KDL: A Version-Native Data Lake Architecture for AI-Driven Enterprise Analytics

Kaman AI Research

March 2026

Abstract

Modern enterprise analytics platforms face a fundamental tension between *data freshness*—the ability to ingest and query data in near-real time—and *auditability*—the ability to reconstruct any historical state of the data for compliance, debugging, or reproducibility. Traditional data lakes address one axis at the expense of the other: append-optimized stores sacrifice efficient point-in-time queries, while snapshot-heavy architectures introduce write-amplification that degrades ingestion throughput.

We present **KDL** (Kaman Data Lake), a version-native lakehouse architecture that resolves this tradeoff through three co-designed tiers: a *relational catalog* backed by PostgreSQL for transactional metadata, an *embedded columnar engine* powered by DuckDB for sub-second analytical queries, and *object storage* (S3-compatible) for durable, immutable Parquet data files. Every mutation in KDL produces an immutable snapshot, enabling SQL-based time travel, change tracking, and full audit trails without external versioning tooling.

KDL is purpose-built for AI agent workloads, where autonomous systems issue heterogeneous queries—ranging from schema introspection to multi-table joins—against continuously ingested data. The system supports multi-tenant isolation, write buffering for high-throughput scenarios, a circuit breaker pattern for graceful degradation, and OpenLineage-compatible data lineage tracking. We describe the architecture, key design decisions, and performance characteristics, and position KDL relative to Delta Lake, Apache Iceberg, and related systems.

1 Introduction

The rise of AI agents in enterprise settings has created a new class of data infrastructure requirements. Unlike human analysts who issue a handful of carefully crafted queries per session, AI agents operate autonomously: they discover schemas, join tables across domains, ingest results from external connectors, and iterate on analytical hypotheses —all within a single conversation turn. This workload profile demands *sub-second query latency* on moderately sized datasets (millions to low billions of rows), *real-time ingestion* from heterogeneous sources, *complete version history* for auditability and reproducibility, and *multi-tenant isolation* in shared deployments.

Existing lakehouse formats address subsets of these requirements. **Delta Lake** [1] provides ACID transactions and time travel over Spark, but its reliance on a JSON-based transaction log and Spark's JVM startup overhead make it unsuitable for the interactive, sub-second query patterns that AI agents require. **Apache Iceberg** [2] offers a more portable metadata layer with manifest files and snapshot isolation, yet its query planning still assumes batch-oriented engines and introduces considerable complexity for operational deployments. **Apache Hudi** [3] focuses on incremental processing and upserts, but its merge-on-read architecture can degrade read performance under high-frequency small writes—precisely the pattern AI connectors produce.

These systems were designed for data engineering pipelines: scheduled ETL jobs, batch transformations, and periodic materialized views. AI agent workloads, by contrast, are *interactive, unpredictable, and write-heavy at small granularity*. A single agent session may create a table, insert rows from an external API, query the result, discover a schema mismatch, alter the table, re-ingest, and compare the current state against a version from minutes ago—all within seconds.

KDL was designed from first principles to serve this workload. Its key contributions are:

1. A **three-tier architecture** that separates catalog metadata, compute, and storage, enabling independent scaling and strong isolation guarantees.
2. A **version-native data model** where every write operation automatically produces an immutable snapshot, enabling SQL-based time travel without external tooling.
3. A **query execution engine** with automatic error recovery, circuit breaker protection, and per-

table write serialization that permits cross-table parallelism.

4. A **multi-layer caching architecture** spanning connection pools, object storage intercepts, metadata caches, and query result caches, designed for warm-start preservation across restarts.
5. An **OpenLineage-compatible lineage tracker** that captures provenance at query granularity without requiring changes to client applications.

The remainder of this paper is organized as follows. Section 2 presents the overall architecture. Section 3 describes the DuckLake catalog layer. Section 4 details version control and time travel. Section 5 covers the query execution engine. Section 6 describes the caching architecture. Section 7 covers data ingestion. Section 8 discusses access control. Section 9 presents lineage tracking. Section 10 provides performance analysis. Section 11 discusses related work, and Section 12 concludes.

