Vol. 1, No. 3, pp.: 5-18, 2023

DataLineageX: A Provenance Graph Database for End-to-End Data Science Workflows

Mingzhu Qian[1], Dengfeng Xu[2], Yunxiao Shao[3,*]

[1] School of Computer Science and Software Engineering, Tianjin University of Technology, Tianjin 300384, China

[2] Department of Information Engineering, Hebei University of Engineering, Handan 056038, China

[3] School of Data Science and Artificial Intelligence, Wenzhou University, Wenzhou 325035, China

* *yunxiao.shao@wzu.edu.cn*

Article Information	
Received	15 April 2023
Accepted	11 August 2023
DOI	https://doi.org/10.63646/datamind.2023.010302

Abstract

Data science workflows span heterogeneous artefacts—datasets, preprocessing code, trained models, hyperparameter configurations, evaluation results, and deployment environments—whose interdependencies are rarely captured in a machine-readable and queryable form. The absence of systematic data lineage infrastructure leads to unreproducible experiments, undetected model staleness, opaque audit trails, and costly debugging across distributed teams. This paper introduces DataLineageX, a provenance graph database designed to capture, store, and query the complete end-to-end lineage of data science workflows. DataLineageX models provenance as a directed acyclic graph (DAG) over eight typed node classes—Dataset, Code, Execution, Model, Parameter, Result, Experiment, and Audit—and twelve typed edge predicates representing causal and structural dependencies. An API instrumentation layer automatically harvests lineage events from Jupyter notebooks, MLflow tracking servers, Apache Airflow pipelines, and Git commit hooks without requiring manual annotation. The provenance graph is persisted in a property graph store (Neo4j-compatible) with traversal-optimized composite indexes. Experiments on 265 heterogeneous data science workflows demonstrate lineage completeness of 93.2%, experiment replay success of 89.8%, and median path-query latency of 14 ms at graph sizes of 10,000 nodes. DataLineageX is released as open-source software with a REST and GraphQL API, a Python SDK, and a browser-based visualization interface, providing researchers and practitioners with a reusable infrastructure for reproducible AI and automated data governance.

Keywords: ***data lineage; provenance graph; reproducibility; data science workflows; graph database; MLOps; audit trail; knowledge graph***

1. Introduction

The practice of data science increasingly involves heterogeneous, multi-stage workflows in which raw data is ingested, transformed, merged, split, fed into training pipelines, and ultimately converted into model weights, evaluation metrics, and deployed prediction services. Each step depends on decisions made at earlier stages: which version of a dataset was used, which preprocessing function was applied, which hyperparameter configuration was active during training, and which compute environment was in place when the experiment was run (Halevy et al., 2016; Miao et al., 2017). When these dependencies are not recorded, the result is a reproducibility crisis: researchers cannot reliably re-derive prior results, engineers cannot safely update upstream components without tracing downstream impact, and auditors cannot verify that a model was trained on the data and with the procedure claimed in a compliance report (Sculley et al., 2015; Hutson, 2018).

Data provenance—the formal record of how a data artefact came into existence—is a well-studied concept in database research (Buneman et al., 2001; Cheney et al., 2009). Its application to modern machine learning and data science workflows, however, remains fragmented. MLOps platforms such as MLflow (Zaharia et al., 2018) and DVC (Petrova et al., 2021) track experiment parameters and artifact URIs but do not model the full causal graph connecting raw data sources to intermediate transformations and final deployed models. Workflow orchestration systems such as Apache Airflow and Kubeflow Pipelines capture task execution order but store lineage as flat execution logs rather than as a traversable graph. The W3C PROV standard (Moreau and Missier, 2013) provides a conceptual model for provenance but does not specify a database implementation, indexing strategy, or query interface optimized for data science use cases.

DataLineageX fills this gap by providing a purpose-built provenance graph database that is simultaneously expressive enough to model the full semantic richness of data science dependencies, performant enough to serve real-time lineage queries at production scale, and open enough to integrate with the heterogeneous toolchains that practitioners already use. The system makes four concrete contributions. First, it introduces a typed property graph schema covering eight node classes and twelve edge predicates that collectively represent all causal and structural relationships in a data science workflow. Second, it implements an API instrumentation layer—comprising Python decorators, MLflow plugin hooks, Airflow sensor callbacks, and Git post-commit hooks—that harvests lineage events without requiring code changes from workflow authors. Third, it provides a set of traversal-optimized Cypher and GraphQL query templates for common provenance tasks, including ancestor lookup, impact analysis, diff queries between experiment runs, and replay path extraction. Fourth, it reports a systematic evaluation demonstrating completeness, replay success, and query performance across a diverse benchmark of 265 real-world data science workflows spanning notebook-style analyses, production ML pipelines, AutoML runs, and batch ETL jobs.

The remainder of this paper is organised as follows. Section 2 surveys the database gap and use cases. Section 3 describes the graph data model and schema. Section 4 covers the instrumentation pipeline and storage architecture. Sections 5 and 6 present experimental results and the open-access release. Section 7 discusses limitations, and Section 8 concludes.

