Vol. 2, No3 , pp.73-93: , 2024

DataMindFlow: An Open-Source Orchestration System for Reproducible Data Engineering Pipelines

Daniel R. Okafor[1], Mei-Ling Tan[2,*], Sofia Almeida[1], Rajiv Menon[3]

[1] Data Systems Group, Department of Computer Science, Riverside Institute of Technology, Riverside, CA 92507, USA

[2] School of Computing and Data Science, National Polytechnic University, Singapore 138632, Singapore

[3] Centre for Computational Discovery, Trans-Pacific Research Laboratory, Vancouver, BC V6T 1Z4, Canada

* *mei-ling.tan@npu.edu.sg*

Article Information	
Received	18 April 2024
Accepted	22 August 2024
DOI	https://doi.org/10.63646/datamind.2024.020306

Abstract

Modern data engineering increasingly relies on multi-stage pipelines that ingest, clean, transform, and model heterogeneous data before any analysis can begin. Although general-purpose workflow schedulers have made these pipelines easier to express and operate, they were not designed to guarantee that a pipeline, once executed, can be reproduced exactly on a different machine, at a later date, or by a different person. The result is a persistent reproducibility gap: re-running the same pipeline frequently yields non-identical artifacts because code, data, parameters, and the software environment are not captured as a single coherent unit. This article presents DataMindFlow, an open-source orchestration system that treats reproducibility as a first-class engineering property rather than an afterthought. DataMindFlow compiles a declarative pipeline into a directed acyclic graph of tasks, derives a deterministic content-addressed cache key for every task from the hash of its code, its resolved inputs, its parameters, and a digest of its execution environment, and persists a complete lineage record in a relational metadata store. A list-scheduling executor based on the Heterogeneous Earliest-Finish-Time heuristic places tasks across local and distributed workers, while a memoisation layer skips any task whose cache key already exists, enabling correct incremental recomputation after partial edits. We describe the system architecture, the metadata schema, the cache-key derivation algorithm, and the public application programming interface, and we evaluate the implementation against five widely used baselines on synthetic graphs of up to five thousand tasks and three realistic pipelines drawn from genomics, natural-language processing, and tabular machine learning. DataMindFlow reduces per-task scheduling overhead by between 2.1 and 29 times relative to the baselines, recovers up to 99.2 percent

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

bit-identical artifacts across three independent hosts, and sustains near-linear throughput scaling to thirty-two workers. We argue that deterministic content addressing, environment capture, and durable lineage should be standard components of data engineering infrastructure, and we release the system, its data dictionary, and all evaluation artifacts under a permissive licence.

Keywords: ***Data engineering; workflow orchestration; reproducibility; data lineage; content-addressed storage; pipeline scheduling; AI data infrastructure***

1. Introduction

Data engineering has become the foundation on which empirical computing rests. Before a model can be trained, a dashboard can be drawn, or a scientific claim can be defended, raw inputs must be ingested from disparate sources, cleaned, validated, joined, transformed, and materialised into analysis-ready artifacts. These steps are rarely a single program; they form a pipeline of interdependent tasks whose collective behaviour determines the trustworthiness of everything downstream. As pipelines have grown in length and complexity, the engineering question has shifted. The difficulty is no longer whether a transformation can be expressed, but whether the entire pipeline can be operated reliably and reproduced exactly when the people, machines, and data underneath it change.

Reproducibility is now widely recognised as a structural problem in computational work. A large survey of practising researchers reported that a majority had failed to reproduce another group's results and that many had failed to reproduce their own (Baker, 2016). The methodological literature has responded with principles and checklists for reproducible computational science (Peng, 2011; Sandve et al., 2013; Stodden et al., 2016), and the data-management community has articulated the complementary goal of making data findable, accessible, interoperable, and reusable (Wilkinson et al., 2016). Yet principles alone do not reproduce a pipeline. Reproduction requires that the exact code, the exact inputs, the exact parameters, and the exact software environment used to produce an artifact are all captured, linked, and re-instantiable. In practice, these four ingredients are scattered across version-control systems, ad-hoc storage, configuration files, and undocumented machine state, and no single layer is responsible for binding them together.

