The Journal of Open Source Software

Metaxy: Record-Level Feature Metadata Management for Multimodal ML Pipelines

Daniel Gafni [1] **and Georg Heiler** [2,3]

1 Anam, United Kingdom **2** Complexity Science Hub Vienna (CSH), Austria **3** Austrian Supply Chain Intelligence Institute (ASCII), Austria

DOI: 10.21105/joss.10449

Software

- Review
- Repository
- Archive

Editor: Abhishek Tiwari

Reviewers:

- @panagiotisanagnostou
- @thc1006
- @Amorfati123

Submitted: 04 March 2026
Published: 17 June 2026

Summary

Software that processes large datasets often repeats expensive computations when any part of the input data or processing logic changes. Metaxy is about perfecting the art of doing nothing: only compute what changed, save time and money, and accelerate exploration. In machine learning pipelines that handle video, audio, and images, these computations run on Graphics Processing Units (GPUs) that cost 10 to 100 times more per hour than standard processors[1]. A small change to one processing step can trigger unnecessary recomputation of unrelated steps, wasting both time and money.

Metaxy is a Python library that tracks which specific data records need reprocessing after a change, rather than rerunning entire datasets. It builds a dependency graph that connects individual data fields across processing steps. When a researcher modifies one step, Metaxy identifies exactly which records are affected and which can be skipped. This selective approach lets downstream systems avoid redundant GPU work when a change does not affect a record, while preserving complete lineage for reproducibility.

The library treats the *metadata store* (which persists version entries), the *compute engine* (which evaluates feature code over dataframes), and the *orchestrator* (which schedules execution) as pluggable abstractions. Concrete integrations for the widely used Python ecosystem ship with the package (Dagster Labs, 2025; Gorelli & others, 2025; Ibis Project, 2025; Moritz et al., 2018), so that any compatible orchestrator can consume Metaxy's record-level diffs and schedule only the necessary GPU workloads.

Statement of Need

Machine learning practitioners iterate rapidly on feature definitions, model architectures, and training strategies. Reproducibility demands that every experiment maintain a clear record of which data and code produced each result. Traditional pipeline tools force a trade-off: rerun entire pipelines to guarantee correctness, or skip tracking to move quickly.

This trade-off is especially costly for multimodal pipelines. Consider a video processing system that extracts both audio transcripts and face detections. When a researcher improves the audio denoising algorithm, should the system recompute face detections that depend only on video frames? Existing orchestrators operate at table granularity, treating feature tables as atomic units. They cannot distinguish which fields within a record changed, forcing unnecessary recomputation of all downstream steps.

Metaxy resolves this problem through field-level dependency tracking at record granularity. Researchers declare features as Python classes that specify which upstream fields each

[1] AWS EC2 on-demand pricing, 2026-04; the exact ratio depends on the instance families compared, with common CPU-to-GPU comparisons falling in this range and top-end accelerators exceeding it.

computation depends on. The system constructs a directed acyclic graph where nodes represent feature fields and edges represent data flow (Figure 1). For each data record, Metaxy computes version hashes that combine code versions with upstream record versions, propagating changes along graph edges. When resolving incremental updates, the system returns only those records whose upstream dependencies have actually changed.

This approach delivers both rapid experimentation and reproducibility. Researchers modify feature definitions freely, and the system identifies exactly which records require reprocessing. The metadata layer preserves complete lineage, enabling retrospective audits of any experiment. By computing diffs before execution, teams quantify the impact of each change and defer updates that do not justify their computational cost.

The target audience includes ML engineers working with multimodal data, MLOps teams managing production feature pipelines, and research labs operating under compute budgets.