2 Architecture Overview

KDL employs a three-tier architecture in which each tier is independently replaceable and horizontally scalable:

1. **Catalog Tier (PostgreSQL).** A relational database stores all metadata: lake registrations, table schemas, snapshot histories, access control policies, lineage graphs, and tenant isolation boundaries. PostgreSQL was chosen for its mature transaction support, row-level locking, and broad operational tooling. The catalog is shared across all KDL instances in a deployment.
2. **Compute Tier (DuckDB).** An embedded columnar OLAP engine executes analytical queries in-process. Each KDL instance maintains a pool of DuckDB connections, sized according to available CPU and memory resources. DuckDB's vectorized execution engine delivers sub-second latency on datasets that fit comfortably in the hundreds-of-gigabytes range, without requiring a distributed query planner.
3. **Storage Tier (S3/Parquet).** Data files are stored as immutable Apache Parquet files on S3-compatible object storage (AWS S3, MinIO, or equivalent). Writes never mutate existing files; instead, new Parquet files are created for each transaction, and the catalog records the mapping from logical tables to physical files.

2.1 Interaction Model

A typical query flows through the tiers as follows. An API request arrives at the FastAPI service layer, which dispatches the operation to a worker in the DuckDB thread pool. The worker checks out a connection from the per-lake connection pool, attaches the DuckLake catalog (which reads metadata from PostgreSQL), and executes the query against Parquet files on S3. Results are serialized and returned to the caller.

Write operations follow a similar path but acquire a per-table asynchronous lock before execution. This lock is scoped to the `(lake, schema, table)` triple, meaning writes to different tables within the same lake proceed concurrently, while writes to the same table are serialized to maintain snapshot consistency.

2.2 Connection Pooling

KDL maintains a connection pool per lake, with lazy growth from zero to a configurable maximum. Each connection in the pool is initialized with the DuckLake extension, configured with tenant-specific S3 credentials, and assigned a dedicated local cache directory. Connections are recycled after a configurable idle timeout to handle credential rotation and prevent resource leaks.

The thread pool that drives DuckDB operations is sized independently of the connection pool. A bounded thread pool executor prevents the asynchronous event loop from blocking on DuckDB's synchronous API, while allowing multiple concurrent reads across different lakes.

Memory allocation follows an adaptive model. Rather than dividing total available memory equally across all pool connections, KDL assumes that only a small subset of connections (k) will execute memory-intensive queries simultaneously:

$$M_{\text{conn}} = \max\left(\frac{M_{\text{total}} \cdot \alpha}{\min(P,k)},\ M_{\text{floor}}\right) \tag{1}$$

where M_{total} is the total system memory, α is the fraction allocated to DuckDB (typically 0.75), P is the pool size, k is the assumed concurrency for heavy queries, and M_{floor} is a minimum per-connection guarantee.

3 The DuckLake Catalog

At the core of KDL's versioning model is **DuckLake**, a catalog extension for DuckDB that stores table metadata in PostgreSQL while keeping data files on object storage. This separation of concerns is the key architectural insight that enables KDL's combination of transactional metadata management and scalable data storage.

3.1 Metadata Separation

Traditional lakehouse formats embed metadata alongside data. Delta Lake stores its transaction log as JSON files in the same directory as the Parquet data files. Iceberg maintains manifest files and manifest lists as Avro files co-located with data. Both approaches couple metadata management to the storage layer, creating challenges for concurrent access, atomic multi-table operations, and metadata queries.

DuckLake takes a different approach: all metadata—table definitions, column schemas, snapshot records, and file references—resides in PostgreSQL tables within a dedicated catalog database. This design offers several advantages:

- **Transactional consistency.** Metadata operations inherit PostgreSQL's ACID guarantees, including row-level locking for concurrent catalog updates.
- **Efficient metadata queries.** Schema introspection, snapshot enumeration, and table listings execute as standard SQL queries against indexed relational tables, rather than requiring file listing operations on object storage.
- **Cross-table atomicity.** Multi-table schema changes can be coordinated through PostgreSQL transactions, a capability that file-based catalogs struggle to provide.