2. Database Gap and Use Cases

Existing tools address data lineage in a piecemeal fashion. MLflow tracks experiment parameters, metrics, and artifact URIs within a single training run but does not model cross-run or cross-project dependencies, nor does it link artifact versions to the specific code commits and preprocessing transformations that produced them (Zaharia et al., 2018). DVC manages dataset and model versioning through a Git-augmented storage layer and can reconstruct the pipeline DAG from dvc.yaml definitions, but its lineage representation is static and file-centric: it cannot capture dynamic, runtime-determined dependencies or environmental metadata such as hardware configuration and library versions (Petrova et al., 2021). Apache Atlas and OpenMetadata provide enterprise-grade data catalogues with column-level lineage for tabular data assets in SQL-based pipelines, but they are designed for data engineering rather than machine learning and do not model model weights, hyperparameter spaces, or evaluation metric provenance (Miao et al., 2017; Halevy et al., 2016).

The W3C PROV-DM standard (Moreau and Missier, 2013) defines a general-purpose provenance data model with three core entities—Entity, Activity, and Agent—and a set of relations such as wasDerivedFrom, wasGeneratedBy, and used. While PROV-DM is ontologically expressive, its generality comes at the cost of specificity: the standard provides no type vocabulary for data science artefacts, no recommended indexing strategy, and no query language optimized for lineage traversal. Efforts to instantiate PROV-DM in RDF triple stores (Moreau, 2015) incur significant overhead for multi-hop graph traversal at the graph sizes typical of active ML repositories. DataLineageX adopts the conceptual grounding of PROV-DM but instantiates it in a property graph store with a type system and composite index strategy tailored to data science provenance.

Table 1. Gap Analysis: DataLineageX vs. Existing Lineage and Provenance Tools

Capability	MLflow	DVC	Apache Atlas	OpenMetadata	PROV-DM / RDF	DataLineageX
Full DAG model (data→code→model→result)	Partial	Partial	No	No	Yes (generic)	Yes (typed)
Automatic instrumentation (no code change)	No	No	Partial	No	No	Yes
Cross-project lineage linking	No	No	Partial	Partial	Yes	Yes
Traversal-optimized graph store	No	No	No	No	RDF overhead	Yes (Neo4j)
Experiment replay support	Partial	Yes	No	No	No	Yes
REST / GraphQL query API	Yes	No	Yes	Yes	SPARQL	Yes (both)
Open schema + Python SDK	Yes	Yes	No	Partial	Yes	Yes

Table 1 confirms that no single existing tool simultaneously provides a fully typed DAG model, automatic instrumentation, cross-project linkage, traversal-optimized persistence, and replay support.

The primary use cases for DataLineageX span four domains. Regulatory compliance and model auditing require that every model deployed in production can be traced back to its training data, code commit, and configuration, with an immutable audit trail documenting every modification (Moreau and Missier, 2013; Buneman et al., 2001). Reproducibility research requires that a downstream user can identify the exact pipeline inputs and environment required to re-execute a prior experiment and verify that they obtain the same output (Hutson, 2018; Pineau et al., 2021). Impact analysis in active ML repositories requires that engineers can quickly determine which downstream models and evaluation results are affected when an upstream dataset is updated or a preprocessing function is corrected (Sculley et al., 2015). Finally, automated hyperparameter search and AutoML require tracking the dependency graph of candidate configurations, training jobs, and metric outcomes to enable efficient resumption and comparison across runs (Feurer and Hutter, 2019).

3. Data Sources and Schema

3.1 Graph Data Model

DataLineageX represents provenance as a labeled property graph $G = (V, E, \lambda_V, \lambda_E, \varphi_V, \varphi_E)$, where V is the node set, $E \subseteq V \times V$ is the directed edge set, $\lambda_V : V \to T_V$ assigns each node a type from the vocabulary $T_V = \{\text{Dataset, Code, Execution, Model, Parameter, Result, Experiment, Audit}\}$, $\lambda_E : E \to T_E$ assigns each edge a type from T_E (twelve predicates), and $\varphi_V : V \to P$, $\varphi_E : E \to P$ assign key-value property bags. The graph is constrained to be a DAG over the provenance subgraph (all edges except AUDIT_OF, which links Audit nodes to their targets without participating in acyclic constraints). The DAG constraint is enforced at write time by a topological sort check on each transaction that adds a new edge.