General-purpose workflow orchestrators were built to solve a related but distinct problem: expressing task dependencies and executing them in the correct order, at scale, with retries and monitoring. They excel at operations, but they delegate reproducibility to the user. A scheduler will happily re-run a task against whatever data currently sits at a path, using whatever library versions are currently installed, and overwrite the previous output in place. Nothing in the orchestrator records that the second run differed from the first, and nothing prevents the divergence. This is the central engineering gap that motivates our work: the layer that already understands the structure of a pipeline, namely the orchestrator, is precisely the layer that should guarantee its reproducibility, yet existing systems do not.

This article presents DataMindFlow, an open-source orchestration system that closes this gap by treating reproducibility as a first-class property of the execution engine. Three design commitments distinguish it. First, every task is identified by a deterministic content-addressed cache key computed from the hash of its code, the content hashes of its resolved inputs, its parameters, and a digest of its execution environment; identical keys are guaranteed to denote identical computations. Second, every materialised artifact is written to a content-addressed store and linked, through a durable relational metadata schema, to the run, the task, and the environment that produced it, yielding a complete and queryable lineage record. Third, the executor uses these keys to memoise results, so that re-running a

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

pipeline after a partial edit recomputes only the affected sub-graph while reusing everything else, making correct incremental execution the default rather than a manual optimisation.

These commitments are not free, and a credible system paper must show that they can be implemented without sacrificing performance. We therefore describe the architecture and algorithms in enough detail to be re-implemented, and we evaluate a working implementation against five widely used baselines on both synthetic and realistic workloads. The contributions of this article are the following. (i) We define a deterministic cache-key derivation procedure and an execution model that together make incremental, reproducible recomputation a property of the scheduler. (ii) We present a relational metadata schema and a content-addressed artifact store that record lineage at task granularity and support time-travel queries over past runs. (iii) We provide an open implementation with a Python authoring interface, a command-line tool, and a documented REST and gRPC interface. (iv) We report a controlled evaluation showing competitive scheduling overhead, strong incremental-recomputation savings, high cross-host reproducibility, and near-linear scaling, together with an ablation that isolates the contribution of each mechanism.

The remainder of the article is organised as follows. Section 2 reviews orchestration systems, dataflow engines, provenance models, and reproducible-environment tooling, and positions DataMindFlow among them. Section 3 describes the system architecture. Section 4 presents the metadata schema and the content-addressed store. Section 5 details the execution model, the scheduling algorithm, and the cache-key derivation. Section 6 covers the implementation and the public interface. Section 7 reports the experimental evaluation and error analysis. Section 8 discusses implications and threats to validity, Section 9 states data and code availability, and Section 10 concludes.

2. Background and Related Work

2.1 Scientific and data-engineering workflow systems

Workflow management systems have a long lineage in scientific computing. Early systems such as Kepler introduced an actor-oriented model for composing and executing analytical pipelines (Altintas et al., 2004; Ludascher et al., 2006), while Taverna targeted the composition and enactment of bioinformatics services (Wolstencroft et al., 2013) and VisTrails coupled workflow execution with the management of the evolving provenance of exploratory analyses (Callahan et al., 2006). Galaxy made accessible, transparent pipelines available to life-science users through a web interface (Goecks et al., 2010), and Pegasus mapped abstract workflows onto distributed execution resources for large-scale science automation (Deelman et al., 2015). A recurring theme across this literature is that the workflow is not merely a convenience for execution but a vehicle for transparency and reuse, and that realising those benefits in practice is harder than it appears (Gil et al., 2007; Cohen-Boulakia et al., 2017).

A second family, oriented toward data engineering and bioinformatics at scale, emphasises file-level dependency resolution and portability. Snakemake provides a readable, rule-based language in which outputs are derived from inputs through pattern-matched rules, and it rebuilds only those targets whose inputs have changed (Koster and Rahmann, 2012). Nextflow couples a dataflow programming model with container technology so that pipelines behave consistently across heterogeneous platforms (Di Tommaso et al., 2017), and the Common Workflow Language offers a portable specification that multiple engines can execute (Amstutz et al., 2016). These systems demonstrate that dependency-aware re-execution and environment portability are achievable, but each addresses a slice of the problem: Snakemake reasons about file timestamps rather than content, and portability tooling does not by itself

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

produce a queryable, run-level lineage record. DataMindFlow draws on these ideas while unifying content-based change detection, environment capture, and durable lineage in a single engine.