3.2 Snapshot-Based Versioning

Every write operation in DuckLake—insert, update, delete, or schema alteration—produces a new *snapshot*. A snapshot is an immutable record in the catalog that captures the complete logical state of the database at a point in time. Snapshots form a linear chain, with each snapshot referencing its parent.

This model differs from Delta Lake's approach, where the transaction log is a sequence of JSON actions that must be replayed to reconstruct state, and from Iceberg's approach, where manifest lists reference manifests that reference data files in a multi-level indirection. DuckLake's snapshot model is conceptually simpler: each snapshot directly records which data files constitute each table at that point, and the catalog provides indexed access to the snapshot history.

Table 1 summarizes the key differences.

Table 1: Catalog design comparison across lakehouse systems.

Property	Delta	Iceberg	DuckLake
Metadata store	JSON files	Avro manifests	PostgreSQL
Versioning unit	Log actions	Snapshots	Snapshots
State reconstruction	Log replay	Manifest traversal	Direct lookup
Cross-table txn	No	Limited	PostgreSQL txn
Metadata queries	File scan	File scan	SQL index scan

4 Version Control and Time Travel

KDL's version-native design means that every data mutation is automatically versioned. Users do not opt into versioning or configure retention policies before they can access historical states—versioning is the default and only mode of operation.

4.1 SQL-Based Time Travel

KDL supports querying any historical snapshot through a version-qualified SQL syntax:

Listing 1: Time travel query syntax.

```sql
-- Query table as of snapshot 42
SELECT * FROM schema.table
  AT (VERSION => 42)
```

The `AT (VERSION =>` *n*`)` clause instructs the DuckLake catalog to resolve table references against snapshot *n* rather than the current head. This resolution is performed entirely within the catalog layer; the query engine itself is unaware of versioning semantics.

4.2 Change Tracking

Beyond point-in-time queries, KDL provides fine-grained change tracking between any two snapshots. The system supports three change query modes:

- **Insertions:** Rows present in the target snapshot but absent from the source snapshot.
- **Deletions:** Rows present in the source snapshot but absent from the target snapshot.
- **Changes:** A unified view that labels each differing row with its change type, enabling audit-trail reconstruction.

Change detection is implemented via set-difference operations on the versioned table references. While this approach has $O(n)$ complexity in the table size, DuckDB's columnar execution and Parquet predicate pushdown make it practical for tables with millions of rows.

4.3 Audit Trail Computation

The combination of automatic snapshots and change tracking enables comprehensive audit trails without external tooling. Each snapshot records a timestamp and a summary of changes (insertions, deletions, modifications per table). The catalog's metadata tables provide indexed access to the snapshot history for any table, including the originating snapshot, the range of snapshots in which a table version was active, and the specific changes made at each snapshot.

This contrasts with systems that require separate audit logging infrastructure. In KDL, the version history *is* the audit trail.

5 Query Execution Engine

KDL's query execution engine wraps DuckDB with several layers of reliability and adaptability.

5.1 Thread Pool Execution Model

DuckDB operates as an embedded, single-process database. To integrate with KDL's asynchronous service layer, all DuckDB operations are dispatched to a bounded thread pool. This design prevents blocking the event loop while allowing concurrent query execution across different lakes.

The thread pool size is independent of the connection pool size. A typical deployment uses a thread pool of 20 workers and per-lake connection pools of 3–5 connections, tuned based on workload characteristics.

5.2 Write Serialization

Concurrent writes to the same table can cause snapshot conflicts in the DuckLake catalog. KDL addresses this with a hierarchical locking scheme:

- **Table-level locks.** An asynchronous lock keyed on the `(lake, schema, table)` triple serializes write operations to the same table. Writes to different tables proceed concurrently.
- **Row-level locks.** For upsert operations, KDL acquires fine-grained locks on individual rows (keyed on primary key values) to prevent race conditions during concurrent upserts. Row locks are acquired in sorted key order to prevent deadlocks.
- **Distributed locking.** In multi-instance deployments, row locks are implemented via a distributed key-value store with atomic compare-and-set semantics and automatic expiration, ensuring that crashed processes do not hold locks indefinitely.

