Metaxy: Field-Level Metadata Management for Incremental Multimodal ML Pipelines

Daniel Gafni
Anam
United Kingdom
danielgafni16@gmail.com

Georg Heiler
Complexity Science Hub Vienna (CSH)
Austria
Austrian Supply Chain Intelligence Institute (ASCII)
Austria
heiler@csh.ac.at

ABSTRACT

Multimodal ML pipelines incur high GPU costs, yet existing orchestrators invalidate entire downstream datasets when any processing step changes—even for unaffected fields. We present **Metaxy**, an open-source Python library providing *field-level dependency tracking at record granularity*. Feature definitions form a directed acyclic graph; for each record, hierarchical version hashes propagate changes along field-level edges. A `resolve_update` operation returns exactly those records that are new, stale, or orphaned, enabling selective recomputation. In production across two organizations processing millions of multimodal samples, Metaxy eliminates redundant GPU work while preserving complete per-record lineage for reproducibility.

CCS CONCEPTS

• **Information systems** → **Data provenance**; • **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

metadata management, incremental processing, feature engineering, multimodal ML, data provenance, reproducibility

1 INTRODUCTION

Modern ML increasingly relies on multimodal data—video, audio, images, and text—processed through GPU-accelerated pipelines. Unlike tabular ETL, where re-execution is cheap, multimodal steps such as speech-to-text transcription or embedding generation can be expensive per record [2]. Data orchestrators like Dagster [6] model pipelines as graphs of *assets* (tables or datasets) and track freshness at the asset level: when a processing step changes, every downstream record is recomputed—even those whose inputs are unchanged.

Consider a video pipeline that extracts audio transcripts and face detections (Figure 1). Improving only the audio denoising algorithm should not trigger recomputation of face detections that depend solely on video frames—yet at table granularity, that recomputation fires anyway. This problem compounds as pipelines grow, and is especially acute for iterative, agent-driven ML development, where rapid experimentation demands that each iteration recomputes only what is strictly necessary.

We address this gap with **Metaxy**,[1] a standalone metadata layer operating at *field* and *record* granularity that enables selective recomputation—both cost-efficient and supportive of rapid experimentation.

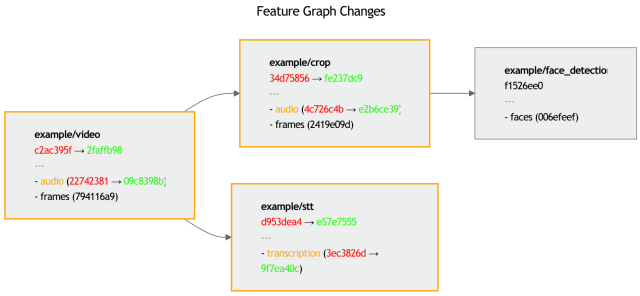

Figure 1: Feature graph after an audio algorithm change. Orange borders mark affected features; gray borders are unchanged. Red/green hashes show old/new field versions. The `audio` change propagates to `crop` and `stt`; `face_detection` (consuming only `frames`) is unaffected.

2 WHY FIELD-LEVEL, PER-RECORD TRACKING IS HARD

Most pipeline tools stop above field × record granularity because finer invalidation requires coordinated choices in the dependency graph, version policy, identifier scheme, and metadata-store execution.

B1. The granularity of the dependency model. Build systems [15], asset orchestrators [6], and feature stores [7] model dependencies between *coarse* units—files, tables, datasets, or features. That granularity makes correctness easy (one timestamp per node) but treats the table as atomic, so any upstream change invalidates the whole downstream node. Tracking dependencies at the level of *individual fields within a feature* requires the metadata layer to carry a vector of per-field provenance per record, not a single timestamp per table.

B2. Inferring semantic change automatically is unsafe. The natural alternative—static analysis of feature code—cannot reliably distinguish a behavior-preserving refactor from a semantically meaningful edit. False negatives silently break downstream correctness; false positives trigger the recomputation we are trying to avoid. Metaxy uses a declarative boundary: each field carries a developer-controlled `code_version` string, and a change is whatever the author marks as a change. This is the same trade-off used in compilers

[1] https://github.com/anam-org/metaxy

and build systems [15]: developers set the boundary, and the system propagates it.

B3. Content-addressable storage does not apply. A natural way to identify computed outputs is by their hashed content (CAS, as in DVC [14] and most artifact stores). For an *incremental* pipeline this does not determine the worklist: the increment must be known *before* the downstream stage runs, so the content does not yet exist. Metaxy hashes the *provenance signature*—field code versions plus upstream record versions—which lets it decide staleness without ever materializing a downstream payload. Users may still attach content-derived versions after the fact via a user-defined data-version hook (for example to deduplicate identical outputs), keeping CAS as an opt-in optimization rather than a requirement.

