 data plane trace in two ways. First, the vault handler data plane side persists the same CDC payload to the immutable Bronze staging layer, establishing the replayable enabler required for deterministic reprocessing. Second, the vault handler's startup sequence blocks on lake readiness and then reads the lake catalog to determine which Bronze tables already exist. At t_0 this catalog query returns the newly created `order` table with schema A_1.

No derived vault structures such as Hub, Link or Satellite exist at t_0. Their materialization is a control plane responsibility and requires explicit SEF operations. None have been issued yet because the t_0 notification carries no schema diff (Table 8.1). What the catalog discovery at t_0 establishes is the structural pre-condition for the t_1 plan: when the SEF later issues `NEW_HUB` and `NEW_LINK`, the vault handler can confirm that the Bronze table they target already exists rather than returning a transient readiness error.

8.3. Control-plane reaction as a schema-evolution trace

Alongside row deltas, the Filewatcher emits schema notifications as specified by the notification contract. The SEF consumes these `schema.notification` events and treats each one as a proposal consisting of the dataset descriptor, a header snapshot (ordered attribute descriptors plus optional primary key and fingerprint) and observation metadata.

8.3.1. Evolution step 1: adding `discount`

At time $t_1 > t_0$, the producer extends the file header to

$$A_2 = [\texttt{id}, \texttt{customer\_id}, \texttt{amount}, \texttt{discount}],$$

and starts writing rows that include `discount`. Concretely, a new row may look like:

```
1004,C-08,20,0.10
```

Listing 8.2: A new row appended at t_1 (introduces `discount`)

Atomization A new schema notification triggers the SEF to diff the previous accepted header snapshot against the newly observed snapshot and to reduce the difference into atomic change statements. Because the reference implementation diffs on `logical_type` rather than the CSV placeholder `physical_type`, the resulting change set contains a single additive statement: `ADD_COLUMN(discount)`.

Policy outcome Policy evaluation is deterministic and yields an outcome consisting of a decision, a compatibility label and optional obligations. Under the default rule set, purely additive changes are admitted as `allow` with `backward` compatibility and may carry structural obligations for the vault layer, such as `NEW_HUB` and `NEW_LINK` for establishing an initial vault structure.

Execution surface: plan operations to handler calls The executor applies the migration plan by dispatching each operation to the corresponding layer handler via the unified handler interface. Each operation carries `layer`, `kind`, `target`, `params`, `plan_id`, `correlation_id` and `idempotency_key`. Operations are dispatched in cross-layer dependency order: vault structural operations that have no lake counterpart such as `NEW_HUB` and `NEW_LINK` can be issued first, while a lake operation must be executed and verified before the corresponding vault operation for the same target.

Concrete vault-side effect (structural and additive) The ordering in the migration plan documented in the online technical supplement [12] matters. `NEW_HUB` creates an initial hub and satellite model for the `order` dataset, establishing a stable structural baseline keyed on `id`. `NEW_LINK` materializes a link structure capturing the relationship between `order` and the `customer_id` foreign-key candidate, as identified by the vault handler's conservative link-probe heuristic. Finally, `ADD_COLUMN` extends the satellite non-destructively by creating or extending a per-plan satellite version that includes the new descriptive attribute `discount`. This is precisely the non-destructive behavior established by the vault handler's evolution semantics: hubs and links capture identity relations, while descriptive drift is absorbed by satellites.

8.3.2. Evolution step 2: widening `amount` from integer to float

At time $t_2 > t_1$, the producer starts emitting fractional amounts, forcing value-based inference to change from `integer` to `float`:

```
1005,C-17,12.50,0.00
```

Listing 8.3: A new row appended at t_2 (fractional `amount`)

Atomization yields a single type-change statement: `CHANGE_TYPE(amount)`. In the reference implementation, the default policy blocks type changes and emits a failure event rather than a plan.

If the operator configures a widening-capable policy, `allow_change_type: widen_only` or a `sandbox` policy, then the change is allowed because the transition from `integer` to `float` is treated as widening. In that case, the planner emits explicit, layer-scoped `CHANGE_TYPE` operations with deterministic keys:

```
lake: 4991abd4f3bcde26e48d205d06a371b932e3320de2302d5faaca9b7ea90ef326
vault: 9e8605cf382dd10189b2f542e8fe5421ae41920eaa7841f13d6e5680e681b890
```

Listing 8.4: Operation keys for `CHANGE_TYPE(amount)` under a widening policy

The lake handler interprets the operation as a schema-alter request targeting the table and column with the desired logical type. If the durable schema already matches the desired type, the handler returns `ALREADY_APPLIED`, mirroring the idempotent convergence contract of the handler interface. On the vault side, `CHANGE_TYPE` is handled non-destructively as a transformation re-run request, consistent with the non-destructive evolution stance.

8.3.3. Evolution step 3: dropping `customer_id` (blocked drift)

At time $t_3 > t_2$, the producer removes `customer_id` from the header. The new header is

$$A_3 = [\texttt{id}, \texttt{amount}, \texttt{discount}].$$

Diffing yields a destructive change statement `DROP_COLUMN(customer_id)`. Under the safe-by-default policy, drops are not automated and the policy engine returns `block` with a `breaking` compatibility label.

Remediation note: One safe resolution is to rename the file to `order_v2.csv`, yielding a new dataset id `order_v2`. The breaking change is thereby isolated into a new lake table and new vault structures, preserving lineage continuity through the metadata repository.

8.4. Vault object state across evolution steps

The three evolution steps traced in the preceding sections each produce a structurally distinct vault posture. To make the cross-layer asymmetry operationally concrete and to substantiate the non-destructive evolution guarantees articulated in Subsection 7.2.2, this section presents the logical state of vault objects at each relevant time point as abstract structural snapshots. These snapshots deliberately mirror the header notation (A_1, A_2, A_3) already employed for the lake layer, and are rendered as logical attribute tables independent of any backend-specific storage format, Data Definition Language (DDL), or modeling toolchain. The structures shown reflect the output of deterministic heuristic inference applied by the vault handler, as bounded by the conservative link-probe semantics described in Subsection 8.3.1. They do not imply domain-semantic interpretation beyond what can be derived from structural evidence, consistent with the automation scope boundary established in Subsection 7.2.2.

8.4.1. State at t_0: bronze persistence only

At t_0, the SEF has received the initial schema notification but, because the notification carries no schema diff relative to a prior accepted version, no evolution plan is issued and no vault operations are dispatched. The vault handler's data-plane side persists the CDC payload to the immutable Bronze staging layer, establishing the replayable foundation required for deterministic reprocessing. No derived vault constructs (no Hub, no Link, no Satellite) exist at this point. The structural vault posture at t_0 is therefore:

Table 8.2.: Vault object state at t_0: Bronze persistence established, no derived structures exist.

Construct	Name	Columns	Notes
Bronze	`bronze_order`	`id`, `customer_id`, `amount`, `_ingestion_ts`	Immutable staging layer, replayable baseline
Hub	n/a	n/a	Not yet materialized
Link	n/a	n/a	Not yet materialized
Satellite	n/a	n/a	Not yet materialized

This absence is architecturally intentional. Vault structure materialization is a control plane responsibility, gated on explicit SEF operations. Vault structure materialization cannot occur as a side effect of data-plane ingestion. The catalog discovery performed by the vault handler at t_0, which confirms that the Bronze table `order` with schema A_1 is present, establishes the structural pre-condition that will be verified when vault operations are issued at t_1.

8.4.2. State at t_1: hub, link, and satellite materialized

The plan issued for t_1 carries three operations in dependency order: `NEW_HUB`, `NEW_LINK`, and `ADD_COLUMN`. Their execution produces the first derived vault structures for the `order` dataset. The structural vault posture after plan completion at t_1 is:

Table 8.3.: Vault object state at t_1: Hub, Link, and Satellite materialized following `NEW_HUB`, `NEW_LINK`, and `ADD_COLUMN` operations.

Construct	Name	Key columns	Descriptive / metadata columns
Bronze	`bronze_order`	n/a	`id`, `customer_id`, `amount`, `discount`, `_ingestion_ts`
Hub	`hub_order`	`order_hk` (derived from `id`)	`id`, `load_ts`, `record_source`
Link	`link_order_customer`	`link_hk`, `order_hk`, `customer_hk`	`load_ts`, `record_source`
Satellite	`sat_order_details`	`order_hk`, `load_ts`	`amount`, `discount`, `load_end_ts`, `hash_diff`

Several structural properties warrant explicit attention. First, `hub_order` stores the business key `id` together with a deterministically derived hash key and provenance metadata. No descriptive attributes are present in the Hub, consistent with the DV 2.0 modeling invariant that Hubs carry only identity (Subsection 7.2.2). Second, `link_order_customer` materializes the relationship between `order` and the `customer_id` foreign-key candidate, as identified by the vault handler's conservative link-probe heuristic. The Link carries two hash key references (`order_hk` and `customer_hk`) but stores no descriptive attributes directly, since relationship-level descriptive attributes would be captured in a Link Satellite if required. Third, `sat_order_details` records the descriptive attributes `amount` and `discount` in an insert-only historized structure. Rows appended prior to t_1, when `discount` was not yet part of the producer schema, will carry `discount = NULL`, preserving the historical record without rewriting earlier states. This is the non-destructive behavior that distinguishes vault-layer evolution from a destructive schema-alter operation: prior Satellite records remain structurally valid, and the addition of `discount` is absorbed by the existing Satellite without introducing a new Satellite version, because the change is purely additive.