Lock acquisition is bounded by a configurable timeout. If a lock cannot be acquired within the timeout, the operation fails with a descriptive error rather than blocking indefinitely.

5.3 Automatic Error Recovery

AI agents frequently generate SQL that contains minor syntactic or semantic errors—date format mismatches, JSON accessor syntax differences, type coercion issues. Rather than propagating these errors back through the agent's reasoning loop (which may cost additional LLM inference), KDL includes an *auto-fix engine* that attempts to correct common errors transparently.

The auto-fix engine maintains a registry of error patterns, each associated with a confidence level and a correction function. When a query fails, the error message is matched against the registry. If a high-confidence pattern matches, the correction is applied and the query is re-executed (up to a bounded number of retries). The response includes metadata indicating whether a fix was applied, preserving transparency for debugging.

This approach amortizes the cost of common errors across all agent interactions, reducing round-trip latency by avoiding unnecessary LLM re-inference for fixable issues.

5.4 Circuit Breaker Pattern

To prevent cascading failures when the compute or storage tier is overloaded, KDL implements a circuit breaker pattern for DuckDB operations. The circuit breaker has three states:

1. **Closed** (normal): Requests pass through. Consecutive failures are counted.
2. **Open** (failing): Requests are rejected immediately without attempting execution. After a configurable reset timeout, the circuit transitions to half-open.
3. **Half-open** (testing): A limited number of requests are allowed through. If they succeed, the circuit closes; if any fails, the circuit reopens.

The failure threshold, success threshold, and reset timeout are independently configurable per subsystem (compute, catalog, cache), reflecting the different recovery characteristics of each tier.

6 Caching Architecture

KDL employs a multi-layer caching strategy to minimize latency and reduce load on the storage and catalog tiers.

6.1 Connection Pool Caching

As described in Section 2, each lake maintains a pool of pre-initialized DuckDB connections. These connections persist the DuckLake catalog attachment, S3 credential configuration, and extension state across queries, avoiding the overhead of repeated initialization.

Connections are recycled after a configurable idle timeout, balancing resource conservation against warm-start preservation. A background sweep thread periodically identifies and closes idle connections that exceed the timeout.

6.2 Object Storage Caching

Parquet files read from S3 are cached on local SSD storage. The caching layer intercepts object storage read operations at the DuckDB extension level, storing fetched objects in per-lake cache directories. This approach is transparent to the query engine: subsequent reads of the same Parquet file are served from local disk rather than S3.

Cache directories are isolated per lake and per connection slot to prevent cross-tenant data leakage and avoid contention on shared cache state. The cache is sized dynamically based on available disk space:

$$C_{\text{size}} = \min\left(C_{\max},\ D_{\text{avail}} \cdot \beta\right) \tag{2}$$

where $C_{\max}$ is the configured maximum cache size, D_{avail} is the available disk space, and β is the fraction allocated to caching (typically 0.5–0.7).

6.3 Metadata Caching

Frequently accessed metadata—lake configurations, table schemas, access control policies, and storage credentials—is cached in a distributed in-memory store with TTL-based expiration. KDL employs a two-level cache hierarchy:

- **L1 (process-local):** A thread-safe in-memory dictionary with short TTLs, serving as a fast path for repeated lookups within the same process.
- **L2 (distributed):** A shared cache store accessible across all KDL instances, with longer TTLs, ensuring consistency in multi-instance deployments.

Cache invalidation is event-driven: DDL operations and write operations invalidate the relevant cache entries at both levels. Security-sensitive entries (access control policies) use shorter TTLs than configuration entries.

6.4 Query Result Caching

Read-only queries can optionally be cached at the result level, keyed on a hash of the SQL text and the target snapshot version. Version-pinned queries are particularly amenable to caching, as the underlying data is immutable by definition. Write operations to a table invalidate all cached query results for that table.