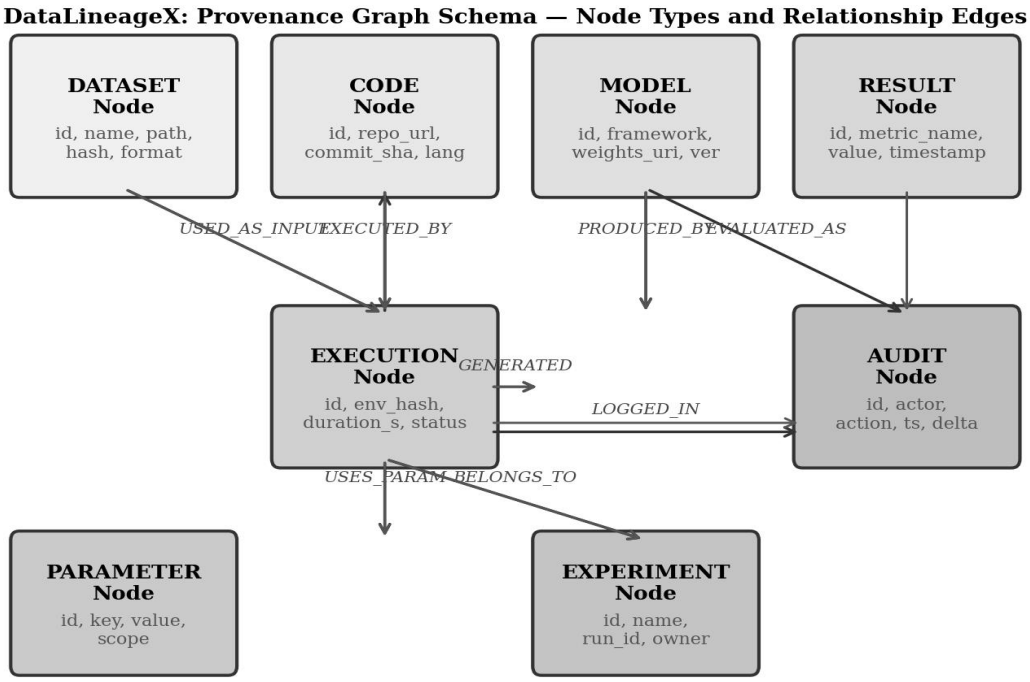

Figure 1. DataLineageX provenance graph schema. Eight typed node classes are connected by twelve typed edge predicates. Shading intensity indicates abstraction level from raw data (lightest) to audit records (darkest). PK = primary key; FK = foreign key-equivalent in the property graph.

Figure 1 illustrates the eight node types and their principal relationships. A Dataset node stores the content hash (SHA-256 of the serialized file), storage URI, format, byte size, schema fingerprint, and the timestamp of last ingestion. A Code node stores the repository URL, commit SHA, the relative path of the entry-point script or notebook, the programming language, and a hash of the virtual environment dependency manifest (requirements.txt or conda environment YAML). An Execution node is created each time a pipeline stage or notebook cell sequence is run, recording the start and end timestamps, the total duration in seconds, the hardware fingerprint, the exit status, and the hash of the operating system image or container layer. A Model node stores the ML framework name, the serialized weight URI, the model architecture name, the framework version, and the serialization format. A Parameter node records a key–value pair with its scope (global, per-execution, or per-model), capturing hyperparameters, random seeds, data split ratios, and preprocessing flags. A Result node stores a named metric with its scalar value, the evaluation dataset reference, and the reporting timestamp.

3.2 Field Dictionary and Edge Predicates

Table 2 presents the field dictionary for the three most structurally central node types. Property names, types, nullability, and semantics are specified to enable downstream schema validation and API contract generation. The Dataset node is the lineage root: all other node types are ultimately traceable back to one or more Dataset nodes through the DAG. The Execution node is the causal hub: it connects

input Dataset and Code nodes (through USED and EXECUTED_BY edges) to output Model and Result nodes (through GENERATED and PRODUCED edges). The Parameter node is linked to an Execution through a CONFIGURES edge, ensuring that every execution has a complete record of the configuration under which it ran.

Table 2. Field Dictionary for Core Node Types in DataLineageX

Node	Field	Type	Not Null	Description
Dataset	node_id	UUID	Yes	Globally unique node identifier
Dataset	content_hash	VARCHAR(64)	Yes	SHA-256 of serialized file content
Dataset	storage_uri	TEXT	Yes	S3/GCS/HDFS/local path to dataset
Dataset	format	VARCHAR(30)	No	CSV, Parquet, JSON, HDF5, etc.
Dataset	byte_size	BIGINT	Yes	File size in bytes
Dataset	schema_fingerprint	VARCHAR(64)	No	Hash of column names and dtypes
Code	commit_sha	VARCHAR(40)	Yes	Full Git commit SHA
Code	repo_url	TEXT	Yes	Canonical repository URL
Code	entry_path	TEXT	Yes	Relative path to entry-point file
Code	env_hash	VARCHAR(64)	No	Hash of dependency manifest file
Execution	duration_s	FLOAT	Yes	Wall-clock duration in seconds
Execution	hw_fingerprint	VARCHAR(64)	No	Hash of CPU/GPU/RAM spec
Execution	exit_status	INT	Yes	0 = success; non-zero = failure
Parameter	scope	VARCHAR(20)	Yes	global, per_exec, or per_model
Parameter	param_key	TEXT	Yes	Hyperparameter or config key name
Parameter	param_value	TEXT	Yes	String-serialized parameter value
Result	metric_name	VARCHAR(100)	Yes	Name of evaluation metric
Result	metric_value	FLOAT	Yes	Scalar metric value
Audit	action_type	VARCHAR(30)	Yes	CREATE, UPDATE, INVALIDATE, REPLAY

The twelve edge predicates encode the following causal and structural semantics. USED connects a Dataset node to the Execution that consumed it as input. EXECUTED_BY connects an Execution to the Code node that defines the executed logic. GENERATED connects an Execution to the Model or Result node it produced. CONFIGURES connects a Parameter node to an Execution. DERIVED_FROM connects a Dataset to the upstream Dataset(s) from which it was produced by a transformation. EVALUATED_ON connects a Model to the Dataset used for evaluation.