8.4.3. State at t_2: lake altered, vault non-destructively updated

If a widening policy is configured, the `CHANGE_TYPE(amount)` operation results in a schema-alter on the lake table widening the `amount` column from `integer` to `float`. On the vault side, the operation is applied as a transformation re-run request rather than a structural mutation. The existing `sat_order_details` structure is preserved as a complete vault object, and the widened type is reflected in the loading logic for subsequent records. The logical structure of vault objects therefore remains identical to that shown in Table 8.3, with no construct added or removed. This is the correct behavior under the non-destructive evolution stance of Subsection 7.2.2: a type widening that preserves all previously representable values does not require a new Satellite version, since historical records carrying `integer`-typed `amount` values remain valid under the widened `float` representation.

Under the default policy, where `CHANGE_TYPE` is blocked, the vault posture is equally unchanged. The SEF publishes a failure outcome without dispatching any handler operations, and the vault objects established at t_1 are left in a consistent state.

8.4.4. State at t_3: block decision and vault-structural rationale

At t_3, the producer removes `customer_id` from the header, yielding the change statement `DROP_COLUMN(customer_id)`. The safe-by-default policy returns `block` with a `breaking` compatibility label. While the policy-layer rationale for this outcome is established in Subsection 8.3.3, the vault-structural dimension of the block decision warrants explicit examination, because the vault-structural dimension provides an architectural justification that is independent of and complementary to the policy rule.

As Table 8.3 illustrates, the attribute `customer_id` is not merely a flat column in the lake table at t_3: the `customer_id` attribute is the business key from which `customer_hk` in `link_order_customer` is deterministically derived. Permitting `DROP_COLUMN(customer_id)` would therefore not dissolve a flat column but would orphan a materialized Link construct whose identity rests on the continued derivability of `customer_hk` from the source schema. Future CDC events arriving without `customer_id` could no longer be joined to the existing link structure without semantic interpolation that the SEF explicitly excludes from the automation scope of the SEF. The vault posture at t_3 therefore remains identical to that established at t_1, since the block decision prevents any handler operations from being dispatched:

Table 8.4.: Vault object state at t_3: all structures preserved as a consequence of the `block` outcome. The Link construct's dependency on `customer_id` derivability provides an independent architectural justification for the governance decision, beyond the policy rule alone.

Construct	Name	Status after t_3
Bronze	`bronze_order`	Unchanged, `customer_id` still present
Hub	`hub_order`	Unchanged
Link	`link_order_customer`	Unchanged, structurally dependent on the continued derivability of `customer_hk` from `customer_id`
Satellite	`sat_order_details`	Unchanged

The remediation path noted in Subsection 8.3.3, renaming the file to `order_v2.csv` to establish a new dataset identifier, carries correspondingly richer implications when read against this structural backdrop. The new dataset would acquire separate Hub, Link, and Satellite structures keyed on the revised schema A_3, which no longer includes `customer_id` and would therefore not trigger a Link materialization for the customer relationship. Meanwhile, the lineage repository preserves the traceability relationship between `order` and `order_v2`, maintaining audit continuity without requiring in-place mutation of the historized `order` vault objects. This separation of lineage tracking from structural mutation is precisely the property that the metadata repository's immutable version semantics, described in Section 5.4.3, are designed to support.

8.5. Retries, evidence and what the verifier checks

The E2E architecture is designed for at-least-once delivery in both the data plane and the control plane. Consequently, the same schema notification and the same plan may be processed more than once due to event-bus replay, bounded retries in the executor or a process restart after partial progress. Correctness under retries relies on deterministic idempotency keys, handler-side convergence semantics and evidence-based verification. This matches the architectural failure model and convergence semantics in Section 4.5.

Concrete retry case Assume the executor applies the lake `ADD_COLUMN` operation from the plan documented in the online technical supplement [12] but crashes after the lake handler has committed the effect and before the corresponding checkpoint is durably persisted. On restart, the executor replays the same operation with the same `idempotency_key`. The lake handler detects that the requested post-state already holds and returns `ALREADY_APPLIED`. The verifier

then confirms the post-state through live introspection before proceeding with subsequent operations.

9. Evaluation

9.1. Objective and evaluation strategy

This chapter evaluates the integrated pipeline described in Chapters 5, 6, and 7 along two axes. First, the evaluation examines whether the implemented pipeline and the Schema Evolution Framework satisfy the functional and non-functional requirements elicited in Appendix B. Second, it evaluates governance behavior and fault-tolerance semantics under the SEF's completion model. Section 9.2 presents the requirements-oriented evidence. Section 9.3 reports the performance evaluation.

The evaluation is anchored to a convergence-based correctness criterion: the test suite does not merely assert that events were emitted, but validates converged state at authoritative persistence surfaces and control plane outcome topics. This criterion follows directly from the architecture: under at-least-once delivery, correctness is expressed as idempotent convergence to a verified published outcome, not as intermediate progress.

9.2. Requirements validation

This section validates the requirements stated in Appendix B against two complementary evidence sources. First, an E2E test is run with the configuration `e2e, no-perf, no-bench`, providing black-box evidence for externally observable properties of the holistic pipeline. The run completed successfully across the configured scenarios, thereby establishing a baseline confirming that the integrated system executes deterministically. Second, requirements that are not fully testable via black-box assertions (either because they depend on operational configuration choices, non-functional characteristics or internal architectural constraints) are validated by structured inspection of the implementation.

The requirements coverage report [28] classifies automated E2E evidence into three categories. These categories are evidence statuses rather than standalone requirement-satisfaction judgments:

- PASS: directly asserted by the suite
- PARTIAL: implemented or architecturally supported, but only partially assertable by the suite
- GAP: not asserted by the suite and therefore requiring manual inspection where the requirement is relevant to the thesis scope

Across the suite run, the report indicates broad functional E2E coverage. The ingestion path and schema evolution control loop are largely asserted E2E, while several operational, packaging and security requirements listed in section B.3 remain outside the scope of automated E2E checks.

9.2.1. Functional requirements

The Filewatcher requirements *FR-1* and *FR-2* (continuous monitoring of a directory and detection of file creation or modification) are validated by the E2E tests driving controlled file events and verifying that the downstream system state changes accordingly. The coverage summary report in Table B.3 classifies both requirements as PASS.

FR-3 and FR-4 (schema evolution detection and event emission into the streaming backbone) are likewise classified as PASS. The control and data plane separation substantiates that schema notifications are treated as first-class control plane inputs and are intentionally decoupled from row deltas. This separation is important for validation because it demonstrates that schema detection is not an incidental side effect of payload processing but a distinct message family with explicit semantics.

FR-5 (use of a streaming service) and FR-6 (consumption of events and conversion into a data frame-like processing representation) are validated in the same E2E manner. The pipeline is explicitly constructed around an event bus based on Apache Kafka, with consumers materializing batches into the backend persistence. The Data Lake Handler further specifies that streaming mode consumes micro-batches continuously and routes records deterministically into table-scoped structures.

Two requirements are classified as PARTIAL because they describe multi-modal behavior that cannot be completely asserted in a single E2E configuration without substantially increasing the suite complexity. FR-7 (storage in real-time and batch mode) and FR-10 (storage into a backend based on configuration) fall into this category. The E2E tests validate the real-time, streaming-first path, but do not exhaustively execute the scheduled bulk-drain path or cover all mode-dependent edge cases. This reflects a testing boundary rather than an implementation

absence. The Data Lake Handler explicitly supports both streaming and bulk modes. The remaining aspects are documented in the online technical supplement [12].

The persistence layer requirements FR-8 and FR-9 are classified as PASS. The ingestion semantics specify deterministic routing from ingestion metadata to a logical table identifier, providing a direct conceptual bridge between the requirement to use the CSV filename as table name and the implemented routing function.

The DV automation requirements FR-11, FR-12 and FR-13 are flagged as PASS. The DV orchestrator governs ingestion using CDC ingestion and streaming consumption as well as automated model materialization and metadata capture under deterministic lifecycle control. The E2E assertions validate not only that data reaches the Raw Vault objects, but that the pipeline converges reproducibly under schema drift episodes in the tested cases.

The schema evolution integration requirements FR-14 and FR-15 are also PASS. This is consistent with the definition of the SEF as a control plane authority consuming schema notifications, performing diffing and lineage-aware impact analysis, evaluating governance policies deterministically and producing an ordered operation plan with idempotency keys and verification evidence prior to publication.

FR-16 (listening to a dedicated streaming topic for schema change events) is classified as PARTIAL. The underlying design is unambiguous, as the architecture defines separate message families for row deltas and schema notifications and explicitly specifies that correctness must not depend on cross-topic ordering. The partial flag is primarily methodological: a black-box test can assert that events exist and are consumed, but the black-box test does not robustly prove topic-level isolation properties under adversarial conditions or enumerate all topic configurations.

Containerization, packaging and operational requirements

The remaining functional requirements FR-17 to FR-21 are operational, packaging and monitoring concerns whose E2E evidence status is limited. FR-17, FR-18, FR-20 and FR-21 are classified as GAP in the automated E2E report, while FR-19 is classified as PARTIAL. They therefore require manual implementation and deployment analysis in addition to the E2E evidence.