7 Data Ingestion Pipeline

KDL supports ingestion from heterogeneous external sources through a connector framework based on the Model Context Protocol (MCP), a standardized protocol for AI agent tool integration.

7.1 MCP-Based Connectors

External data sources—relational databases, APIs, document stores, SaaS platforms—are exposed as MCP resources. Each connector implements resource listing, schema discovery, and data reading through the MCP protocol. KDL's ingestion pipeline connects to these sources, reads available resources, and maps them to tables in the data lake.

The MCP client implementation includes exponential backoff retry logic, connection validation, and detailed diagnostic capture for failed connections. Non-retryable errors (authentication failures, resource not found) are detected via pattern matching and fail immediately rather than consuming retry budget.

7.2 Schema Detection and Column Mapping

When a new data source is connected, KDL automatically detects the source schema and generates default column mappings. The mapping layer supports:

- Type coercion between source and target type systems.
- Column renaming and filtering.
- Schema change detection via hash-based comparison of source schema snapshots.
- Incremental sync modes with cursor-based tracking of the last ingested position.

Schema changes in the source are detected at sync time by comparing the current source schema against a stored snapshot. When changes are detected, the system flags the data source for review rather than silently applying potentially destructive schema alterations.

7.3 Write Buffering

For high-throughput ingestion scenarios, KDL provides an optional write buffer that decouples ingestion rate from write execution rate. Operations are buffered in a distributed queue, grouped by target table, and flushed in configurable batches.

The buffer supports two flush triggers:

- **Time-based:** A background task flushes all pending operations at a configurable interval (default: 5 seconds).
- **Quantity-based:** When the number of buffered operations for a single table exceeds a threshold (default: 1000 operations), an immediate flush is triggered.

Each buffered operation receives a unique acknowledgment identifier that clients can use to track operation status (buffered, flushing, completed, or failed). A distributed lock prevents multiple KDL instances from flushing the same buffer concurrently.

7.4 Bulk Insert Queue

For large-scale batch ingestion, KDL provides a durable message queue-based bulk insert system. Insert requests are accepted immediately and assigned a job identifier. A background worker processes jobs sequentially, using the DuckDB write lock to prevent conflicts. The queue provides at-least-once delivery semantics with automatic retry and exponential backoff for transient failures.

8 Access Control and Multi-Tenancy

KDL provides fine-grained access control at multiple levels, integrated with external authentication services.

8.1 Multi-Tenant Storage Isolation

Each tenant (organization) is assigned isolated storage paths on object storage. Lake creation automatically scopes the data path to the tenant's storage prefix, ensuring that data files from different tenants never share storage locations. Each tenant can configure independent S3 credentials and storage endpoints, supporting hybrid deployments where different tenants use different cloud providers.

8.2 Row-Level Security

KDL supports row-level security (RLS) through declarative filter policies. Each policy specifies a predicate expression and the roles to which it applies. When a query is executed, applicable row filters are transparently injected into the query's `WHERE` clause.

Row filters support dynamic parameterization through user context fields, enabling policies such as "users in the sales department can only see rows where `region` matches their assigned territory." The filter injection operates at the SQL level, allowing the DuckDB optimizer to incorporate row filters into its query plan for efficient predicate pushdown.

8.3 Column Masking

Sensitive columns can be masked through declarative masking policies. KDL supports several built-in mask functions:

- **Redaction:** Complete replacement with a sentinel value.
- **Hashing:** Deterministic one-way hash (useful for referential integrity without revealing val-

ues).
- **Partial masking:** Revealing a configurable number of leading and trailing characters.
- **Format-specific masking:** Specialized masks for email addresses, phone numbers, and similar structured data.

Masking policies are applied as SQL expression rewrites during query planning. The original column values are never exposed to unauthorized roles, even in intermediate query results.

8.4 RBAC Integration

Access control policies are role-based and integrate with KDL's external authentication service. User roles are resolved at query time from the authentication token, and the applicable row filters and column masks are determined based on the intersection of the user's roles and the policies defined for each table.

Policy definitions are cached with short TTLs (on the order of minutes) to balance security responsiveness against lookup performance.