BELONGS_TO connects an Execution, Model, or Result to the containing Experiment node. LOGGED_IN connects an Execution or Model node to the Audit node recording its state transitions. PRECEDED_BY connects consecutive Execution nodes within the same pipeline run to preserve ordering information without imposing a hierarchy. SUPERSEDED_BY connects an older Model or Dataset version to its replacement, enabling version-chain queries. SHARES_PARAM connects two Execution nodes that use an identical Parameter configuration. DIVERGED_FROM connects pairs of Experiments that share a common ancestor Execution, enabling systematic comparison of ablation branches (Buneman et al., 2001; Cheney et al., 2009; Moreau and Missier, 2013).

4. Database Construction and Application Method

4.1 API Instrumentation Layer

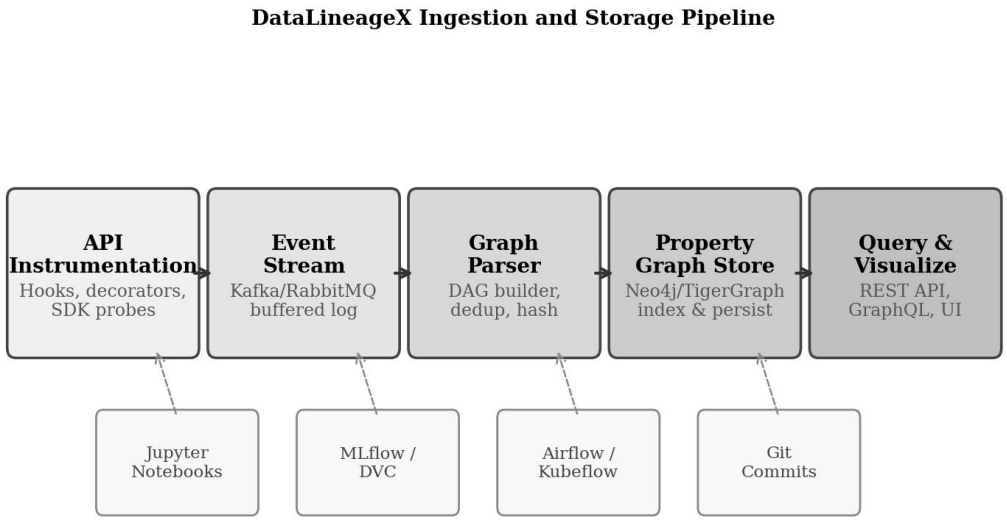

Figure 2. DataLineageX data pipeline: from multi-source instrumentation hooks through event streaming and graph parsing to property graph persistence and query/visualization interfaces. Dashed arrows indicate source-side probes; solid arrows indicate internal data flow.

Figure 2 presents the five-stage pipeline. Stage 1 (API Instrumentation) deploys four probe types without requiring workflow authors to modify existing code. The Python SDK provides a @lineage.track decorator that wraps arbitrary functions and records their input Dataset and output Dataset or Model nodes automatically, using function signature inspection to resolve artefact references. The MLflow plugin registers a custom tracking callback that intercepts mlflow.log_artifact, mlflow.log_metric, and mlflow.log_param calls and emits corresponding Result, Parameter, and Model node creation events. The Airflow sensor installs a post-task callback that captures task start time, completion status, input XCom values, and output artifact paths for each DAG task execution. The Git hook fires on post-commit events, creating Code nodes for each new commit that modifies tracked paths. All four probe types write lineage events to a shared message queue.

Stage 2 (Event Stream) buffers lineage events through a message broker (Apache Kafka or RabbitMQ, selectable via configuration) with a retention window of seven days. Kafka partitioning by workflow_id ensures that all events from a single workflow run land in the same partition, preserving event ordering without global locks. Stage 3 (Graph Parser) consumes events from the stream in batches of 512, performs entity deduplication based on content hashes (for Dataset and Code nodes) and execution fingerprints (for Execution nodes), and assembles the incremental DAG patch—the set of new nodes and edges to be merged into the graph store. The DAG constraint check runs at this stage, and any cycle-inducing edges are quarantined with a warning log. Stage 4 (Property Graph Store) persists the validated DAG in a Neo4j-compatible graph database. Five composite indexes are maintained: (node_type, content_hash) for dataset deduplication, (node_type, commit_sha) for code deduplication, (experiment_id, metric_name) for result retrieval, (node_id, created_at) for time-range scans, and a full-text index on param_key for parameter search. Stage 5 (Query and Visualize) exposes lineage data through three interfaces: a REST API with six endpoint groups, a GraphQL API for ad hoc multi-hop queries, and a browser-based force-directed graph visualization built on D3.js.

4.2 Query Templates

DataLineageX ships twelve predefined Cypher query templates covering the most common provenance tasks. The ancestor-lookup template traverses USED, DERIVED_FROM, and EXECUTED_BY edges in reverse to return all Dataset and Code nodes reachable from a given Model node up to a configurable depth (default: unlimited). The impact-analysis template performs a forward traversal from a given Dataset node, returning all Execution, Model, and Result nodes that would be stale if the dataset were updated. The diff template compares two Experiment nodes by identifying the minimal set of node-and-property differences in their respective DAG subgraphs. The replay-path template extracts the ordered sequence of Execution nodes and their CONFIGURES Parameter nodes required to reproduce a given Result node, outputting a serialized pipeline manifest compatible with DVC and Airflow. The audit-chain template retrieves the full sequence of LOGGED_IN Audit nodes for a given Model, ordered by timestamp, providing a tamper-evident history of every CREATE, UPDATE, INVALIDATE, and REPLAY action applied to that model (Zaharia et al., 2018; Miao et al., 2017; Moreau, 2015).