FR-17 requires that each major component is deployed in a dedicated Docker container. The deployment and packaging view documented in the online technical supplement [12] describes a containerized topology with four application containers (Filewatcher, Data Lake Handler, Data Vault Handler and SEF Core) plus infrastructure containers. The DV orchestrator is modeled as a single authoritative instance that co-governs CDC production, streaming consumption and

model materialization as one reproducible system of record. Strict one-role-per-container decomposition is intentionally relaxed to preserve determinism and avoid split-brain orchestration.

FR-18 mandates that each major component can be packaged as a Python package. Static inspection of the service artifacts shows that the components are structured as importable Python modules executed directly as applications, but they do not expose standard packaging metadata such as a wheel build. Strictly interpreted, this requirement is only partially met: modularization exists but distributable packaging in the installable sense is not fully realized in the current form.

FR-19 requires orchestration via Docker Compose. The implementation and the evaluation harness rely on Docker Compose to coordinate application containers and infrastructure dependencies. This requirement is validated as met at the level of the delivered runtime topology and evaluation environment.

FR-20 and FR-21 concern logging and monitoring. Static inspection of the source code indicates systematic use of multi-level logging calls (DEBUG, INFO, WARNING, ERROR, CRITICAL). The thesis provides additional support for context details in diagnostics by emphasizing E2E traceability identifiers (dataset identifiers, correlation and plan identifiers, idempotency keys and evidence IDs) that are propagated across asynchronous boundaries.

9.2.2. Non-functional requirements

The performance and scalability requirements NFR-1 to NFR-3 are listed as GAP in the automated E2E coverage report, as the E2E configuration is set to *no perf and no bench*. The implementation nonetheless directly supports the qualitative intent of these requirements. The Data Lake Handler's streaming mode is explicitly designed for low-latency micro-batch persistence with a backlog-aware loop and bounded recovery behavior. Whether bulk processing is efficient in the quantitative sense is an empirical question addressed in the online technical supplement [12].

The reliability requirements NFR-4 and NFR-5 are classified as PARTIAL. The implementation is explicitly designed for restart-safe convergence through persisted state and checkpoints, at-least-once tolerance at the ingestion boundary and deterministic restart behavior in the orchestrators. The requirement's wording also implies operational policies such as automatic restart of failed services that depend on deployment configuration. Roll-forward completion is preferred over rollback because rollback can introduce inconsistency. Rollback procedures are intentionally not the primary recovery mechanism. The requirement is therefore satisfied in the integrity dimension while the rollback dimension is met only insofar as the system compensates via re-planning and safe completion semantics.

The security requirements NFR-6 and NFR-7 are both flagged as GAP in the automated E2E coverage report. Basic access control is present where the chosen infrastructure supports access control and configuration parameters are externalized for portable deployment. However, secured communication between components is only partially realized, as the evaluation topology prioritizes local reproducibility over hardened production security. Under strict interpretation, NFR-7 is not fully met without explicit TLS, SASL or equivalent protocols for inter-service communication.

The maintainability and extensibility requirements are weakly represented in automated E2E assertions but are well aligned with the implementation when inspected architecturally. NFR-8 (modular organization) is supported through the unified handler interface and layer-specific responsibilities. NFR-9 (detailed logging and error reporting) follows from the systematic adoption of structured traceability identifiers across asynchronous boundaries. NFR-10 (future enhancements) is directly anticipated by the architecture, as the policy evaluation is parameterized via an external policy file and the separation of responsibilities provides clear extension points for additional pipelines and sources.

Portability requirements NFR-11 and NFR-12 are validated primarily by the deployment model and configuration surfaces. The system is entirely containerized with explicit infrastructure dependencies and persistent state for restart-safe convergence. Configuration is explicitly externalized, allowing the same artifact to be deployed across development and testing environments with behavior controlled through configuration rather than recompilation.

9.2.3. Summary and Interpretation of Requirements Validation Outcome

The E2E suite provides strong evidence for the core functional path. Requirements classified as PARTIAL correspond to implemented capabilities whose remaining dimensions are difficult to validate through black-box E2E assertions, most notably multi-mode operation and topic-level isolation guarantees beyond mere existence. Requirements classified as GAP are mainly non-functional or deployment-oriented concerns that are assessed through implementation and deployment analysis. In areas such as security hardening and distributable Python packaging, they mark points where the implementation deliberately favors research reproducibility over production completeness.

The requirements-oriented evaluation in Section 9.2 provides the primary E2E evidence that the implemented artifact satisfies the core functional path within the stated thesis scope. The explicit PASS/PARTIAL/GAP distinction separates automated E2E evidence from additional implementation and deployment analysis, preventing the coverage table from being read as a single unqualified satisfaction score.

Since the problem is formulated as a cross-layer coordination problem under replay and partial failure assumptions, the relevant success criterion is not the initiation of local operations but the achievement of a verified and publishable completion state. The evaluation procedure in Chapter 9 is consistent with this premise and supports the internal coherence of the thesis' methodology and artifact validation.

9.3. Performance evaluation: governance behavior and fault tolerance

The requirements validation establishes correctness under drift scenarios. This section extends that evidence by evaluating two behavioral properties of the SEF that are directly relevant to RQ3. The first is the operational distinction between admitted and blocked changes under policy evaluation. The second is the correctness and latency effect of retries under at-least-once delivery. Both subsections reuse the same E2E harness and preserve the same convergence-based correctness boundary. Metrics are collected only for runs that satisfy the same convergence conditions used in the functional test suite.

Operational benchmark data, including E2E throughput, evolution latency, scaling under evolution intensity, phase breakdown and burst workload sensitivity, are provided in the online technical supplement [12]. Those results quantify `NFR-1` to `NFR-3` and characterize backend-specific convergence behavior but are not required to answer RQ1 to RQ3.

9.3.1. Policy evaluation correctness and time-to-outcome

The E2E validation already establishes that the control plane publishes policy outcomes consistently with the intended governance semantics. In the static benchmark suite, policy evaluation is therefore primarily used to quantify the time dimension of governance: how quickly an authoritative outcome becomes available to downstream consumers after a schema change notification, and how this delay differs between immediately blocked evolutions and evolutions that must pass through backend actuation and verification.

Across both backends, all expected `evolved` cases are observed as `schema.evolved` (90/90) and all expected `failed` cases are observed as `schema.evolution.failed` (90/90). The policy decision is faithfully reflected by the outcome publication, allowing the performance discussion below to focus on time-to-outcome rather than outcome correctness.

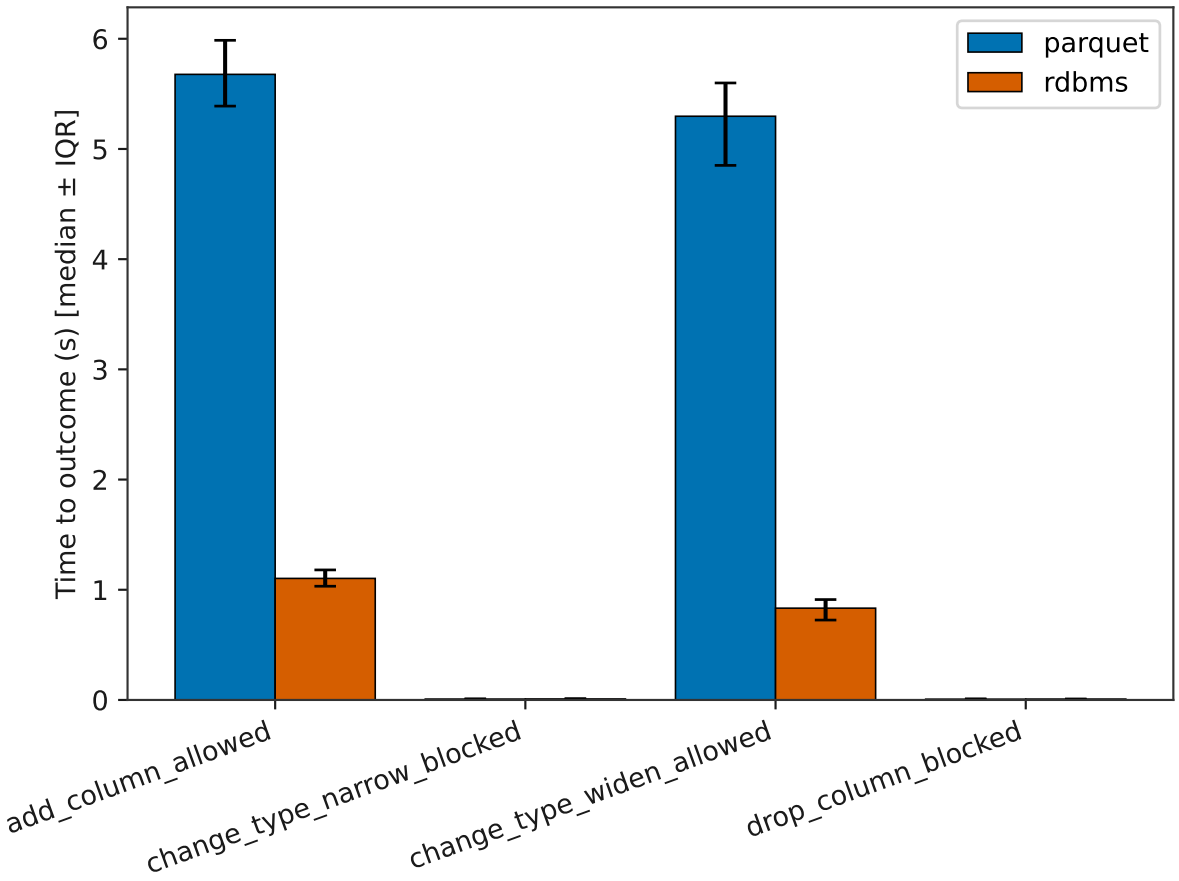


Figure 9.1.: Policy evaluation time-to-outcome by scenario and backend.