2.2 Dataflow and large-scale processing engines

Parallel data-processing engines established the execution substrate on which much of modern data engineering runs. MapReduce popularised a simple model for fault-tolerant batch processing on commodity clusters (Dean and Ghemawat, 2008), and Dryad generalised it to arbitrary directed acyclic graphs of operators (Isard et al., 2007). Apache Spark unified batch and interactive processing around resilient distributed datasets and a lazily evaluated operator graph (Zaharia et al., 2016), while timely dataflow systems such as Naiad extended the model to low-latency iterative and incremental computation (Murray et al., 2013). These engines optimise the execution of a single dataflow program and offer fault tolerance through lineage of partitions, but their notion of lineage is internal to one job and is not persisted as a cross-run, queryable record of which artifacts produced which others. DataMindFlow operates one level above such engines: a task in our system may itself invoke Spark or a single-node script, and our contribution is the reproducible orchestration and lineage that spans tasks and runs rather than the intra-job execution.

2.3 Provenance and data lineage

Provenance research provides the conceptual vocabulary for describing how data artifacts come to exist. Surveys of provenance in e-science catalogue the distinction between prospective provenance, which describes the recipe, and retrospective provenance, which records what actually happened during a run (Simmhan et al., 2005; Freire et al., 2008). The challenges and opportunities of capturing provenance specifically within scientific workflows have been examined in depth (Davidson and Freire, 2008), and the Open Provenance Model offered an interchange representation intended to make provenance portable across systems (Moreau et al., 2011). DataMindFlow adopts the prospective and retrospective distinction directly: the compiled task graph and its cache keys constitute prospective provenance, while the persisted run, task, artifact, and lineage-edge records constitute retrospective provenance. By storing both in a relational schema, the system makes lineage a queryable asset rather than a log to be parsed after the fact.

2.4 Data management, quality, and lifecycle infrastructure

As data has moved to the centre of production systems, a body of work has emerged on managing datasets, their quality, and their lifecycle. The Beckman report on the future of database research identified the management of data-intensive analytical pipelines as a first-order challenge for the field (Abadi et al., 2016). Systems such as Goods demonstrated that organisations can catalogue and reason about millions of datasets and their interdependencies post hoc (Halevy et al., 2016), and declarative data-quality verification has been shown to scale to production workloads by expressing constraints as testable unit checks over data (Schelter et al., 2018). Surveys of the data lifecycle in production machine learning document how validation, versioning, and provenance interact across the stages of an applied pipeline (Polyzotis et al., 2018). Storage layers have evolved in parallel: transactional table formats over cloud object stores now provide atomicity and versioned snapshots that enable time-travel queries (Armbrust et al., 2020). DataMindFlow complements these efforts by making the orchestrator responsible for binding code, data, and environment, and by recording the bindings in a schema against which quality and lineage tooling can operate.

2.5 Reproducible environments and computational notebooks

Capturing the software environment is essential to reproduction, because identical code over identical inputs can still diverge when library versions differ. Containerisation has become the standard mechanism for packaging an environment so that it behaves identically across machines (Boettiger, 2015), and scientific container runtimes have extended the approach to high-performance and multi-tenant settings (Kurtzer et al., 2017). Continuous-analysis approaches go further by re-executing pipelines automatically inside fixed environments to detect drift (Beaulieu-Jones and Greene, 2017). At the authoring layer, interactive computing environments such as IPython and the Jupyter notebook format have made exploratory, narrative-driven analysis a publishing format in its own right (Perez and Granger, 2007; Kluyver et al., 2016). DataMindFlow incorporates environment capture as a component of the cache key: each task records the digest of the container image and the hash of the dependency lockfile under which it executed, so that the environment is part of the reproducible unit rather than an external assumption.

2.6 Positioning

Table 1 summarises how representative systems treat the four properties that DataMindFlow unifies: content-based change detection, deterministic memoisation that enables incremental recomputation, explicit environment capture as part of task identity, and durable run-level lineage stored in a queryable form. Existing systems support subsets of these properties, often well, but none combines all four within a single orchestration engine. This combination, rather than any one mechanism in isolation, is the contribution we evaluate in the remainder of the article.