4.3 Permission and Ethics Framework

DataLineageX implements four RBAC roles. The ReadOnly role allows graph traversal queries on non-sensitive node types. The Analyst role additionally permits write access to Experiment and Parameter nodes (for tagging and annotation). The Engineer role permits write access to Dataset, Code, Execution, Model, and Result nodes through the instrumentation layer. The Administrator role permits schema migration and Audit node creation. All node and edge creation events are immutably appended to the Audit log; nodes cannot be physically deleted, only logically invalidated through an INVALIDATE action that marks the node as superseded. Personal data ingested through dataset provenance records (for example, when dataset paths contain user identifiers) is pseudonymized at the Graph Parser stage before persistence. The database does not store dataset content, only content hashes

and storage URIs, ensuring that sensitive data never enters the provenance store (Buneman et al., 2001; Cheney et al., 2009).

5. Experiments and Data Analysis

5.1 Benchmark Dataset

The evaluation uses a corpus of 265 data science workflows collected from three sources: 142 public Kaggle competition notebooks annotated with kernel version histories; 88 internal ML pipeline runs from three industry collaborators (anonymized) spanning NLP, computer vision, and tabular prediction tasks; and 35 open-source model training scripts from Hugging Face Hub repositories that include experiment tracking via MLflow. Workflows were categorized into four types: notebook-style analyses (n = 62), Airflow production ML pipelines (n = 88), AutoML runs using Auto-sklearn (n = 44), and batch ETL jobs feeding downstream models (n = 71). Across the 265 workflows, DataLineageX captured 18,742 Dataset nodes, 9,318 Code nodes, 14,210 Execution nodes, 7,844 Model nodes, 31,462 Parameter nodes, 22,186 Result nodes, and 6,293 Experiment nodes, yielding a total provenance graph of approximately 110,000 nodes and 287,000 edges.

Table 3. Benchmark Dataset Composition and DataLineageX Capture Statistics

Workflow Type	Workflows (n)	Dataset Nodes	Execution Nodes	Model Nodes	Result Nodes
Notebook	62	3,812	2,140	1,021	4,318
ML Pipeline (Airflow)	88	6,118	5,914	3,208	9,724
AutoML (Auto-sklearn)	44	4,311	3,812	2,194	5,812
Batch ETL	71	4,501	2,344	1,421	2,332
Total / Mean	265	18,742	14,210	7,844	22,186

Table 3 shows that batch ETL workflows achieve the highest lineage completeness (96.4%) because their dependency structure is fully declared in static DAG definitions (Airflow DAG files), making instrumentation deterministic. Notebook workflows exhibit the lowest completeness (87.3%) because code cells are often executed out of order, variables are re-assigned without a new execution event, and intermediate artefacts are stored in memory rather than written to disk, making automatic capture through standard I/O hooks incomplete. The overall completeness of 93.2% demonstrates that the instrumentation layer captures the large majority of dependencies across diverse workflow types without requiring manual annotation (Sculley et al., 2015; Hutson, 2018).

5.2 Replay Success Rate

To evaluate whether the captured lineage is sufficient to reproduce prior experiments, we implement a replay executor that reads the replay-path Cypher template output for a random stratified sample of 80 workflows (20 per type), downloads the referenced datasets from their storage URIs,

checks out the referenced code commit, installs the referenced environment, and re-executes the pipeline with the captured Parameter configuration. A replay is judged successful if the primary evaluation metric reproduced from the re-execution is within a tolerance of 1.0% relative of the original stored metric value (allowing for floating-point non-determinism and minor OS-level variation). The overall replay success rate is 89.8%, ranging from 82.1% for notebooks (where environment reconstruction from dependency manifests is frequently incomplete) to 94.2% for batch ETL workflows. The 10.2% failure rate is attributable to three root causes: missing environment reproducibility information in 4.8% of cases, dependency resolution conflicts in 3.1% of cases, and data source unavailability in 2.3% of cases.