B4. The metadata itself must be cheap at record granularity. Coarse metadata (one timestamp per table) is naturally tiny. Per-record, per-field metadata over millions of multimodal samples is not: a naive implementation pulls full metadata tables into Python and hashes them client-side. Metaxy pushes record-level hashing and the increment join into the metadata store via SQL using Ibis [11]; only the computed increment crosses the wire. This is the same colocation principle that underpins adaptive incremental computation [1]: keep the diff next to the data that determines it.

Metaxy couples these choices: the field-level dependency graph defines what can change, explicit code versions define when it changes, provenance hashes identify stale records before recomputation, and store-side SQL keeps the per-record diff cheap.

3 DESIGN

3.1 Feature definitions and dependency graph

Feature definitions are registered through a common specification interface.[2] Each feature specifies identifier columns, computed fields with explicit code versions, and field-level dependencies on upstream features via a `FeatureSpec`. At initialization, Metaxy constructs a global DAG whose nodes are feature fields and whose edges encode data flow. This graph enables topological version propagation: when an upstream field changes, only transitively dependent downstream fields are marked stale. The system handles one-to-one, one-to-many, and many-to-one lineage relationships, covering common multimodal patterns such as expanding a video into per-frame crops or aggregating audio segments into a transcript summary.

3.2 Hierarchical versioning

Version computation operates through deterministic hashing at multiple levels: *(1) field code versions*—user-specified strings marking algorithmic changes (B2); *(2) field provenance*—per-field hashes combining current code versions with upstream record field versions; *(3) record provenance*—a sample-level hash derived from the per-field provenance map. Metaxy also exposes user-controlled data versions that default to provenance, allowing applications to encode direct data-version decisions when needed (B3). When `resolve_update` is invoked, Metaxy compares expected provenance (computed from the current graph and upstream data) against stored provenance:

```
inc = store.resolve_update("video/stt")
# inc.new not yet processed
# inc.stale expected provenance changed
# inc.orphaned removed upstream
todo = pl.concat([inc.new, inc.stale])
```

The returned `Increment` contains exactly those records requiring work; unchanged records are skipped entirely.

3.3 Pluggable storage, orchestration, and execution

Metaxy separates metadata from storage, orchestration, and execution. The `MetadataStore` abstraction supports multiple backends (DuckDB for local prototyping, ClickHouse, BigQuery, DuckLake or lakehouse formats for production). Version computation is pushed into the store via SQL where possible (B4), avoiding data transfers; backend-agnostic dataframe interoperability is achieved through Narwhals [9] and Ibis [11].

For orchestration, Metaxy integrates with Dagster assets. The `metaxify` decorator enriches asset definitions with lineage metadata while leaving scheduling decisions to the orchestrator. For execution, Metaxy can be paired with Ray [17] via dedicated datasource and datasink classes. The orchestrator decides *which feature* to compute; Metaxy resolves *which samples actually require it*; the execution engine processes only the delta.

4 EVALUATION

The evaluation has two bounded parts. Section 4.1 gives a topology-only before/after calculation on a small multimodal DAG to show how field dependencies change the recomputation set. This calculation is illustrative; it is not a measurement of production throughput, latency, or cost. Section 4.2 reports `resolve_update` wall-clock from a public DuckDB benchmark suite and compares the added metadata work with coarse asset-level freshness checks.

4.1 Avoided recomputation

The pipeline in Figure 1 extends to a seven-feature DAG with a `Video` root, fields `{audio, frames}`, an audio path (`audio_denoise` → `stt` → `text_embed`), a visual path (`crop` → `face_detection`), and a fusion stage (`video_embed`) that consumes `crop` and `audio_denoise`. There are six *downstream* features (excluding the root). The baseline is the coarse asset/table tracking used before Metaxy: a change to the shared multimodal sample asset reprocesses all six downstream features over all N records. Under Metaxy, only features whose field-level dependencies transitively consume the changed field are reprocessed; record-level selectivity narrows this further when a change affects only a subset of records. Table 1 uses normalized work units: (features reprocessed) × (records reprocessed). It is a structural calculation over the example DAG, not a production measurement. Table 1 summarizes five example changes.

The table separates field-level dependency pruning from record selectivity. Rows 1–3 isolate which downstream features consume

[2] The most common authoring pattern uses Python classes extending a Pydantic base model, but the same graph can also be populated from `metaxy.lock` or directly from the metadata store.