The time until the authoritative outcome is published bounds the time-to-convergence perceived by consumers, because downstream components cannot safely assume a converged schema before the corresponding outcome event is available. Figure 9.1 reports the time-to-outcome distributions for `add_column_allowed`, `change_type_narrow_blocked`, `change_type_widen_allowed` and `drop_column_blocked`, using the median and Interquartile Range (IQR) across 45 runs per scenario and backend.

For *blocked* evolutions, outcome publication is effectively immediate and backend-insensitive. For parquet, `drop_column_blocked` completes with a median of 0.007 s (IQR 0.006-0.010 s) and `change_type_narrow_blocked` with a median of 0.008 s. For the relational backend, both blocked scenarios are in the same millisecond regime. This matches the intended control-flow: once a change is denied by the policy engine, the system can publish a failure outcome without performing backend actuation or waiting for evidence from verification.

For *allowed* evolutions, time-to-outcome increases by orders of magnitude, because the outcome must be gated on plan creation, backend actuation, persistence of execution evidence and verification. For parquet, `add_column_allowed` completes with a median time-to-outcome of 5.677 s (IQR 5.389-5.986 s) and `change_type_widen_allowed` with 5.297 s. For the relational backend, the same allowed scenarios complete substantially faster: `add_column_allowed` has a median of 1.103 s and `change_type_widen_allowed` 0.833 s. Expressed as ratios of medians, parquet is approximately 5.15$\times$ slower than the relational backend for `add_column_allowed` and 6.36$\times$ slower for `change_type_widen_allowed`, while blocked outcomes are effectively equivalent in absolute terms at roughly 7 to 10 ms.

Time-to-outcome is the operational cost of evidence-based completion. Under blocked changes, governance behaves like a fast classifier and terminates early. Under allowed changes, governance acts as a correctness gate that defers completion until backend-specific execution and verification are complete. Even when governance logic is backend-neutral, the time until an admissible evolution is confirmed and published remains constrained by the backend's ability to execute, persist and verify the change.

9.3.2. Fault tolerance and retry-cost effects on completion latency

The static run includes a fault-injection evaluation in which transient failures are introduced into the execution path and the system is allowed to recover under the configured at-least-once retry discipline. Across both backends and all tested fault scenarios, the SEF converges to an authoritative outcome publication consistent with the fault-free result. No run produces an incorrect outcome or misses an outcome under transient fault injection. This result shows

that deterministic idempotency keys, handler-side convergence semantics, and evidence-based verification are sufficient to preserve correctness under the modeled failure class.

The latency cost of retry-induced convergence delays is backend-dependent and exhibits heavy-tailed behavior for the parquet backend. The quantitative characterization of retry-induced extra latency, including per-backend distributions, p90 values and baseline-versus-fault scatter, is documented in the online technical supplement [12]. Since this thesis defines correctness at the point of verified outcome publication, retry-induced tail amplification is operationally relevant: retry-induced tail amplification directly bounds the time until the system can safely assert convergence at the control plane boundary and is discussed further in Section 9.3.3.

9.3.3. Interpretation of the performance evaluation

Section 9.3 evaluates the two behavioral properties of the SEF directly relevant to RQ3: policy-based governance and retry-cost under at-least-once delivery. Operational benchmark findings covering throughput, latency, scaling and phase breakdown are documented in the online technical supplement [12].

Policy evaluation correctness and time-to-outcome semantics

The policy evaluation results show that the policy layer is not merely a logical admission filter but an essential part of the operational completion semantics. The time-to-outcome distributions demonstrate a clear asymmetry between blocked and allowed changes: blocked changes terminate early and publish failure outcomes quickly, whereas allowed changes traverse planning, actuation, evidence persistence and verification before an authoritative success outcome is emitted.

This asymmetry directly supports the thesis' governance model. In an evidence-gated evolution pipeline, governance and runtime behavior must be interpreted together because both determine when the system may legitimately claim convergence.

Fault tolerance and retry-cost effects under at-least-once delivery

The retry-cost evaluation is especially significant because asynchronous processing with at-least-once delivery, duplicates, and retries is part of the intended execution model rather than an exceptional case. The results show that correctness is preserved under transient faults and replay, but the latency effects depend on the backend and often appear as tail amplification rather

than uniform slowdown. Because correctness is defined at the point of authoritative outcome publication, retry-induced tail growth delays the moment at which downstream components may safely rely on the convergence signal. Retry behavior is therefore part of the E2E completion model at the control plane boundary.

9.4. Discussion

This section interprets the evaluation results reported in Chapter 9 in relation to the design problem stated in Chapter 1, the research questions RQ1 to RQ3, and the Design Science Methodology positioning of this thesis. The discussion now focuses on implications, limitations, and the explicit return to the research questions; the two detailed interpretation subsections are placed directly after the evidence they interpret.

9.4.1. Implications

The evaluation results support several implications for the design and operation of systems built around the SEF.

The authoritative contract for downstream integration is the published completion outcome, not intermediate execution traces or partial handler states. Downstream automation must be anchored to durable evidence and outcome publication rather than to provisional progress indication.

Service level expectations for schema evolution must be formulated in terms of time-to-outcome distributions including tail behavior under retry, rather than only average latency under fault-free conditions. The fault tolerance evaluation shows that upper-tail behavior can dominate operational experience even when median values remain comparatively stable.

Backend selection affects not only the time required to complete admissible changes but also the tail behavior under transient faults. Backend choice must therefore be treated as an architectural decision with direct consequences for operational convergence guarantees.

9.4.2. Limitations

The measured performance values are specific to the implemented artifact, the selected topology and the execution environment, and should be read as feasibility evidence for a specific, reproducible configuration, not as general performance benchmarks. The evaluated topology

prioritizes local reproducibility over production hardening: inter-service communication uses plaintext transport and the broker is configured without TLS or SASL, as acknowledged in the requirements validation for NFR-6 and NFR-7. The operational conclusions of this thesis are therefore bounded to the evaluated configuration and do not extend to production-security requirements.

The thesis explicitly targets structural schema evolution and excludes semantic evolution. The SEF is also designed and evaluated within a deliberately fixed layered DL and DV architecture, which improves internal consistency and traceability of design and evaluation but leaves open the degree to which the same control plane principles transfer to other warehouse or lakehouse settings. Finally, the Raw Vault model inference relies on deterministic naming- and suffix-based heuristics with conservative fallback strategies, prioritizing information preservation over semantic interpretation. The present implementation therefore automates structural coordination rather than domain-semantic modeling decisions.

9.4.3. Discussion with respect to the research questions

This section maps the evaluation evidence back to RQ1 to RQ3.

RQ1: detection and representation The E2E validation demonstrates that drift observations can be transformed into control plane inputs and processed deterministically through policy, planning and execution pathways. Determinism follows from the classification algorithm, which reduces observed header differences to a finite set of atomic change statements (`ADD_COLUMN`, `DROP_COLUMN`, `CHANGE_TYPE`) over a normalized type alias vocabulary, ensuring that equal inputs always produce equal change representations. RQ1 is therefore addressed for structural drift classes derivable from technical evidence, consistent with the thesis scope. Semantic drift remains intentionally out of scope.

RQ2: cross-layer propagation The evaluation supports the conclusion that the SEF, together with the implemented handlers, provides a workable mechanism for cross-layer propagation under the assumed execution semantics. The key evidence is the coordinated treatment of accepted changes from admission through planning, actuation, verification, and final outcome publication. This reduces the need for isolated per-layer migration work, even though the thesis does not include a quantitative before-and-after measurement of manual effort. RQ2 can therefore be answered positively with respect to coordination and correctness, while runtime behavior remains dependent on backend semantics.

RQ3: reliability, governance and quality assurance The evaluation provides the strongest empirical support for the thesis answer to RQ3. In the present setting, safe automation requires at least three conditions: explicit policy-based admission, replay-safe and idempotent actuation, and evidence-based verification before authoritative outcome publication. The policy evaluation results demonstrate the operational difference between blocked and allowed paths, while the retry-cost findings show why replay safety and convergence timing must be considered together under at-least-once assumptions. RQ3 is therefore not only conceptually addressed but also empirically supported. Policy and verification are core elements of the completion model required for safe automation in asynchronous and failure-prone environments.

In summary, the evaluation results support the central claim of this thesis: the Schema Evolution Framework is a viable control plane mechanism for replay-safe, evidence-gated schema evolution coordination in a layered DL and DV architecture under asynchronous execution and at-least-once delivery. These findings directly motivate the future work directions presented in Chapter 10.3.

9.5. Summary of the evaluation outcomes

The evaluation examined the artifact along two complementary axes: requirements-oriented validation and performance evaluation. The `PASS/PARTIAL/GAP` evidence classification makes the automated testing boundary explicit and shows that the core contribution lies in controlled and verifiable schema evolution coordination, while some non-functional and deployment-hardening aspects remain intentionally outside the tested E2E scope.