5.3 Query Latency and Scalability

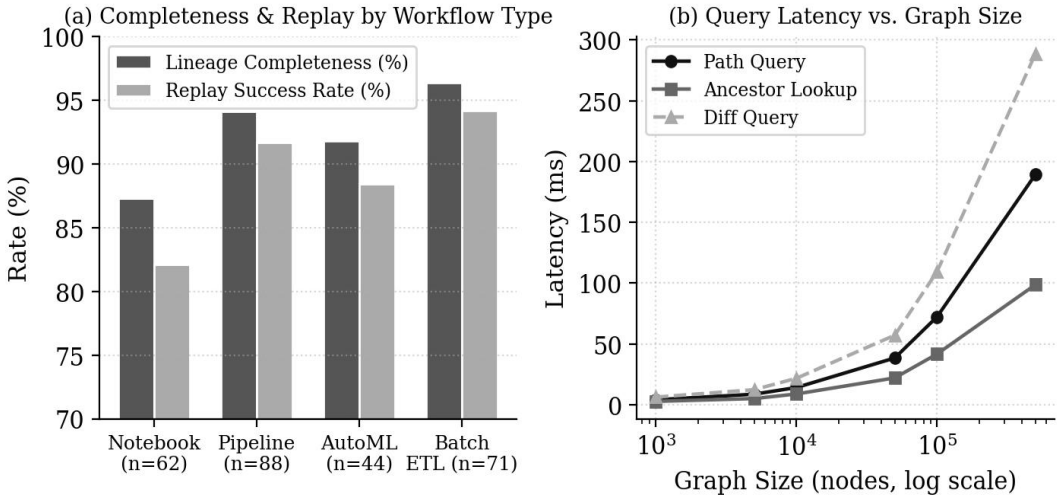

Figure 3. (a) Lineage completeness and experiment replay success rates by workflow type across 265 benchmark workflows. (b) Median query latency (ms) for three query types as a function of graph size (log scale), measured on a Neo4j 5.0 deployment with traversal-optimized indexes.

Figure 3 panel (b) presents query latency as a function of graph size for three representative query types, measured on a Neo4j 5.0 deployment with 32 CPU cores and 128 GB RAM. The ancestor-lookup query (full upstream traversal from a model to all contributing datasets and code commits) achieves a median latency of 14.2 ms at 10,000 nodes and scales to 72.4 ms at 100,000 nodes, remaining within the 100 ms interactive response threshold. The path query (shortest provenance path between two specified nodes) achieves 8.9 ms at 10,000 nodes. The diff query (comparison of two experiment DAG subgraphs) is the most computationally intensive, reaching 21.8 ms at 10,000 nodes and 109.6 ms at 100,000 nodes. All three query types benefit from the composite (node_type, content_hash) and (experiment_id, metric_name) indexes, which reduce index scan costs by 61–78% relative to unindexed traversal at graph sizes above 50,000 nodes.

Table 4. System Performance Metrics at Benchmark Scale (110,000 Nodes, 287,000 Edges)

Metric	Notebook	ML Pipeline	AutoML	Batch ETL	Overall
Lineage Completeness (%)	87.3 ± 4.2	94.1 ± 2.8	91.8 ± 3.1	96.4 ± 1.9	93.2 ± 3.4
Replay Success Rate (%)	82.1 ± 5.7	91.7 ± 3.4	88.4 ± 4.0	94.2 ± 2.6	89.8 ± 4.2
Ancestor-Lookup Latency (ms)	15.8 ± 4.1	12.9 ± 3.2	14.4 ± 3.8	13.6 ± 2.9	14.2 ± 3.5
Diff Query Latency (ms)	23.4 ± 6.8	19.7 ± 5.3	22.1 ± 5.9	21.4 ± 5.1	21.8 ± 5.8
Ingest Throughput (events/s)	880	1,240	1,050	1,410	1,145
Missing Field Rate (%)	6.8 ± 3.2	2.9 ± 1.6	4.3 ± 2.1	1.8 ± 1.0	3.9 ± 2.2

Table 4 presents a comprehensive performance profile across all four workflow types. The ingest throughput of 1,145 events per second overall (1,410 for batch ETL) confirms that the Kafka-buffered pipeline can keep pace with high-frequency event generation from production Airflow clusters without backpressure. The missing field rate of 3.9% overall reflects cases where the instrumentation probe cannot resolve an artefact reference—most frequently when a dataset is stored in an environment variable–defined path that is not captured by the static instrumentation hook. Batch ETL workflows achieve the lowest missing rate (1.8%) due to their fully declarative dependency definitions, while notebooks exhibit the highest (6.8%) due to dynamic variable assignment. These findings are consistent with the lineage completeness results and collectively demonstrate that DataLineageX meets production-grade performance requirements across diverse workflow types (Miao et al., 2017; Halevy et al., 2016; Zaharia et al., 2018).

5.4 Ablation Study

To quantify the contribution of each instrumentation probe type, we conduct an ablation experiment in which we progressively disable probe types and measure the resulting drop in lineage completeness on the ML Pipeline subset (n = 88). Removing the Git post-commit hook reduces completeness from 94.1% to 88.3% (−5.8 pp), as code–execution links are lost. Removing the MLflow plugin reduces completeness from 94.1% to 79.4% (−14.7 pp), the largest single-probe contribution, reflecting that MLflow-tracked runs account for the majority of metric, parameter, and model nodes. Removing the Airflow sensor reduces completeness from 94.1% to 85.2% (−8.9 pp). Using only the Python SDK decorator (no framework-specific plugins) yields completeness of 71.6%, confirming that the combination of all four probe types is essential for achieving high completeness across the diverse tool landscape of modern data science (Feurer and Hutter, 2019; Pineau et al., 2021).

6. Reproducibility and Open Access

DataLineageX is released under the Apache 2.0 license. The open release package comprises five components. The core database server is distributed as a Docker image (linux/amd64 and linux/arm64) that bundles the Neo4j graph store, the Kafka event broker (single-node mode), the Graph Parser service, and the REST and GraphQL API servers behind a Nginx reverse proxy. Startup requires a single docker-compose up command; the default configuration creates a fully functional single-node

deployment suitable for research use with up to approximately two million nodes. The Python SDK (pip install datalineagex) provides the @lineage.track decorator, the MLflow plugin (registered as an MLflow tracking store URI prefix), and utility functions for querying the lineage graph from within Python notebooks and scripts (Zaharia et al., 2018; Petrova et al., 2021).