Table 1: Illustrative before/after calculation for the seven-feature example DAG of Figure 1 (six downstream features over N records). *Asset* = invalidate all downstream features over all records; *Metaxy* = recompute only transitively affected (feature, record) pairs. For subset changes, p is the fraction of records actually changed.

Change	Asset	Metaxy
Audio denoiser code bump	$6 \cdot N$	$4 \cdot N$
Crop-resolution code bump	$6 \cdot N$	$3 \cdot N$
STT model swap	$6 \cdot N$	$2 \cdot N$
Refresh subset of records	$6 \cdot N$	$6 \cdot pN$
Audio bump on subset of records	$6 \cdot N$	$4 \cdot pN$

Table 2: Median wall-clock of `resolve_update` on DuckDB across record counts N, in milliseconds (10 rounds per cell, cold cache, fresh DB file per round).

Scenario	N	new (ms)	stale (ms)
`simple`	10^4	497	591
`simple`	10^5	559	712
`simple`	10^6	1,172	1,271
`simple`	10^7	7,798	7,133
`wide`	10^4	764	834
`wide`	10^5	862	1,098
`wide`	10^6	1,910	2,408
`wide`	10^7	11,871	14,695

the changed field; rows 4–5 add record-level scope. Production savings depend on actual stage costs, change frequency, and record distributions.

4.2 Overhead

The benchmark suite reported in [8] measures `resolve_update` on DuckDB using two graphs: a `simple` graph (one root, single-field leaf) and a `wide` graph (two roots with four fields each, two-field leaf) that exercises a wider provenance join. Each cell is ten rounds with a fresh DuckDB file per round (cold buffer pool, cold OS page cache); the configuration enables the native `Map` datatype. Table 2 reports the median wall-clock for the initial materialization (`resolve_new`) and for the incremental diff after bumping a subset of records (`resolve_stale`) on an Apple M2 Max laptop.

End-to-end wall-clock scales near-linearly with N on both graphs, showing that the SQL pushdown strategy absorbs join width well in this benchmark. Coarse asset-level freshness checks are essentially constant time in the number of records because they compare one version per asset. Metaxy's extra benchmark cost is the row-level metadata diff. The overhead is dominated by hash construction and join inside the metadata store, not by Python; the same workload on a warehouse-class backend (ClickHouse, BigQuery) shifts the constant but preserves the scaling pattern.

5 PRODUCTION EXPERIENCE

Metaxy has been in production at Anam[3], a conversational avatar company, since December 2025, processing millions of training samples for a real-time interactive avatar platform powered by a custom video generation model. The pipeline spans audio extraction, speech-to-text, face detection, video cropping, and embedding generation; the most expensive stages are GPU-accelerated. Before Metaxy, changing crop resolution triggered recomputation of all downstream steps, including audio-only stages unaffected by the change. With field-level tracking, only genuinely affected records are reprocessed. The append-only metadata design preserves complete per-record lineage, enabling retrospective audits and experiment reproducibility without additional bookkeeping.

At ASCII (Austrian Supply Chain Intelligence Institute), Metaxy powers incremental document processing for knowledge extraction from large-scale web crawls [10].

The same feature definitions can be reused across local prototyping and production deployments, reducing operational friction.

6 DISCUSSION: LESSONS FOR THE SAO COMMUNITY

The deployments give four lessons for multimodal ML data systems.

Metadata granularity should match cost granularity. A pipeline's most expensive stages dictate the granularity at which invalidation must be precise. For tabular ETL on cheap CPUs, table-level invalidation is fine; rerunning the table is cheaper than the metadata bookkeeping required to avoid it. Once per-record stages become expensive (multimodal GPU work, LLM calls, retrieval over large embeddings), the asymmetry inverts: the cost of *getting the increment wrong* dwarfs the cost of computing a precise increment. The same cost model appears in agentic data systems: cost is paid per record or LLM call, while many code-version edits affect only a subset of fields.

Iteration rate amplifies the value of precise increments. The savings compound with iteration: if a researcher runs k experiments per week and each touches only a subset of fields, asset-granularity invalidation pays the full $N \cdot$(#features) cost k times, while field-granularity pays a structurally smaller cost per iteration. This effect is largest for agentic loops, where many candidate code versions are tried and most are discarded; cheap, precise increments make exploration affordable.