9 Data Lineage with OpenLineage

KDL implements data lineage tracking using the OpenLineage standard, an open framework for collecting and analyzing metadata about data pipelines.

9.1 Event-Based Lineage Capture

Every query executed in KDL generates an OpenLineage event that records:

- The **job** (the query or operation being performed).
- The **run** (a unique execution instance with start, complete, and fail events).
- The **input datasets** (tables read by the query).
- The **output datasets** (tables written by the query).
- **Facets** (additional metadata such as the SQL text, schema information, and data quality metrics).

Resources are identified using a URI scheme: `kdl://`*`lake/schema/table`*, enabling unambiguous cross-lake lineage tracking.

9.2 Lineage Graph Construction

Lineage events are processed into a directed acyclic graph (DAG) of *lineage nodes* (tables) and *lineage edges* (data dependencies). Edges are typed (e.g., `DERIVES_FROM`, `USES`, `PRODUCES`) and annotated with the transformation SQL and the originating run.

The lineage graph supports both upstream and downstream traversal with configurable depth limits, enabling impact analysis queries such as "what downstream tables would be affected if I dropped this column?"

9.3 Impact Analysis

KDL provides an impact analysis API that, given a proposed schema change, traverses the downstream lineage graph and returns the set of affected tables. This enables AI agents to reason about the consequences of schema modifications before executing them, reducing the risk of cascading failures in complex data pipelines.

10 Performance Characteristics

We characterize KDL's performance across several dimensions relevant to AI agent workloads.

10.1 Query Latency

Table 2 presents representative latency measurements for common operation types on a dataset of approximately 10 million rows across 20 columns, stored as Parquet on S3 with local disk caching enabled.

Table 2: Representative latency by operation type (10M rows, cached).

Operation	p50 (ms)	p95 (ms)	p99 (ms)
Point query (indexed)	8	25	45
Aggregation (GROUP BY)	85	180	320
Time travel query	90	200	350
Change tracking (2 versions)	150	400	700
Single-row insert	12	35	60
Batch insert (1K rows)	45	120	250
Batch insert (100K rows)	350	800	1400
Schema introspection	5	15	30

Time travel queries incur only a modest overhead (∼5–10%) compared to current-version queries, because the version resolution happens in the catalog tier (a PostgreSQL index lookup) rather than requiring data-layer operations.

10.2 Write Throughput

KDL's write performance is bounded by three factors: the table-level write lock, Parquet file generation, and S3 upload latency. For single-row inserts, throughput is approximately 80–100 operations per second per table. The write buffer increases effective throughput to 2,000–5,000 rows per second by amortizing lock acquisition and S3 upload costs across batches.

10.3 Scaling Strategies

KDL supports both vertical and horizontal scaling:

- **Vertical:** Increasing CPU cores and memory directly improves DuckDB's vectorized execution performance. Memory allocation is adaptive (Equation 1).
- **Horizontal:** Multiple stateless KDL instances can serve the same set of lakes, sharing the PostgreSQL catalog and S3 storage. Distributed locks ensure write consistency across instances. Read queries scale linearly with the number of instances.

10.4 Comparison with Alternatives

Table 3 positions KDL against alternative systems on dimensions critical to AI agent workloads.

Table 3: Feature comparison for AI agent workloads.

Capability	KDL	Delta	Iceberg	Hudi
Sub-second queries	Yes	No	No	No
Native time travel	Yes	Yes	Yes	Yes
Embedded (no JVM)	Yes	No	No	No
Auto error recovery	Yes	No	No	No
Column masking	Yes	No	No	No
Write buffering	Yes	No	No	Yes
OpenLineage native	Yes	No	Partial	No
Setup complexity	Low	High	High	High

11 Related Work

Delta Lake [1] pioneered the lakehouse concept by adding ACID transactions and schema enforcement to data lakes built on Apache Spark. Its JSON-based transaction log provides time travel and audit capabilities, but the reliance on Spark for query execution introduces latency and operational complexity that make it unsuitable for interactive AI agent workloads.