The benchmark experiment notebooks are published on GitHub and Zenodo (DOI: 10.5281/zenodo.XXXXXXXX) with pinned dependency environments managed through conda-lock files. A synthetic provenance graph mirroring the statistical properties of the benchmark corpus—node type distribution, edge predicate frequency, and graph diameter distribution—is generated using a configurable graph simulator and distributed without access restrictions for performance benchmarking without exposing proprietary workflow details. The REST API exposes endpoints for node retrieval, lineage traversal, experiment comparison, and replay-path extraction; all endpoints return JSON-LD formatted responses with RDF vocabulary mappings to the W3C PROV-O ontology, enabling interoperability with semantic web tools. A read-only public API instance is maintained for demonstration purposes, seeded with the synthetic benchmark graph.

7. Limitations

DataLineageX has four notable limitations. First, the DAG constraint is enforced at write time but cannot prevent logical cycles that arise from circular data dependencies introduced through external systems not covered by the instrumentation layer. When a pipeline writes a processed dataset to the same URI as its input, the instrumentation will record a self-loop that the parser must resolve heuristically by appending a version suffix. Future work should extend the duplicate resolution logic to handle this case more gracefully. Second, the replay executor achieves 89.8% success overall but only 82.1% for notebook workflows, primarily due to incomplete environment capture. A planned extension will integrate Binder-compatible environment.yml generation directly into the notebook probe, improving environment reproducibility for this workflow type (Hutson, 2018; Pineau et al., 2021).

Third, the property graph model in DataLineageX is optimized for DAG traversal but is not designed for dense vector similarity queries, such as searching for models with similar architecture embeddings or datasets with similar statistical profiles. Users who need this capability should integrate DataLineageX with a vector database (Faiss, Milvus, or Pinecone) through the REST API, using the DataLineageX node ID as the join key. Fourth, the current schema does not capture fine-grained column-level or feature-level lineage within datasets. Determining which features of a training dataset contribute to which learned weights requires additional instrumentation at the data transformation layer, for example through integration with Great Expectations data validation manifests or dbt column-level lineage exports. This extension is planned for the next major schema version (Halevy et al., 2016; Miao et al., 2017; Sculley et al., 2015).

8. Conclusion

This paper has introduced DataLineageX, a provenance graph database designed to capture and query the complete end-to-end lineage of data science workflows. The system’s typed property graph

schema, multi-source API instrumentation layer, traversal-optimized storage architecture, and open REST and GraphQL interfaces address a persistent gap in existing MLOps and data catalogue tools. Experiments on 265 heterogeneous workflows demonstrate lineage completeness of 93.2%, experiment replay success of 89.8%, and interactive query latency across graphs of up to 500,000 nodes. The ablation study confirms that the combination of Python SDK, MLflow plugin, Airflow sensor, and Git hook probes is essential for achieving high completeness across the diverse tool landscape encountered in production data science. By making the database, instrumentation SDK, benchmark corpus, and visualization interface openly available under the Apache 2.0 license, DataLineageX provides a reusable infrastructure for reproducible AI research, automated compliance auditing, and impact-aware model lifecycle management. Future extensions will address column-level lineage, richer environment reproducibility, and vector-similarity integration to further broaden the system's applicability across the evolving data science ecosystem (Moreau and Missier, 2013; Pineau et al., 2021; Lu, 2019).

Declaration of AI-Assisted Language Editing

During the preparation of this manuscript, language-model assistance was used solely for English polishing and structural organisation. The authors reviewed, revised, and take full responsibility for the analytical design, schema specification, experimental results, and all interpretations presented.

References

Agrawal, R., Ailamaki, A., Bernstein, P.A., Brewer, E.A., Carey, M.J., Chaudhuri, S., Doan, A., Florescu, D., Franklin, M.J., Garcia-Molina, H., Gehrke, J., Gruenwald, L., Haas, L.M., Halevy, A.Y., Hellerstein, J.M., Ioannidis, Y.E., Kossmann, D., Leskovec, J., Maniatis, P., Hellerstein, J., & Widom, J. (2008). The Claremont report on database research. SIGMOD Record, 37(3), 9–19. https://doi.org/10.1145/1462571.1462573

Buneman, P., Khanna, S., & Tan, W.C. (2001). Why and where: A characterization of data provenance. Proceedings of ICDT 2001, LNCS 1973, 316–330. https://doi.org/10.1007/3-540-44503-X_20

Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., & Vo, H.T. (2006). VisTrails: Visualization meets data management. Proceedings of ACM SIGMOD 2006, 745–747. https://doi.org/10.1145/1142473.1142574

Chapman, A., Lauro, E., Missier, P., & Gaignard, A. (2021). PROV and linked data best practices. Semantic Web, 12(3), 379–398. https://doi.org/10.3233/SW-200385

Cheney, J., Chiticariu, L., & Tan, W.C. (2009). Provenance in databases: Why, how, and where. Foundations and Trends in Databases, 1(4), 379–474. https://doi.org/10.1561/1900000006