The functional evaluation confirms E2E convergence. Across the tested scenarios, the integrated pipeline executes deterministically, the policy matrix produces the intended outcomes on both backends, and transient fault injection does not lead to incorrect or missing authoritative outcomes.

The governance evaluation shows a clear operational asymmetry between blocked and allowed paths. Blocked changes resolve quickly because the system can publish a failure outcome immediately after policy evaluation. Allowed changes require planning, backend actuation, evidence persistence, and verification before a success outcome can be published. Backend choice therefore affects not only execution cost but also the time-to-outcome perceived by downstream consumers. In the tested allowed scenarios, the parquet backend shows median latencies that are roughly five to six times higher than those of the relational backend. Retry-induced latency growth is also backend-dependent and operationally relevant because retry-induced latency growth delays the point at which the system can safely assert convergence.

Operational benchmark findings covering throughput, evolution latency, scaling, and burst-workload sensitivity are documented in the online technical supplement [12].

10. Conclusion

This thesis treated schema evolution as a continuous operational concern in event-driven data platforms that combine Data Lake (DL) persistence with a Data Vault 2.0 (DV) integration layer. Structural schema drift at the ingestion boundary can propagate across ingestion, persistence, change derivation, vault modeling, and downstream consumption contracts. Structural schema drift therefore creates a coordination problem that cannot be resolved through isolated per-layer migrations. Within Wieringa's Design Science Methodology, the thesis addressed how schema evolution can be automated in combined DL and DV environments while preserving data quality and ensuring reliable completion under asynchronous execution.

10.1. Summary

The primary contribution is the *Schema Evolution Framework* (SEF), introduced as an explicit control plane mechanism for coordination and verification of *structural* schema evolution across a combined DL and DV setting. The SEF is realized in a prototype integrating an ingestion boundary observer, layer handlers for Data Lake and Data Vault 2.0 processing and an event bus for message orchestration.

The defining architectural decision is the strict separation of schema observation from schema completion. A detected schema drift is treated as a proposal that must pass admission control, explicit planning, idempotent actuation and evidence-based verification before the system publishes an authoritative completion signal. The artifact scope was deliberately limited to *structural* schema evolution in a combined DL and DV architecture under an event-driven execution model with duplicates, retries and replay-safe convergence requirements, preserving a technically well-defined and measurable problem.

The contribution of this thesis can be summarized at four levels. At the conceptual level, the thesis frames schema evolution in combined DL and DV environments as a cross-layer coordination problem under asynchronous execution and replay assumptions rather than as a local schema migration problem. At the design level, the thesis introduces the SEF as an explicit control plane mechanism that separates schema observation from schema completion and organizes

admission control, planning, actuation, and evidence-gated verification. At the artifact level, the thesis delivers a prototype that integrates an ingestion-boundary observer, layer handlers, and message-based orchestration across the targeted combined DL and DV pipeline. Finally, the artifact described and implemented in this thesis is systematically evaluated following a convergence-oriented evaluation boundary that combines E2E requirements validation with governance behavior and fault-tolerance analysis.

The thesis also retains clear boundaries. Semantic evolution remains out of scope, and the current automation surface is intentionally conservative. The implementation is evaluated in one fixed combined DL and DV setting. These boundaries do not weaken the contribution. They define the conditions under which the artifact has been specified and evaluated, and they provide a clear basis for the future work proposed in Section 10.3. Within those boundaries, the SEF remains a viable and tested mechanism for replay-safe schema evolution coordination under asynchronous and failure-prone execution assumptions.

10.2. Research questions revisited

All three research questions can be answered within the defined scope. The following points summarize the final position of the thesis.

RQ1: Detection and representation. *How can structural schema drift at the ingestion boundary be detected and represented in a normalized form that supports deterministic downstream processing?*
Structural schema drift at the ingestion boundary is derived from technical evidence and represented in a normalized control plane form, which enables deterministic downstream processing. The answer is therefore limited to structural drift classes that can be inferred from observable technical metadata. Semantic drift remains outside the thesis scope.

RQ2: Cross-layer propagation. *How can schema evolution be propagated consistently across a layered DL and DV architecture while minimizing manual refactoring effort?*
The framework propagates accepted schema changes consistently across the layered DL and DV architecture by governing the evolution process from observation through verified completion. The contribution therefore concerns coordination and correctness of propagation. A quantitative before-and-after comparison of manual refactoring effort remains outside the current evaluation scope.

RQ3: Reliability, governance and quality assurance. *Which policy and verification mechanisms are required to automate schema evolution safely under at-least-once delivery and partial failure, while preserving data quality and maintaining stable consumption contracts?*
Safe automation under at-least-once delivery and partial failure requires the combination of

explicit policy-based admission, replay-safe actuation and evidence-based verification before authoritative outcome publication. The evaluation provides support for this combination and demonstrates that policy and verification are not supplementary features but core components of the completion semantics.

10.3. Future work

This section outlines the most relevant extensions that follow from the evaluation and the scope boundaries defined in this thesis. The artifact demonstrates E2E convergence for the targeted setting, but several boundaries were kept deliberately narrow so that the automation problem remained technically well defined and empirically testable. In particular, the current implementation relies on deterministic naming and suffix heuristics for Raw Vault model inference, excludes semantic evolution, focuses on the Raw Vault actuation boundary rather than Business Vault or presentation-layer artifacts, and evaluates the SEF within one fixed architectural setting. The following points therefore develop four natural next steps: stronger hub identification, an operator-facing User Interface (UI), deeper support for the broader Data Vault stack, and a more generic framework realization.

Better hub identification. Hub modeling requires explicit selection and standardization of business keys that are stable, meaningful in the business domain and consistently derivable across sources. The current approach extracts and ranks key candidates using deterministic heuristics and applies a conservative fallback strategy when confidence is low. Future work can strengthen this step through additional evidence channels.

(a) *Dataset crawling and stability detection.* Extending the initial and periodic crawl through the data to compute data-driven stability signals for candidate key columns, including uniqueness, null ratios, collision rates and persistence across versions, would move hub selection from purely syntactic inference to an evidence-based decision with explicit scoring and auditability.

(b) *GenAI integration with existing datasets and a domain knowledge database.* Although semantic evolution is out of scope for this thesis, a controlled advisory layer can leverage GenAI together with curated domain knowledge, such as a business glossary, reference entities or canonical identifiers, to propose or rank hub keys, detect synonymy and recommend standardization. To preserve the SEF premise, such outputs should remain recommendations requiring explicit governance and verification rather than implicit automated rewrites.

Operator-facing UI for the complete pipeline. The SEF emphasizes traceability, governance, and verification through handler evidence and E2E correlation identifiers. In the current artifact, these properties are mainly exposed through logs, configuration files, and evidence interfaces rather than through a unified operator view. A dedicated UI should therefore cover the complete pipeline. The operator-facing UI should support validated configuration management across components, structured log and trace exploration by dataset, correlation identifier, and plan identifier, and a readable view of process state. The operator-facing UI should also expose the current posture of each dataset, the last accepted version, pending plans, and the evidence artifacts that support verification outcomes.

Further Data Vault enablement. While the implementation applies Raw Vault evolution as an explicit and governed actuation process under SEF control, DV 2.0 architectures commonly include additional layers and artifacts central to enterprise delivery. Two extensions would be particularly relevant.

(a) *Business Vault support.* The Business Vault introduces reusable, governed derivations on top of the Raw Vault, such as reconciled attributes, enterprise-wide calculations or navigation aids such as bridge tables, while preserving the Raw Vault as an auditable record of observed data. Future work should extend the handler and plan vocabulary so that the SEF can manage evolution not only of Raw Vault structures but also of Business Vault derivations as first-class, versioned artifacts with explicit verification.

(b) *Transformation and presentation layers.* In a DV architecture, Information marts and consumption-facing views are derived from the vault, frequently requiring historization strategies and domain-oriented semantics such as dimensions, facts, point-in-time (PIT) tables or bridge tables. A natural next step is to connect the impact analysis of the SEF and the verification to downstream transformation graphs, so that schema evolution plans can propagate through to presentation layer artifacts and validate that delivery contracts remain stable or are evolved in a controlled and observable manner.

Generic, architecture-agnostic framework implementation. The thesis deliberately fixes the architectural setting to a layered DL and DV platform so that the automation problem remains precise and measurable. Even so, the core control plane principles of the SEF are not tied to this single setting. A useful next step is therefore to generalize the implementation through an architecture-neutral abstraction layer for change detection and normalization, operation planning, and handler capabilities. In practical terms, the current layer handler could evolve into a set of pluggable adapters for different DWH patterns, such as lakehouse-only persistence with

contract views, ELT into dimensional models, or alternative integrated cores. The SEF would remain the stable control plane responsible for admission, orchestration, and verification.

Bibliography

[1] Zouhaier Brahmia, Fabio Grandi, and Barbara Oliboni. "A Literature Review on Schema Evolution in Databases". In: *Computing Open* 2.1 (2024), pp. 1–37. DOI: 10.1142/S2972370124300012 (cit. on pp. 1, 2, 7, 9).