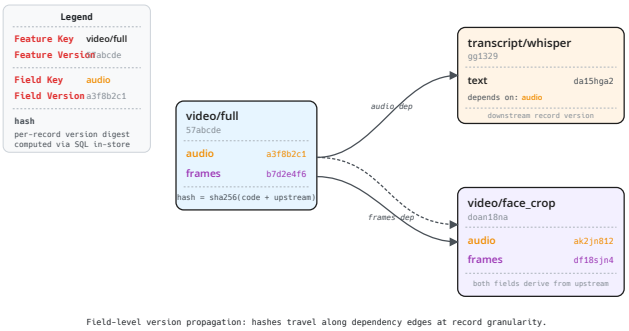

Figure 1: Metaxy tracks separate version hashes for each field of every record. In this example, the `audio` and `frames` fields of a video feature propagate independently through downstream features. A change to the audio processing algorithm only triggers recomputation of audio-dependent downstream fields, leaving frame-dependent fields unchanged.

State of the Field

Several tools address aspects of feature management, but none, to our knowledge, provide field-level dependency tracking at record granularity as a standalone metadata layer. DVC (Kuprieiev et al., 2025) versions datasets at the file level, treating each file as an opaque artifact without tracking individual records or fields within it. Feast (Feast Community, 2025) focuses on feature definition, materialization, and online serving. Recent Feast releases include DAG-based feature computation, but Feast does not expose field-level provenance for propagating version changes and deciding which downstream records require recomputation after an upstream modification. Apache Hamilton (Krawczyk & Gilani, 2024) builds dataflows from Python functions and can report lineage over the resulting DAG, typically at the level of nodes, columns, and dataframe outputs. It does not maintain per-record, per-field provenance for selective downstream invalidation. DataChain (Iterative, Inc., 2025) combines metadata management with a Python-native data processing framework and supports delta processing over new or changed records. However, its incremental model is tied to dataset processing within that framework rather than to a standalone field-level dependency graph that can drive recomputation across multiple compute backends.

Metaxy fills the gap by separating metadata from compute: it tracks field-level dependencies at record granularity and propagates version changes topologically through the dependency graph. This separation allows teams to integrate Metaxy with any orchestrator and compute engine, while retaining precise control over which records require reprocessing.

Software Design

Metaxy represents features as declarative models organized into a directed acyclic graph. Each feature declares its identifier columns, computed fields, and dependencies on upstream feature fields. The system constructs a global dependency graph at initialization, enabling downstream version propagation before any data processing occurs.

After the graph is built, Metaxy separates versioning into graph-level and record-level work. Graph-level feature hashes are recomputed when feature definitions change: each field hash combines the developer-supplied field code version with the upstream graph structure using a fixed `hashlib.sha256` implementation. Developers bump field code versions when they make an algorithmic change. Metaxy makes this boundary explicit because static analysis cannot reliably distinguish a behavior-preserving refactor from a semantic change, and unnecessary invalidation can trigger expensive GPU recomputation.

Record-level hashes are computed inside the metadata store. For each data row, the store combines record identifiers with the upstream record versions for only the fields declared as dependencies; this function is configurable and defaults to `xxhash32` where the backend supports it. The result is an expected provenance signature for each output record before the compute engine runs. Because each signature is tied to declared field dependencies, changes propagate only along affected paths in the graph. When `resolve_update` compares these expected signatures with the append-only signatures already stored, it returns the new or stale records for the orchestrator to schedule. The compute engine then processes only that increment, while records whose relevant upstream dependencies are unchanged are skipped.

The metadata store is append-only: version entries are never overwritten, preserving the lineage needed for retrospective audits. Embedded stores suit prototyping while warehouse- and lakehouse-class stores scale to production workloads, all reached through the same feature API. The compute engine is equally pluggable: a backend-agnostic dataframe abstraction lets users swap implementations without touching feature code.

Design Trade-offs

Metaxy's architecture rests on three deliberate design commitments.

First, the metadata layer is decoupled from the compute engine and orchestrator by design rather than by compromise: Metaxy is a pluggable library that exposes a record-level dependency graph which any orchestrator can consume. The same version graph drives embedded prototyping, warehouse-scale production, and distributed GPU scheduling without changes to feature definitions, reusing the portable dependency-graph abstraction long studied in build-systems and data-lineage research (Buneman et al., 2001; Cui & Widom, 2003; Mokhov et al., 2018).