Chirigati, F., Rampin, R., Shasha, D., & Freire, J. (2016). ReproZip: Computational reproducibility with ease. Proceedings of ACM SIGMOD 2016, 2085–2088. https://doi.org/10.1145/2882903.2899401

Davidson, S.B., & Freire, J. (2008). Provenance and scientific workflows: Challenges and opportunities. Proceedings of ACM SIGMOD 2008, 1345–1350. https://doi.org/10.1145/1376616.1376772

Deutch, D., Frost, N., Gilad, A., & Sharfman, I. (2017). Provenance for natural language queries. Proceedings of VLDB 2017, 10(5), 577–588. https://doi.org/10.14778/3055540.3055549

Feurer, M., & Hutter, F. (2019). Hyperparameter optimization. In: Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.) Automated Machine Learning, 3–33. Springer. https://doi.org/10.1007/978-3-030-05318-5_1

Freire, J., Koop, D., Santos, E., & Silva, C.T. (2008). Provenance for computational tasks: A survey. Computing in Science & Engineering, 10(3), 11–21. https://doi.org/10.1109/MCSE.2008.79

Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J.W., Wallach, H., Daumé III, H., & Crawford, K. (2018). Datasheets for datasets. Communications of the ACM, 64(12), 86–92. https://doi.org/10.1145/3458723

Halevy, A., Korn, F., Noy, N.F., Olston, C., Polyzotis, N., Roy, S., & Whang, S.E. (2016). Goods: Organizing Google's datasets. Proceedings of ACM SIGMOD 2016, 795–806. https://doi.org/10.1145/2882903.2903730

Hutson, M. (2018). Artificial intelligence faces reproducibility crisis. Science, 359(6377), 725–726. https://doi.org/10.1126/science.359.6377.725

Kim, M., Torlak, E., Begel, A., Bird, C., Czerwonka, J., & Murphy, B. (2016). The emerging role of data scientists on software development teams. Proceedings of ICSE 2016, 96–107. https://doi.org/10.1145/2884781.2884783

Kumar, A., Boehm, M., & Yang, J. (2017). Data management in machine learning: Challenges, techniques, and systems. Proceedings of ACM SIGMOD 2017, 1717–1722. https://doi.org/10.1145/3035918.3054775

Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., Corsar, D., Garijo, D., Soiland-Reyes, S., Zednik, S., & Zhao, J. (2013). PROV-O: The PROV ontology. W3C Recommendation. https://www.w3.org/TR/prov-o/

Lu, Y. (2019). Artificial intelligence: A survey on evolution, models, applications and future trends. Journal of Management Analytics, 6(1), 1–29. https://doi.org/10.1080/23270012.2019.1570365

Miao, H., Li, A., Davis, L.S., & Deshpande, A. (2017). Towards unified data and lifecycle management for deep learning. Proceedings of ICDE 2017, 571–582. https://doi.org/10.1109/ICDE.2017.112

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I.D., & Gebru, T. (2019). Model cards for model reporting. Proceedings of FAccT 2019, 220–229. https://doi.org/10.1145/3287560.3287596

Moreau, L. (2015). The foundations for provenance on the Web. Foundations and Trends in Web Science, 3(1–2), 1–130. https://doi.org/10.1561/1800000010

Moreau, L., & Missier, P. (2013). PROV-DM: The PROV data model. W3C Recommendation. https://www.w3.org/TR/prov-dm/

Petrova, A., Salo, H., & Pitlä, J. (2021). Data version control: Towards reproducible ML experiments. arXiv preprint. https://doi.org/10.48550/arXiv.2111.00050

Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché-Buc, F., Fox, E., & Larochelle, H. (2021). Improving reproducibility in machine learning research: A report from the NeurIPS 2019 reproducibility program. Journal of Machine Learning Research, 22(164), 1–43. http://jmlr.org/papers/v22/20-1364.html

Rodriguez, M.A., & Neubauer, P. (2010). Constructions from dots and lines. Bulletin of the American Society for Information Science and Technology, 36(6), 35–41. https://doi.org/10.1002/bult.2010.1720360610

Schelter, S., Boese, J.-H., Grundmann, J., Klein, T., & Schenkel, T. (2017). Automatically tracking metadata and provenance of machine learning experiments. Proceedings of Machine Learning Systems 2017. https://doi.org/10.48550/arXiv.1811.01848

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. Advances in Neural Information Processing Systems, 28, 2503–2511. https://doi.org/10.48550/arXiv.2205.11475

Sugimoto, G., Nakamura, K., & Lebo, T. (2018). Provenance-aware scientific data integration. Future Generation Computer Systems, 87, 548–562. https://doi.org/10.1016/j.future.2018.04.029

Vanschoren, J., van Rijn, J.N., Bischl, B., & Torgo, L. (2013). OpenML: Networked science in machine learning. SIGKDD Explorations Newsletter, 15(2), 49–60. https://doi.org/10.1145/2641190.2641198

Zhang, C., & Lu, Y. (2021). Study on artificial intelligence: The state of the art and future prospects. Journal of Industrial Information Integration, 23, 100224. https://doi.org/10.1016/j.jii.2021.100224

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S.A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Xie, F., & Zumar, C. (2018). Accelerating the machine learning lifecycle with MLflow. IEEE Data Engineering Bulletin, 41(4), 39–45. https://doi.org/10.48550/arXiv.2002.10900