Pipeline shape predicts the gain. The example in Table 1 prunes work because the graph has a clear bifurcation between audio and visual fields, with only a fusion stage downstream that recombines them. Pipelines that are deep and narrow (every stage consumes every upstream field) gain less from field-level tracking; pipelines that fan out by modality, language, or feature family gain more. Practitioners can estimate likely benefit before adopting Metaxy by counting, for a representative recent change, the fraction of downstream features whose declared field dependencies actually consume the changed field.

[3] https://www.anam.ai

Declarative change boundaries should be reviewable. Code-version strings make the boundary between "what counts as a semantic change" explicit and reviewable, much as database migrations or API versioning do. For systems whose code is partly generated, a reviewed version boundary is easier to audit than a heuristic.

7 RELATED WORK

DVC [14] versions datasets at file level without tracking individual records. Asset orchestrators such as Dagster [6] schedule recomputation at asset granularity rather than tracking per-record, per-field provenance. Feast [7] focuses on feature definition, materialization, and online serving and does not expose field-level provenance for propagating version changes. Apache Hamilton [13] reports column-level lineage over Python dataflows but at table granularity. DataChain [12] supports delta processing for multimodal workflows but couples its incremental model to its own framework. Conceptually, Metaxy draws on the why/where provenance distinction [3, 4] and the W3C PROV data model [16], applied to record-level multimodal feature pipelines; the colocation of incremental computation with its data follows the line of work on adaptive computation [1, 5]. Metaxy fills this gap as a *standalone* metadata layer providing field-level tracking at record granularity, decoupled from any specific compute or storage backend.

8 CONCLUSION

Metaxy introduces field-level dependency tracking at record granularity as a standalone metadata layer for multimodal ML pipelines. The example calculation shows how field dependencies reduce the recomputation set in a branched audio/visual DAG, while the DuckDB benchmark shows that record-level diffing can be executed inside the metadata store. The implementation combines a field-level dependency graph, explicit code versions, provenance hashes, and store-side diffing so stale records can be identified before downstream GPU work starts. For the SAO community, the practical rule is to match metadata granularity to cost granularity: workloads that pay per record or per LLM call need increments at the same grain. The library is open-source at https://github.com/anam-org/metaxy.

REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Transactions on Programming Languages and Systems* 28, 6 (2006), 990–1034. https://doi.org/10.1145/1186632.1186634

[2] Anyscale. 2024. GPU vs. CPU cost analysis for ML inference workloads. AWS EC2 on-demand pricing. On-demand cloud pricing data; representative GPU-to-CPU hourly cost ratio of 10–100× depending on instance family.

[3] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT)*. 316–330. https://doi.org/10.1007/3-540-44503-X_20

[4] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474. https://doi.org/10.1561/1900000006

[5] Yingwei Cui and Jennifer Widom. 2003. Lineage Tracing for General Data Warehouse Transformations. *The VLDB Journal* 12, 1 (2003), 41–58. https://doi.org/10.1007/s00778-002-0083-8

[6] Dagster Labs. 2025. Dagster: Data Orchestration Platform. https://dagster.io.

[7] Feast Community. 2025. Feast: Feature Store for Machine Learning. https://feast.dev.

[8] Daniel Gafni and Georg Heiler. 2026. Metaxy: Record-Level Feature Metadata Management for Multimodal ML Pipelines. https://docs.metaxy.io/latest/publications/2026-introducing-metaxy/. Software paper draft.

[9] Marco Gorelli and contributors. 2025. Narwhals: Write Once, Run on Many DataFrames. https://narwhals.readthedocs.io.

[10] Georg Heiler, Hernan Picatto, Maximilian Heß, and Martin Pfister. 2025. dagster-slurm: Reproducible Data Orchestration on HPC. https://github.com/ascii-supply-networks/dagster-slurm.

[11] Ibis Project. 2025. Ibis: Portable DataFrames for Different Backends. https://ibis-project.org.

[12] Iterative, Inc. 2025. DataChain: AI Data Warehouse for Multimodal Data. https://datachain.ai.

[13] Stefan Krawczyk and Elijah Gilani. 2024. Hamilton: A Micro-Framework for Creating Dataflows from Python Functions. In *Proceedings of the 23rd Python in Science Conference (SciPy 2024)*. https://doi.org/10.25080/gerudo-f2bc6f59-014

[14] Ruslan Kuprieiev, Dmitry Petrov, et al. 2025. *DVC: Data Version Control*.

[15] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à la Carte. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 79:1–79:29. https://doi.org/10.1145/3236774

[16] Luc Moreau, Paolo Missier, Khalid Belhajjame, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. 2013. *PROV-DM: The PROV Data Model*. W3C Recommendation. World Wide Web Consortium (W3C). https://www.w3.org/TR/prov-dm/

[17] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melanie Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.