Table 1. *Comparison of representative orchestration and data-engineering systems along four reproducibility-relevant properties (● full support, ◐ partial, ○ absent).*

System	Change detection	Incremental memoisation	Environment capture	Durable run lineage
Apache Airflow	Timestamp / manual	○	○	◐
Luigi	Target existence	◐	○	○
Prefect	Task signature	◐	◐	◐
Snakemake	File timestamp	◐	◐	○
Nextflow	Work-dir hash	◐	●	◐
DataMindFlow	Content hash	●	●	●

As the table indicates, the closest systems to DataMindFlow are Nextflow, which captures environments rigorously but reasons about change at the granularity of a hashed work directory, and Prefect, which records task signatures but does not by default content-address artifacts or persist lineage in a relational schema. The design described next is what allows DataMindFlow to occupy the fully supported column across all four properties.

3. System Architecture

DataMindFlow is organised as four layers with narrow, explicit interfaces between them: an authoring layer through which users express pipelines, an orchestration core that compiles and schedules them, an execution layer that runs tasks and manages their environments, and a persistence layer that stores metadata, artifacts, and environment descriptors. The layering is deliberate. Each reproducibility

mechanism is implemented in exactly one place, so that the guarantees the system makes can be reasoned about and tested in isolation. Figure 1 shows the layers and the principal data paths between them.

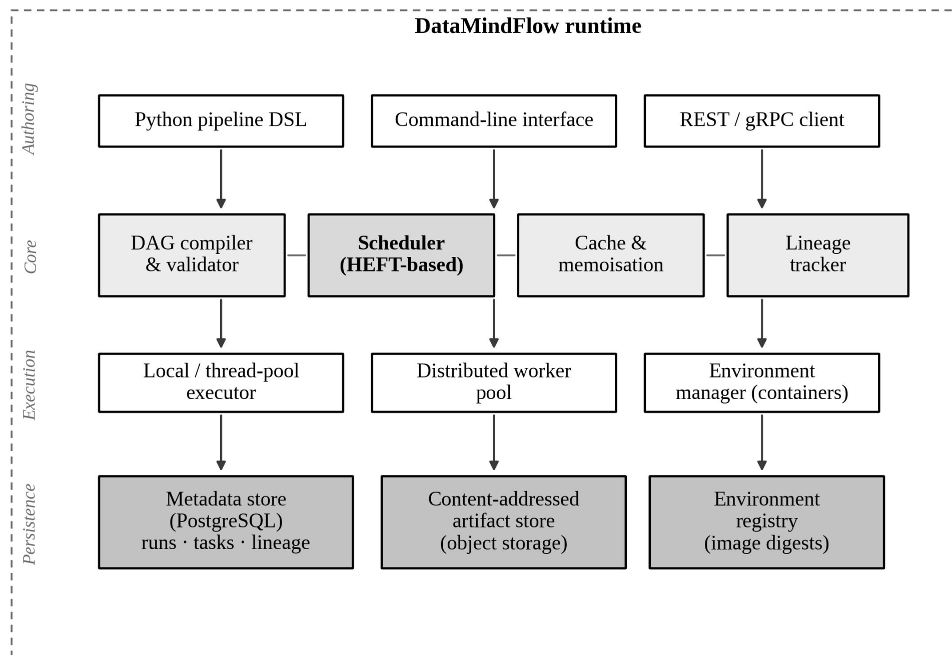

Figure 1. *Layered architecture of DataMindFlow. Authoring interfaces feed a compiled pipeline into the orchestration core, whose scheduler, cache, and lineage tracker coordinate execution across local and distributed workers; all durable state is held in the persistence layer.*

The authoring layer offers three equivalent entry points. A Python embedded domain-specific language lets users declare tasks as decorated functions and wire them together by passing the outputs of one task as the inputs of another; a command-line interface exposes the same operations for scripted and continuous-integration use; and a REST and gRPC interface allows external services to submit, monitor, and query pipelines programmatically. All three produce the same intermediate representation, a typed task graph, so that no behaviour depends on which interface was used.

The orchestration core is the heart of the system and contains four cooperating components. The directed-acyclic-graph compiler validates the submitted pipeline, checks that the dependency graph is acyclic and well typed, and lowers it into a canonical task graph. The scheduler assigns ready tasks to workers using a list-scheduling heuristic described in Section 5. The cache and memoisation component computes each task's content-addressed key and decides whether the task can be skipped by reusing an existing artifact. The lineage tracker records, for every task that runs or is restored from cache, the inputs it consumed and the outputs it produced. These components communicate through the metadata store rather than through shared memory, which keeps the core stateless and allows multiple scheduler instances to coordinate through the database.