Second, record-level hashing and the increment diff are pushed into the metadata store via SQL rather than computed client-side. This yields two concrete benefits: the caller never streams full metadata tables out of the store (only the computed increment crosses the wire), and the hash-and-join work runs inside the store, so `resolve_update` stays lightweight enough to invoke from laptops or dashboards while the store scales the compute (Table 1), consistent with the principle of colocating incremental computation with its data (Acar et al., 2006).

Third, Metaxy hashes provenance signatures (code version plus upstream record versions) instead of the raw payloads that content-addressable storage (CAS) would require. CAS is not merely expensive here, it is inapplicable in principle: the increment must be known *before* downstream computation runs, so there is no content to address yet. Hashing the provenance signature lets Metaxy decide staleness without ever materialising the downstream payload, matching the provenance-first view of PROV (Cheney et al., 2009; Moreau et al., 2013); users may still attach content-derived versions after the fact through the user-defined data-version hook, for instance to deduplicate identical outputs.

Performance

Per-record hashing and the downstream diff are executed as SQL inside the metadata store, so their throughput is bounded by the vectorised hash kernels of the backend rather than by Python. The benchmark in `publications/2026-introducing-metaxy/` is implemented as a `pytest-benchmark` suite that exercises two feature graphs: a `simple` graph (one root with two fields feeding a single-field leaf) and a `wide` graph (two roots with four fields each feeding a two-field leaf, so the leaf must join two upstream tables and aggregate eight field versions per record). For each graph and record count N, the benchmark measures `resolve_update` in two phases: the initial materialisation (`resolve_new`) and, after bumping the upstream `audio` provenance for 10% of records, the incremental diff (`resolve_stale`).

To isolate caching effects between rounds, each measurement runs in `pytest-benchmark`'s pedantic mode with a `setup` callback that allocates a fresh DuckDB file in its own temporary directory and reseeds it from scratch. The DuckDB buffer pool, temp-table state, and file-system page cache for the new inode are therefore cold at the start of every round, so no round can inherit state from a previous one. The configuration under test sets `enable_map_datatype = true` in `metaxy.toml`, reflecting the upcoming default; the provenance column is stored and exchanged as a native `Map`. We report ten independent rounds per cell on an Apple M2 Max (64 GiB RAM, Python 3.10.19, DuckDB 1.4.3, `xxhash64`): the median as a robust central estimate, the 25th–75th percentile interquartile range (IQR) as a non-parametric confidence band, and mean $\pm$ standard deviation for readers preferring a parametric view (Table 1). End-to-end wall-clock scales near-linearly with N on both graphs (Table 1, Figure 2); the constant-factor gap between `simple` and `wide` at any given N reflects the additional upstream join and the richer provenance struct rather than super-linear slowdown, confirming that the SQL pushdown strategy absorbs join width well.

Table 1: `resolve_update` wall-clock time on DuckDB for the `simple` and `wide` graphs at N records over 10 pedantic rounds per cell (10% change fraction, fresh DuckDB file per round, `enable_map_datatype = true`). Times are milliseconds; `IQR` is $[q_{25}, q_{75}]$.