[2] Alberto Hernández Chillón et al. "A Generic Schema Evolution Approach for NoSQL and Relational Databases". In: *IEEE Transactions on Knowledge and Data Engineering* 36.7 (2024), pp. 2774–2789. DOI: 10.1109/TKDE.2024.3362273 (cit. on pp. 1, 2, 5, 7, 8, 9).

[3] Irena Holubová. "Evolution Management in Multi-Model Databases". In: *Data & Knowledge Engineering* 136 (2021), p. 101932. DOI: 10.1016/j.datak.2021.101932 (cit. on pp. 1, 2, 9).

[4] Dan Linstedt and Michael Olschimke. *Building a Scalable Data Warehouse with Data Vault 2.0*. Morgan Kaufmann (Elsevier), 2015. ISBN: 978-0-12-802510-9 (cit. on pp. 1, 2, 14, 15, 16, 87, 88, 89, 90).

[5] Peter Gluchowski. "Data Vault as a Modeling Concept for the Data Warehouse". In: *Modern Data Warehousing*. Springer, 2022. DOI: 10.1007/978-3-030-84655-8_17 (cit. on pp. 1, 15, 87, 88, 89, 90).

[6] Marios Fragkoulis, Asterios Katsifodimos, et al. "A Survey on the Evolution of Stream Processing Systems". In: *The VLDB Journal* (2024). DOI: 10.1007/s00778-023-00819-8 (cit. on pp. 1, 2, 10, 87).

[7] John F. Roddick. "A Survey of Schema Versioning Issues for Database Systems". In: *Information and Software Technology* 37.7 (1995), pp. 383–393. DOI: 10.1016/0950-5849(95)91494-K (cit. on pp. 2, 5, 7).

[8] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. "Provenance in Databases". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. 2007, pp. 1171–1173. DOI: 10.1145/1247480.1247646 (cit. on pp. 4, 7, 88).

[9] Borja Pérez, Mariano Rodríguez-Muro, and José Manuel Gómez-Pérez. "PROV-IDEA: Supporting Interoperable Schema and Data Provenance within Database Evolution". In: *ACM Transactions on Software Engineering and Methodology* (2025). DOI: 10.1145/3697008 (cit. on pp. 4, 7, 8, 9, 10).

[10] Hong-Linh Truong and Ngoc Nhu Trang Nguyen. "TENSAI – Practical and Responsible Observability for Data Quality-Aware Large-Scale Analytics". In: *Journal of Data and Information Quality* 16.4 (2024), Article 25. DOI: 10.1145/3708014 (cit. on pp. 4, 7, 10).

[11] Roel Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014 (cit. on pp. 4, 5, 7).

[12] Alexander Mayr. *Schema Evolution Framework – Additional Documentation and Support Material*. Accessed: 2026-03-01. 2026. DOI: 10.5281/zenodo.18962188. URL: https://doi.org/10.5281/zenodo.18962188 (cit. on pp. 4, 6, 18, 23, 38, 39, 43, 46, 49, 50, 51, 54, 56, 58, 63, 67, 68, 70, 73, 77).

[13] Andrea Hillenbrand et al. "Self-Adapting Data Migration in the Context of Schema Evolution in NoSQL Databases". In: *Distributed and Parallel Databases* 40.1 (2022), pp. 5–25. DOI: 10.1007/s10619-021-07334-1 (cit. on pp. 7, 8).

[14] André Conrad et al. "EvoBench: Benchmarking Schema Evolution in NoSQL". In: *Performance Evaluation and Benchmarking*. Vol. 13169. Lecture Notes in Computer Science. Cham: Springer, 2022, pp. 45–63. DOI: 10.1007/978-3-030-94437-7_3 (cit. on pp. 7, 10).

[15] Panos Vassiliadis and Alexandros Karakasidis. "Time-Related Patterns of Schema Evolution". In: *Proceedings of the 28th International Conference on Extending Database Technology (EDBT 2025)*. 2025, pp. 310–323. DOI: 10.48786/edbt.2025.25 (cit. on pp. 7, 10).

[16] Rihan Hai et al. "Data Lakes: A Survey of Functions and Systems". In: *IEEE Transactions on Knowledge and Data Engineering* 35.12 (2023), pp. 12571–12590. DOI: 10.1109/TKDE.2023.3270101 (cit. on pp. 2, 7, 10, 14, 88, 90, 91).

[17] Muriki G Yamanaka et al. "Statistical validation of column matching in the database schema evolution of the brazilian public school census". In: *arXiv preprint arXiv:2407.09885* (2024) (cit. on pp. 8, 9).

[18] 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. DOI: 10.14778/3415478.3415560 (cit. on pp. 10, 14, 88, 89, 90, 91).

[19] William H. Inmon. *Building the Data Warehouse*. USA: John Wiley & Sons, Inc., 1992. ISBN: 0471569607 (cit. on p. 12).

[20] Alejandro Vaisman and Esteban Zimányi. *New Data Warehouse Technologies*. Springer Berlin Heidelberg, 2014. ISBN: 9783642546556. DOI: 10.1007/978-3-642-54655-6_13. URL: http://dx.doi.org/10.1007/978-3-642-54655-6_13 (cit. on p. 12).

[21] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. "Conceptual Modeling for ETL Processes". In: *Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. 2002, pp. 14–21. DOI: 10.1145/583890.583893 (cit. on p. 13).

[22] Franck Ravat and Yan Zhao. "Data Lakes: Trends and Perspectives". In: *Database and Expert Systems Applications (DEXA 2019)*. Lecture Notes in Computer Science. Springer, 2019, pp. 304–313. DOI: 10.1007/978-3-030-27615-7_23 (cit. on p. 13).

[23] Patrick Sawadogo and Jérôme Darmont. "On Data Lake Architectures and Metadata Management". In: *Journal of Intelligent Information Systems* 56 (2021), pp. 597–624. DOI: 10.1007/s10844-020-00608-7 (cit. on pp. 13, 88, 91).

[24] Christoph Giebler et al. "Leveraging the Data Lake: Current State and Challenges". In: *Business Information Systems*. Lecture Notes in Business Information Processing. Springer, 2019. DOI: 10.1007/978-3-030-27520-4_13 (cit. on p. 14).

[25] Ahmed A. Harby and Farhana H. Zulkernine. "Data Lakehouse: A Survey and Experimental Study". In: *Information Systems* 127 (2025), p. 102460. DOI: 10.1016/j.is.2024.102460 (cit. on pp. 14, 89).

[26] Dani Schnider. *Data Warehousing With Oracle*. Accessed: 2026-01-06. June 12, 2022. URL: https://danischnider.wordpress.com/2022/06/12/data-vault-database-vault/ (cit. on p. 15).

[27] Jeff Kramer and Jeff Magee. "Self-managed systems: an architectural challenge". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 259–268 (cit. on p. 26).

[28] Alexander Mayr. *SEF Evaluation - Requirements coverage report*. 2026. DOI: 10.5281/zenodo.18525138. URL: https://doi.org/10.5281/zenodo.18525138 (cit. on p. 65).

[29] Apache Software Foundation. *Apache Kafka Documentation*. 2026. URL: https://kafka.apache.org/documentation/ (visited on 06/29/2026) (cit. on pp. 87, 89, 90).

[30] Alejandro Vaisman and Esteban Zimányi. *Data Warehouse Systems: Design and Implementation*. Springer, 2014. ISBN: 978-3-642-54654-9. DOI: 10.1007/978-3-642-54655-6 (cit. on pp. 87, 89, 91).

[31] Apache Software Foundation. *Structured Streaming Programming Guide*. 2026. URL: https://spark.apache.org/docs/latest/streaming/apis-on-dataframes-and-datasets.html (visited on 06/29/2026) (cit. on pp. 87, 90, 91).

[32] IBM. *Control Plane vs. Data Plane*. 2026. URL: https://www.ibm.com/think/topics/control-plane-vs-data-plane (visited on 06/29/2026) (cit. on pp. 87, 88).

[33] Ronny Eichler et al. "Modeling Metadata in Data Lakes—A Generic Model". In: *Data & Knowledge Engineering* 136 (2021), p. 101931. DOI: 10.1016/j.datak.2021.101931 (cit. on pp. 88, 90).

[34] Delta Lake Project. *Delta Lake Documentation*. 2026. URL: https://docs.delta.io/ (visited on 06/29/2026) (cit. on pp. 88, 90, 91).

[35] Docker Inc. *Docker overview*. 2026. URL: https://docs.docker.com/get-started/docker-overview/ (visited on 06/29/2026) (cit. on p. 88).

[36] Docker Inc. *Docker Compose Documentation*. 2026. URL: https://docs.docker.com/compose/ (visited on 06/29/2026) (cit. on p. 88).

[37] Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. "Automating the Database Schema Evolution Process". In: *The VLDB Journal* 22 (2013), pp. 73–98. DOI: 10.1007/s00778-012-0302-x (cit. on p. 88).

[38] Stefano Rizzi. "A Survey of Data Warehouse Design Techniques". In: *ACM SIGMOD Record* 35.3 (2006), pp. 38–45. DOI: 10.1145/1183512.1183515 (cit. on p. 89).

[39] Apache Software Foundation. *KafkaConsumer API Documentation: Offsets and Consumer Position*. 2026. URL: https://kafka.apache.org/documentation/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html (visited on 06/29/2026) (cit. on p. 89).