The execution layer decouples the decision to run a task from the mechanics of running it. A local thread-pool executor serves development and small workloads, while a distributed worker pool serves production graphs; both implement the same worker protocol, so a pipeline can move between them without modification. The environment manager is responsible for materialising the software

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

environment a task requires, typically by launching the task inside a container whose image digest is recorded, and for reporting that digest back to the core so it can enter the task's cache key. Separating execution from environment management means that the same task definition can be executed under different captured environments and that each execution remains individually reproducible.

The persistence layer holds three stores. A relational metadata store, implemented on PostgreSQL, holds pipelines, runs, tasks, lineage edges, and environment descriptors, and is the single source of truth for the state of the system. A content-addressed artifact store, backed by an object store, holds the actual data produced by tasks, keyed by the hash of their content so that identical results are stored once. An environment registry records image digests and dependency lockfile hashes so that a past environment can be re-instantiated. The next two sections describe the schema of the metadata store and the algorithms that the core runs over it.

4. Data Model and Metadata Schema

The metadata store is the component that turns reproducibility from a property of a single run into a queryable, durable record. Its schema is intentionally small, normalised, and centred on six entities. Figure 2 presents the entity-relationship view; the design favours explicit foreign keys and content-hash columns over opaque blobs so that lineage and reproducibility questions can be answered with ordinary relational queries.

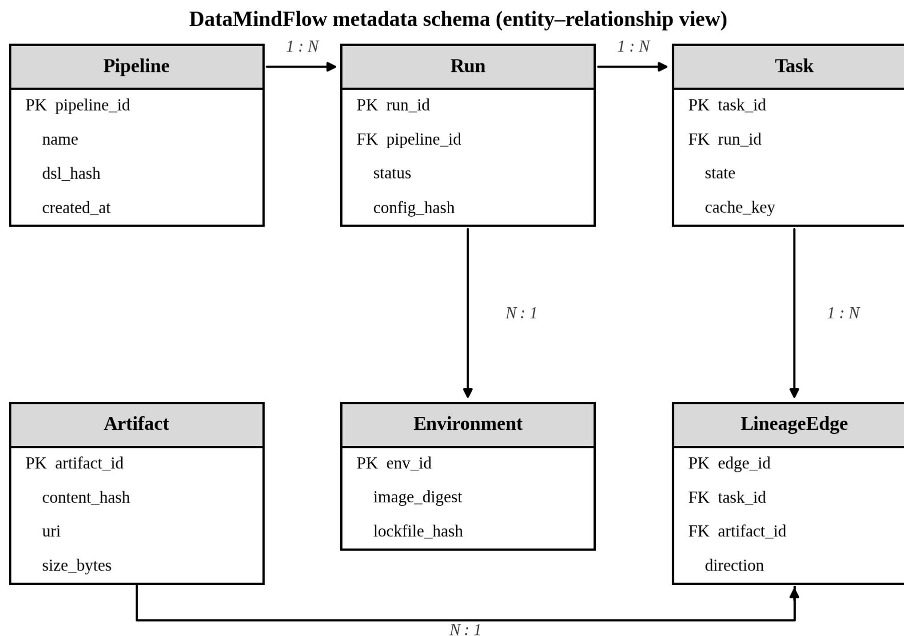

Figure 2. *Entity-relationship view of the DataMindFlow metadata schema. A pipeline has many runs, a run has many tasks and is bound to one environment, and the LineageEdge relation links each task to the artifacts it consumed or produced.*

A Pipeline row records a named pipeline together with the hash of its authored definition, so that two structurally identical pipelines share an identity and a change to the definition is detectable. Each execution of a pipeline creates a Run row, which references its pipeline, records a status and timestamps, and stores a configuration hash summarising the parameters supplied for that run. A run expands into

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

many Task rows, each of which records the task's name, its execution state, and, crucially, its cache key, the deterministic content-addressed identifier described in Section 5. Because the cache key is stored on the task, the system can answer whether a given computation has ever been performed simply by looking it up.