scenario	N	new median (ms)	new IQR (ms)	new mean $\pm$ sd (ms)	stale median (ms)	stale IQR (ms)	stale mean $\pm$ sd (ms)
simple	10,000	497.2	[489.6, 546.2]	547 $\pm$ 116	590.8	[558.1, 606.2]	585 $\pm$ 32
simple	100,000	558.8	[536.2, 582.7]	583 $\pm$ 81	711.6	[696.6, 741.6]	801 $\pm$ 296
simple	1,000,000	1,171.9	[1,056.9, 1,370.8]	1,225 $\pm$ 203	1,270.8	[1,243.2, 1,455.3]	1,342 $\pm$ 167
simple	10,000,000	7,797.9	[6,563.7, 8,594.4]	7,811 $\pm$ 1,176	7,133.4	[6,950.9, 7,504.3]	7,285 $\pm$ 586
wide	10,000	763.8	[717.6, 798.9]	774 $\pm$ 79	834.0	[808.1, 855.4]	830 $\pm$ 47
wide	100,000	861.5	[846.6, 882.7]	866 $\pm$ 23	1,097.9	[1,061.1, 1,125.2]	1,089 $\pm$ 40
wide	1,000,000	1,909.6	[1,841.5, 1,956.8]	1,908 $\pm$ 69	2,408.2	[2,279.9, 2,458.5]	2,394 $\pm$ 114
wide	10,000,000	11,870.8	[11,651.2, 12,436.3]	12,070 $\pm$ 597	14,695.4	[14,554.2, 16,532.9]	15,290 $\pm$ 1,052

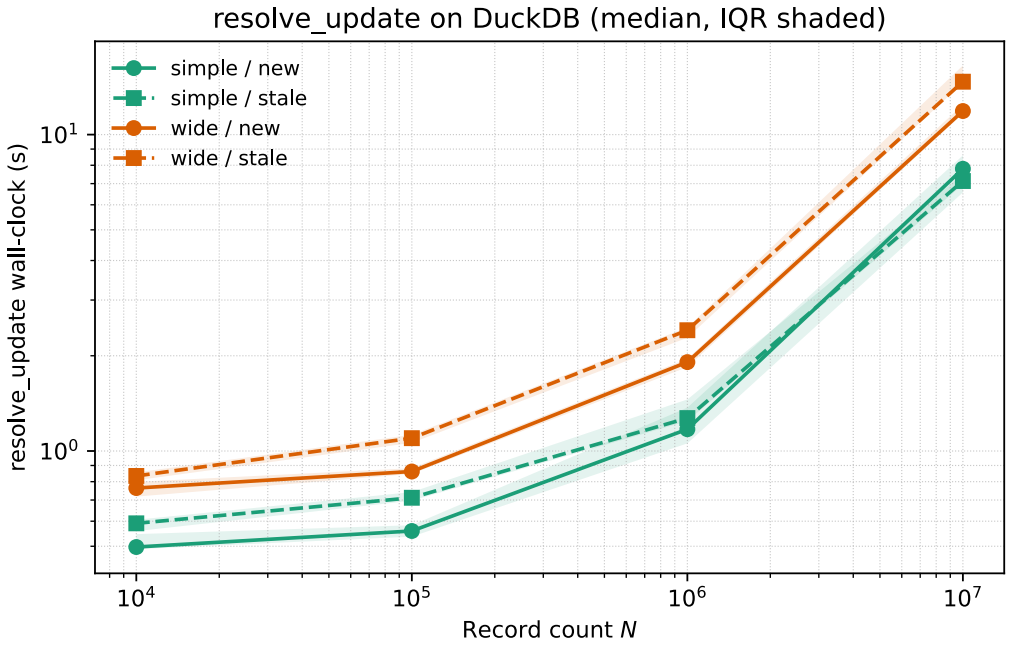

Figure 2: Wall-clock time of `resolve_update` on DuckDB as a function of record count N for both feature graphs, over 10 pedantic rounds per cell with a fresh DuckDB file per round. Lines show the median and the shaded band spans the 25th–75th percentile IQR. Both axes are logarithmic. Near-linear scaling with N is visible on both graphs; the `wide` graph pays a roughly constant-factor penalty for the extra upstream join.

Quality Control

Correctness is validated through automated tests covering three properties. Snapshot testing is used to ensure version computation consistency across Metaxy codebase changes. Incremental update tests verify that the system returns exactly those records whose upstream dependencies changed, neither missing updates nor triggering false positives. Cross-backend tests ensure different metadata store backends produce identical versioning results, matching with the "golden" DuckDB implementation.