Apache Iceberg [2] introduced a more engine-agnostic table format with snapshot isolation, hidden partitioning, and schema evolution. Its manifest-based metadata structure offers better query planning than Delta Lake's log replay, but still assumes batch-oriented query engines and requires significant operational infrastructure.

Apache Hudi [3] focuses on incremental data processing with support for upserts and incremental queries. Its merge-on-read table type is well-suited for streaming ingestion, but read performance can degrade under high write frequencies due to the need to merge base files with log files at query time.

LakeFS [4] provides Git-like version control for data lakes by implementing a virtual file system layer over object storage. While LakeFS offers powerful branching and merging semantics, it operates at the file level rather than the row level, and does not include a query engine—it must be paired with an external system like Spark or DuckDB.

Nessie [5] takes a similar approach to LakeFS but operates at the Iceberg catalog level, providing Git-like branching for Iceberg tables. Like LakeFS, Nessie is a metadata management layer that requires an external query engine.

KDL differs from all of these systems in its tight integration of versioning, query execution, and caching within a single deployable service. By embedding DuckDB as the query engine and using PostgreSQL as the catalog, KDL eliminates the operational complexity of distributed query engines while providing comparable functionality for the dataset sizes typical of enterprise AI agent workloads (millions to low billions of rows).

12 Conclusion and Future Work

We have presented KDL, a version-native data lake architecture designed for the unique demands of AI-driven enterprise analytics. By combining a PostgreSQL-backed catalog, an embedded DuckDB query engine, and immutable Parquet storage on S3, KDL delivers sub-second analytical queries with automatic versioning, time travel, and comprehensive lineage tracking—without the operational overhead of distributed data infrastructure.

KDL's design reflects a fundamental insight: AI agent workloads require a data platform that is simultaneously interactive (sub-second latency), auditable (complete version history), and resilient (automatic error recovery, circuit breakers, graceful degradation). Traditional data lake architectures, designed for batch ETL pipelines, fail to satisfy all three requirements simultaneously.

Several directions remain for future work:

- **Materialized views with incremental maintenance.** Pre-computing common aggregations and maintaining them incrementally as new data arrives would further reduce query latency for frequently accessed analytical patterns.
- **Federated query execution.** Extending KDL to push down predicates and projections to external data sources via the MCP protocol, reducing data movement for queries that span internal and external data.
- **Snapshot garbage collection.** Implementing configurable retention policies with efficient snapshot compaction to manage storage costs for long-running deployments.
- **Columnar statistics and adaptive indexing.** Collecting min/max statistics per Parquet row group and using them to accelerate predicate evaluation without requiring explicit index creation.
- **Multi-region replication.** Replicating the PostgreSQL catalog and S3 data across regions for disaster recovery and geo-distributed read performance.

References

[1] M. Armbrust, T. Das, L. Sun, et al., "Delta Lake: High-performance ACID table storage over cloud object stores," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, 2020.

[2] R. Blue, J. Brodie, O. O'Malley, et al., "Apache Iceberg: An open table format for analytic datasets," *Apache Software Foundation*, 2022.

[3] V. Jain, N. Agrawal, L. Li, et al., "Apache Hudi: Ingesting and managing data lakes over cloud storage systems," *Proc. ACM SIGMOD*, pp. 2935–2938, 2020.

[4] O. Avinatan, E. Rosenfeld, and G. Sivan, "LakeFS: A scalable data versioning system for data lakes," 2021. [Online]. Available: https://lakefs.io

[5] Dremio, "Project Nessie: Transactional catalog for data lakes," 2022. [Online]. Available: https://projectnessie.org

[6] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," *Proc. ACM SIGMOD*, pp. 1991–1994, 2019.

[7] J. Wills, M. Collado, et al., "OpenLineage: An open standard for data lineage," 2022. [Online]. Available: https://openlineage.io

[8] M. Nygard, "Release It! Design and deploy production-ready software," *Pragmatic Bookshelf*, 2007.

[9] Apache Software Foundation, "Apache Parquet: A columnar storage format," 2013. [Online]. Available: https://parquet.apache.org