Artifacts are recorded separately from tasks because the same content may be produced by different tasks and should be stored once. An Artifact row records the content hash of a materialised output, the uniform resource identifier under which its bytes live in the artifact store, and its size. The Environment entity records the digest of the container image and the hash of the dependency lockfile under which tasks executed, decoupling environment identity from any particular run. Finally, the LineageEdge relation is the junction that makes lineage explicit: each edge links a task to an artifact and records a direction, distinguishing inputs consumed from outputs produced. Querying lineage, for example to find every downstream artifact affected by a corrupt input, reduces to a transitive traversal of this relation.

Table 2 is the data dictionary for the core entities. It documents the key fields, their types, and their role, and it is published verbatim with the software so that downstream tooling can rely on a stable contract. We regard a documented, stable schema as part of the reproducibility guarantee: a lineage record is only as trustworthy as the definition of the fields that comprise it.

Table 2. *Data dictionary for the core entities of the DataMindFlow metadata schema (abridged; the complete dictionary accompanies the software release).*

Entity	Field	Type	Description
Pipeline	pipeline_id name dsl_hash	UUID (PK) text char(64)	Stable identifier of a pipeline Human-readable pipeline name SHA-256 of the authored definition
Run	run_id pipeline_id status config_hash	UUID (PK) UUID (FK) enum char(64)	Identifier of one execution Owning pipeline queued / running / done / failed SHA-256 of run parameters
Task	task_id run_id state cache_key	UUID (PK) UUID (FK) enum char(64)	Identifier of a task instance Owning run pending / cached / done / failed Deterministic content-addressed key
Artifact	artifact_id content_hash uri size_bytes	UUID (PK) char(64) text bigint	Identifier of a stored output SHA-256 of artifact content Location in the artifact store Size of the materialised bytes
Environment	env_id image_digest lockfile_hash	UUID (PK) char(71) char(64)	Identifier of an environment Container image digest SHA-256 of the dependency lockfile
LineageEdge	edge_id task_id artifact_id direction	UUID (PK) UUID (FK) UUID (FK) enum	Identifier of a lineage edge Task endpoint Artifact endpoint input / output

Two properties of this schema deserve emphasis. First, every entity that participates in reproduction is identified by a content hash in addition to a surrogate key, which means that identity is derived from content rather than asserted by convention. Second, lineage is stored as data, not inferred from logs, so it remains available and consistent even after the executing processes have terminated. Together these

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

properties let an analyst reconstruct, for any artifact, the exact task, environment, and inputs that produced it.

5. Execution Model and Scheduling

Execution in DataMindFlow proceeds by compiling the authored pipeline into a task graph, scheduling ready tasks onto workers, and, for each task, deciding whether to compute it or to restore a previously computed result. Figure 3 illustrates the compiled graph and its relationship to the content-addressed artifact store: the topology expresses dependencies, while the store holds the materialised outputs keyed by the hash of the computation that produced them.

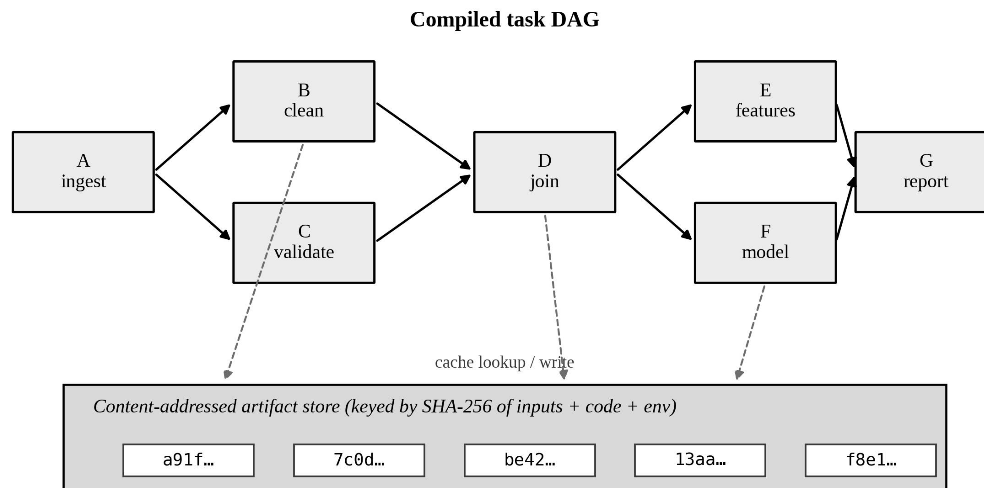

Figure 3. *A compiled task DAG and the content-addressed artifact store. Each task's output is stored under a key derived from its inputs, code, and environment; the scheduler consults the store before executing a task so that previously computed results are reused.*