[40] Apache Software Foundation. *Apache Parquet Documentation*. 2026. URL: https://parquet.apache.org/ (visited on 06/29/2026) (cit. on p. 90).

[41] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (visited on 06/29/2026) (cit. on p. 90).

A. Glossary

apply-and-verify An iterative working pattern in which a change (e.g., to code, configuration, or data processing logic) is applied first and then validated through checks such as tests, sanity metrics, or outputs to confirm correctness and impact. 26

at-least-once A delivery/processing guarantee where messages may be processed more than once in failure scenarios, requiring idempotent handling to avoid duplicates [29, 6]. ii, 1–3, 6, 10, 18, 23, 24, 35, 40, 44, 45, 48, 50, 51, 63, 65, 68, 70, 72, 73, 76, 79

bridge table A derived structure that materializes relationship navigation patterns (e.g., many-to-many or hierarchical traversals) to simplify queries and improve performance, typically used in Data Vault information delivery [4, 30]. 16, 81

business key A stable, business-defined identifier for an entity used to link records across systems and time [4]. 15, 16, 80

Business Vault A derived layer of a Data Vault that applies business rules and enrichments on top of the Raw Vault while preserving historization [4, 5]. 15, 16, 80, 81

checkpoint Persisted state used by a streaming system to track progress and enable recovery after failures [31]. 23, 26, 30, 32, 37, 39

control plane The logical channel used to communicate schema and coordination metadata, such as notifications, evolution plans, policy decisions and verification outcomes. In this thesis, the term follows the control/data-plane distinction in which control-plane logic determines how work is coordinated, while data-plane logic moves or processes payload data [32]. ii, 1, 3, 4, 6, 9, 10, 17–21, 23, 24, 45, 46, 48–51, 55–57, 60, 63, 65–67, 70, 73–76, 78, 79, 81, 82

Data Lake A centralized repository for raw and curated data (structured, semi-structured, and unstructured), typically stored on an object store and consumed under schema-on-read [16, 23]. 91, 93, 94, 97

data lineage Recorded information about the origin and derivation of a dataset, including the upstream sources and transformations that produced it. Lineage supports traceability, auditing, impact analysis, and reproducible reprocessing [8, 33]. 2, 5, 7

data plane The logical channel used to communicate and persist payload data, such as row deltas, that is subject to ingestion and transformation. In this thesis, the term follows the control/data-plane distinction in which the data plane carries the operational data path and the control plane coordinates processing decisions [32]. 4, 6, 17, 18, 20, 23, 24, 45, 46, 49, 50, 55, 57, 63

data swamp A degraded data lake in which missing or low-quality metadata and governance prevent efficient discovery, understanding, and reuse of stored data [16, 23]. 13

Data Vault A data warehouse modeling approach that separates business keys (hubs), relationships (links), and descriptive attributes (satellites) to support historization and change [4, 5]. 93, 94, 97

Delta Lake A storage layer that adds ACID transactions and schema evolution to data lake tables via a transaction log [18, 34]. 10, 14, 88

Delta table A table abstraction provided by Delta Lake that stores data as immutable data files in object store and manages table state through a transactional log. Delta tables support Atomicity, Consistency, Isolation, Durability (ACID)-like guarantees, schema enforcement and controlled schema evolution and reproducible reads via table-versioning [18, 34]. 14

Docker An open-source platform for building, packaging, and running applications in lightweight, isolated containers. It uses container images to provide consistent runtime environments across development and production and supports workflows such as container builds (Dockerfiles), registries, and orchestration integration [35]. 67, 88, 94–96

Docker Compose A tool for defining and running multi-container applications with Docker. Services, networks, and volumes are declared in a YAML configuration (typically `compose.yaml` or `docker-compose.yml`), enabling reproducible local environments and one-command workflows such as building, starting, and stopping a complete application stack [36]. 68, 94

evolution plan A concrete sequence of schema and data transformations required to transition a target system from one schema state to another under safety constraints [37]. 4

Filewatcher An ingestion-boundary component that monitors a directory, derives observations, and publishes schema notifications and row deltas to downstream systems. 1, 4, 5, 19, 20, 26, 27, 35, 43–46, 55–57, 66, 67, 92, 94

hash key A derived identifier computed from one or more key fields (e.g., a business key) to obtain a fixed-length surrogate used in modeling and joins [4]. 16

hashdiff A hash value computed from descriptive attributes to detect changes efficiently, commonly used in Data Vault satellites [4]. 16

Hub A Data Vault construct that stores unique business keys and their identity metadata [4, 5]. 1, 15, 16, 25, 27–29, 32, 41, 53, 57, 60, 80

Information mart A consumption-oriented, dimensional delivery layer derived from an integrated backend (e.g., Data Vault or EDW), providing user-facing facts and dimensions optimized for analytics and reporting [30, 38]. 15, 81

interpret-and-plan An analytical working pattern in which evidence (e.g., logs, metrics, data characteristics, or results) is interpreted first to form a diagnosis or understanding, followed by planning targeted actions or experiments before making changes. 26

Kafka A distributed event streaming platform used here as an event bus for decoupling producers and consumers, buffering, and replay [29]. 1, 18, 66, 91

Kafka offset A numerical position assigned to each record within a Kafka partition. A consumer position denotes the next offset to be read, and committed offsets allow consumers and consumer groups to resume processing or intentionally replay earlier records after failure or restart [29, 39]. 18

lakehouse An analytical platform style that combines data lake storage with warehouse-like management features such as transactional guarantees, schema enforcement, and governed evolution [25, 18]. 6, 10, 12, 14, 40, 75, 81

Link A Data Vault construct that represents associations between business keys (e.g., between hubs) and supports relationship historization [4, 5]. 1, 15, 16, 27–29, 32, 41, 52, 53, 57, 60

metadata Data describing other data, used to enable interpretation, management, and governance of datasets. In analytical platforms, metadata typically covers structural information (schemas, types), operational information (ingestion times, processing status, lineage), and

semantic information (business definitions, ownership, policies) [33, 16]. 1, 2, 4, 7–11, 14, 24, 26–29, 31, 32, 35, 37, 41, 45, 48, 51, 67, 68, 79

micro-batch A streaming execution mode that processes incoming data in small batches, typically with periodic trigger intervals [31]. 90

object store A storage system that exposes immutable objects via key-based APIs and typically decouples storage from compute, commonly used as the persistence layer for data lakes [18, 34]. 13, 88

observation A single ingestion unit derived from an input file, comprising a schema snapshot and corresponding row deltas used for downstream processing and evolution. 3, 4, 9, 43, 78, 79

Parquet A columnar storage format optimized for analytical workloads, commonly used for table-like data in data lakes and lakehouses [40]. 94, 97, 98

payload batch A collection of individual payloads (e.g., records or messages) grouped and handled as a single unit for transmission or processing—often to amortize per-message overhead and improve throughput; in streaming systems it commonly corresponds to the set processed in one micro-batch [31]. 21

publisher A component role responsible for emitting control-plane or data-plane events to the event bus [29]. 37

Raw Vault The foundational layer of a Data Vault that stores ingested data with minimal business transformation while preserving history [4, 5]. 15, 16, 67, 75, 80, 81, 93

REST An architectural style for networked applications in which resources are identified by URIs and manipulated via a uniform interface (commonly HTTP methods such as GET, POST, PUT, DELETE). REST emphasizes stateless client–server interactions, cacheability, and layered system design; web APIs that follow these constraints are often called RESTful [41]. 32

row delta A set of row-level changes derived from comparing successive observations of an input file, emitted for downstream ingestion and transformation. 4

Satellite A Data Vault construct that stores descriptive attributes and change history for a hub or link [4, 5]. 1, 15, 16, 25, 27–30, 32, 41, 52, 53, 57, 60

schema notification A control-plane event that announces a schema snapshot and enables downstream components to plan and verify schema evolution. 4

schema snapshot A point-in-time representation of a dataset schema (e.g., field names, types, and ordering) for a given observation. 2, 5, 9

schema-on-read A data management principle where structural interpretation is applied at consumption time rather than enforced during ingestion [16, 23]. 13, 14, 88

schema-on-write A data management principle where data is validated and conformed to a predefined schema at ingestion time [30]. 13, 14

Spark A distributed data processing engine used here for streaming ingestion and transformation [31]. 13

stable persistence surface A durable, queryable persistence contract at the boundary between an upstream change stream (e.g., deltas published to Kafka) and the Data Lake. It stabilizes ingestion for downstream consumers by materializing semi-structured change envelopes into persistent, table-scoped lake objects (i.e., well-defined tables/partitions plus the required metadata to support consistent reads, recovery, and reproducibility). 46

time travel The ability to query historical snapshots of a dataset or table as of a prior version or timestamp [18, 34]. 14

B. Requirements

B.1. Overview and elicitation

This appendix documents the functional and non-functional requirements for the prototype data platform and SEF. The requirements serve as the basis for deriving the architecture, implementing the prototype and structuring the evaluation.

B.2. Functional requirements

Table B.1.: Functional requirements (FR)

ID	Area	Requirement
FR-1	File monitoring & ingestion	The system shall monitor a monitored directory (which could be a local filesystem, SMTP server or fileshare) for CSV file changes.
FR-2	File monitoring & ingestion	The Filewatcher shall detect file modifications and creations.
FR-3	File monitoring & ingestion	The Filewatcher shall detect schema evolution events within CSV files, including new columns, changed data types and attribute deletions.
FR-4	File monitoring & ingestion	Upon detecting a file change or schema evolution, the Filewatcher shall send an event to the streaming service with relevant details (including file path, change action and schema evolution flag).