Ongoing Research Projects

Example pipelines demonstrate the system's impact on multimodal workflows. A video processing pipeline defines features for audio transcription and face detection with field-level dependencies. When only the audio processing algorithm changes, the system correctly schedules transcription updates for affected records while leaving face detection metadata unchanged. This selective recomputation is the core value proposition: the metadata layer exposes the exact update set, so teams can schedule only affected records instead of maintaining that decision logic manually.

Metaxy has been running in production at Anam on multimodal video pipelines whose training corpora reach the low millions of samples. Before Metaxy, achieving selective recomputation at this scale required manual metadata edits, ad-hoc overrides, and custom per-pipeline bookkeeping; Metaxy standardizes those decisions into a declarative model so that correct, auditable incremental updates become the default path rather than a bespoke engineering effort for each new feature. Combined with the append-only metadata layer introduced above, every experiment remains reproducible, closing a critical gap in machine learning workflows.

The system bridges prototyping and production through vendor-neutral abstractions. Researchers define features once and deploy them across embedded stores on laptops, warehouse-class stores in data centers, or managed cloud stores, without modifying feature code. This portability lowers the barrier to disciplined metadata management, making reproducibility the default rather than an afterthought.

The project welcomes contributions at https://github.com/anam-org/metaxy. Documentation is available at https://docs.metaxy.io.

AI Usage Disclosure

Generative AI tools were used during development for code completion and documentation drafting. All AI-generated content was reviewed and refined by the authors. Large parts of the documentation were written by hand.

Acknowledgements

We thank the maintainers of the open-source projects Metaxy builds upon, and the contributors who tested early releases across heterogeneous hardware. Funding and in-kind support were provided by Anam, the Complexity Science Hub Vienna, and the Austrian Supply Chain Intelligence Institute.

References

Acar, U. A., Blelloch, G. E., & Harper, R. (2006). Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *28*, 990–1034. https://doi.org/10.1145/1186632.1186634

Buneman, P., Khanna, S., & Tan, W.-C. (2001). Why and where: A characterization of data provenance. *Proceedings of the 8th International Conference on Database Theory (ICDT)*, 316–330. https://doi.org/10.1007/3-540-44503-X_20

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

Cui, Y., & Widom, J. (2003). Lineage tracing for general data warehouse transformations. *The VLDB Journal*, *12*(1), 41–58. https://doi.org/10.1007/s00778-002-0083-8

Dagster Labs. (2025). *Dagster: Data orchestration platform*. https://dagster.io.

Feast Community. (2025). *Feast: Feature store for machine learning*. https://feast.dev.

Gorelli, M., & others. (2025). *Narwhals: Write once, run on many DataFrames*. https://narwhals.readthedocs.io.

Ibis Project. (2025). *Ibis: Portable DataFrames for different backends*. https://ibis-project.org.

Iterative, Inc. (2025). *DataChain: AI data warehouse for multimodal data*. https://datachain.ai.

Krawczyk, S., & Gilani, E. (2024). Hamilton: A micro-framework for creating dataflows from Python functions. *Proceedings of the 23rd Python in Science Conference (SciPy 2024)*. https://doi.org/10.25080/gerudo-f2bc6f59-014

Kuprieiev, R., Petrov, D., & others. (2025). *DVC: Data version control*. https://dvc.org.

Mokhov, A., Mitchell, N., & Peyton Jones, S. (2018). Build systems à la carte. *Proceedings of the ACM on Programming Languages*, *2*(ICFP), 79:1–79:29. https://doi.org/10.1145/3236774

Moreau, L., Missier, P., Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles, S., Myers, J., Sahoo, S., & Tilmes, C. (2013). *PROV-DM: The PROV Data Model* [W3C Recommendation]. World Wide Web Consortium (W3C). https://www.w3.org/TR/prov-dm/

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, M., Jordan, M. I., & Stoica, I. (2018). Ray: A distributed framework for emerging AI applications. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://arxiv.org/abs/1712.05889