Continued on next page

ID	Area	Requirement
FR-5	Data streaming & processing	The system shall use streaming for data change events.
FR-6	Data streaming & processing	The streaming consumer shall listen to and process data events and convert CSV data into a data frame.
FR-7	Data streaming & processing	The processing component shall support two operational modes: (i) real-time streaming mode, for immediate processing and update of the Data Lake, and (ii) batch processing mode, for aggregated change handling while still using streaming as a buffer.
FR-8	Data Lake handling	The Data Lake handler shall automatically create the required storage if the required storage is not already available.
FR-9	Data Lake handling	The system shall use the CSV file's name as the table name in the DBMS or as the name of the file if using file storage.
FR-10	Data Lake handling	The Data Lake handler shall store processed data based on the configured mode (real-time updates or batch changes).
FR-11	Data Vault integration	The system shall ingest data from the Data Lake into a Data Vault consisting of a Raw Vault.
FR-12	Data Vault integration	The Raw Vault component shall automatically ingest any new or updated data from the Data Lake.
FR-13	Data Vault integration	The system shall support fully automated Raw Vault operations, tracking data changes and schema evolutions to update the vault accordingly.

Continued on next page

ID	Area	Requirement
FR-14	SEF	The SEF shall be integrated with the Filewatcher, the Data Lake handler and the Data Vault layer.
FR-15	SEF	On detection of a schema evolution, the SEF shall automatically: (i) update the schema in the Data Lake (adjusting table definitions or Parquet file structures), and (ii) update the relevant objects in the unified Data Vault.
FR-16	SEF	The SEF shall listen to a dedicated streaming topic for schema change events.
FR-17	Containerization & packaging	Each major component (Filewatcher, streaming service, streaming consumer, Data Lake handler, Data Vault, ETL framework) shall be deployed in a dedicated Docker container.
FR-18	Containerization & packaging	Each major component shall be packaged as a Python package to encapsulate functionality, dependencies and allow for version control.
FR-19	Containerization & packaging	Docker Compose shall be used to orchestrate and manage the dependencies between the components.
FR-20	Logging & monitoring	The system shall implement detailed logging at multiple levels: DEBUG (detailed troubleshooting and internal state information), INFO (general operation and successful processing messages), WARNING (unexpected situations that do not halt processing), ERROR (significant issues that could impact functionality), CRITICAL (unrecoverable errors).

Continued on next page

ID	Area	Requirement
FR-21	Logging & monitoring	Log messages should include timestamps, log levels and context details to assist with diagnostics.

B.3. Non-functional requirements

Table B.2.: Non-functional requirements (NFR)

ID	Area	Requirement
NFR-1	Performance & scalability	The system shall support real-time streaming with low-latency processing.
NFR-2	Performance & scalability	The application should be scalable, allowing additional instances of processing components to be spawned as needed.
NFR-3	Performance & scalability	Batch processing mode should efficiently process aggregated data changes without performance degradation.
NFR-4	Reliability & availability	The system shall be highly available, utilizing Docker container orchestration to automatically restart failed services.
NFR-5	Reliability & availability	Data integrity must be maintained during storage and transformation and rollback procedures should be in place in case of errors.
NFR-6	Security	Data access must be secured, with proper authentication and authorization for access to RDBMS and other data stores.

Continued on next page

ID	Area	Requirement
NFR-7	Security	Communication between components shall be secured, with best practices applied for streaming service and Docker container networking.
NFR-8	Maintainability & extensibility	The codebase shall be modular and organized into separate packages, making individual components easy to update or replace.
NFR-9	Maintainability & extensibility	Detailed logging and error reporting are essential for maintainability.
NFR-10	Maintainability & extensibility	The system architecture should allow for future enhancements, such as adding new processing pipelines or integrating additional data sources.
NFR-11	Portability	The entire system should be containerized and easily portable across different environments (development, testing, production).
NFR-12	Portability	Configuration parameters such as storage mode, streaming settings and database credentials must be externalized via environment variables or configuration files to allow for flexibility.

B.4. Requirement coverage

The statuses in this table describe automated E2E evidence coverage, not a standalone final judgment of requirement satisfaction. Requirement satisfaction is discussed in Chapter 9 by combining this automated evidence with architectural and implementation inspection.

Table B.3.: E2E evidence coverage mapping for the E2E-only run

ID	Type	Area	Rule	E2E overall	RDBMS	Parquet	E2E test IDs
FR-1	FR	File monitoring & ingestion	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE
FR-2	FR	File monitoring & ingestion	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-IDEM E2E-SE
FR-3	FR	File monitoring & ingestion	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-SE E2E-PLAN-ADD E2E-PLAN-WIDEN E2E-BLOCK-DROP E2E-POLICY-NARROW
FR-4	FR	File monitoring & ingestion	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE E2E-ORDER
FR-5	FR	Data streaming & processing	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE E2E-ORDER
FR-6	FR	Data streaming & processing	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE
FR-7	FR	Data streaming & processing	partial_by_design	PARTIAL	PASS	PASS	E2E-DL E2E-SE
FR-8	FR	Data Lake handling	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE
FR-9	FR	Data Lake handling	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE
FR-10	FR	Data Lake handling	partial_by_design	PARTIAL	PASS	PASS	E2E-DL E2E-SE
FR-11	FR	Data Vault integration	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE
FR-12	FR	Data Vault integration	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-DL E2E-SE E2E-IDEM
FR-13	FR	Data Vault integration	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-SE E2E-PLAN-ADD E2E-PLAN-WIDEN E2E-ORDER E2E-CONV-RESTART

Continued on next page

ID	Type	Area	Rule	E2E overall	RDBMS	Parquet	E2E test IDs
FR-14	FR	SEF	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-SE E2E-PLAN-ADD E2E-PLAN-WIDEN E2E-BLOCK-DROP E2E-POLICY-NARROW
FR-15	FR	SEF	all_tests_pass_all_backends	PASS	PASS	PASS	E2E-SE E2E-PLAN-ADD E2E-PLAN-WIDEN
FR-16	FR	SEF	partial_by_design	PARTIAL	PASS	PASS	E2E-SE E2E-ORDER
FR-17	FR	Containerization & packaging	gap	GAP	GAP	GAP	–
FR-18	FR	Containerization & packaging	gap	GAP	GAP	GAP	–
FR-19	FR	Containerization & packaging	partial_by_design	PARTIAL	PASS	PASS	E2E-DL
FR-20	FR	Logging & monitoring	gap	GAP	GAP	GAP	–
FR-21	FR	Logging & monitoring	gap	GAP	GAP	GAP	–
NFR-1	NFR	Performance & scalability	gap	GAP	GAP	GAP	–
NFR-2	NFR	Performance & scalability	gap	GAP	GAP	GAP	–
NFR-3	NFR	Performance & scalability	gap	GAP	GAP	GAP	–
NFR-4	NFR	Reliability & availability	partial_by_design	PARTIAL	PASS	PASS	E2E-CONV-RESTART
NFR-5	NFR	Reliability & availability	partial_by_design	PARTIAL	PASS	PASS	E2E-IDEM E2E-CONV-RESTART E2E-VERIFY-FAIL
NFR-6	NFR	Security	gap	GAP	GAP	GAP	–
NFR-7	NFR	Security	gap	GAP	GAP	GAP	–
NFR-8	NFR	Maintainability & extensibility	gap	GAP	GAP	GAP	–
NFR-9	NFR	Maintainability & extensibility	partial_by_design	PARTIAL	PASS	PASS	E2E-VERIFY-FAIL E2E-POLICY-NARROW E2E-BLOCK-DROP
NFR-10	NFR	Maintainability & extensibility	gap	GAP	GAP	GAP	–
NFR-11	NFR	Portability	partial_by_design	PARTIAL	PASS	PASS	E2E-DL
NFR-12	NFR	Portability	partial_by_design	PARTIAL	PASS	PASS	E2E-DL E2E-SE

Table B.4.: Legend for compact E2E test identifiers used in Table B.3

Test ID	Full test identifier
E2E-DL	tests.e2e.test_pipeline_functional::test_data_loading_path
E2E-SE	tests.e2e.test_pipeline_functional::test_schema_evolution_path
E2E-IDEM	tests.e2e.test_idempotency::test_reprocessing_same_file
E2E-PLAN-ADD	tests.e2e.test_schema_evolution_plan_correctnes::test_plan_correctness_add_column
E2E-PLAN-WIDEN	tests.e2e.test_schema_evolution_plan_correctnes::test_plan_correctness_change_type_widening
E2E-BLOCK-DROP	tests.e2e.test_schema_evolution_negative::test_schema_evolution_drop_is_blocked
E2E-POLICY-NARROW	tests.e2e.test_policy_widen_only::test_policy_widen_only_blocks_narrowing_change
E2E-ORDER	tests.e2e.test_fault_injection::test_out_of_order_notifications_do_not_drop_schema
E2E-CONV-RESTART	tests.e2e.test_idempotency_convergence::test_convergence_duplicate_notifications_with_transient_handler_restart
E2E-VERIFY-FAIL	tests.e2e.test_fault_injection::test_verification_fails_when_vault_handler_down