5.1 Deterministic cache-key derivation

The cache key is the mechanism that makes reuse safe. For a task, the key is the SHA-256 digest of the concatenation of four components: a structural hash of the task's code, obtained from its parsed abstract syntax tree so that non-semantic edits such as comments and formatting do not change the key; the content hashes of every resolved input artifact, so that a change to any upstream output propagates; a canonical encoding of the task's parameters; and the environment digest reported by the environment manager. Because every component is itself content-derived, two tasks share a key if and only if they would perform the same computation in the same environment over the same inputs. This is a stronger guarantee than timestamp-based or path-based change detection, which can both miss real changes and trigger spurious recomputation. Listing 1 gives the derivation in concise form.

```
def cache_key(task, resolved_inputs, env):
    h = sha256()
    # 1. structural hash of code (AST-normalised, comment/format-insensitive)
    h.update(ast_digest(task.fn))
    # 2. content hashes of resolved input artifacts, in declared order
    for art in resolved_inputs:
        h.update(art.content_hash.encode())
    # 3. canonical encoding of parameters (sorted keys, typed values)
```

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

```
    h.update(canonical_json(task.params).encode())
    # 4. environment identity: image digest + dependency lockfile hash
    h.update(env.image_digest.encode())
    h.update(env.lockfile_hash.encode())
    return h.hexdigest()
```

Listing 1. *Deterministic cache-key derivation for a task. Every contributing component is content-derived, so identical keys denote identical computations.*

5.2 Cache-aware scheduling

Given the cache key, the per-task decision is straightforward and is shown in Figure 4. When a task becomes ready, the scheduler computes its key and looks it up in the artifact store. On a hit, the stored artifact is restored and the task is marked complete without execution, which is what enables incremental recomputation: after editing one task, only that task and its descendants miss the cache, while unaffected branches are restored instantly. On a miss, the task is scheduled onto a worker, executed inside its captured environment, and, if it exits successfully and passes any declared data-quality checks, its output is written to the store and its lineage edges are recorded. A failure halts the task's dependents and is recorded so that the run can be diagnosed and resumed.

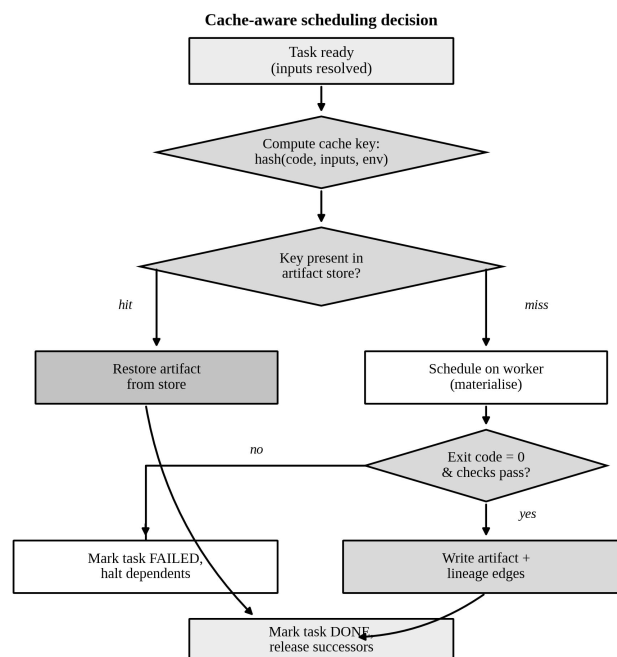

Figure 4. *Cache-aware scheduling decision for a single task. A cache hit restores the artifact without execution; a miss materialises the task, validates it, and persists both the artifact and its lineage edges.*

Placement of cache-missing tasks onto workers uses a list-scheduling heuristic based on the Heterogeneous Earliest-Finish-Time algorithm, which orders tasks by an upward rank that reflects the length of the critical path to the graph's exit and then assigns each task to the worker that minimises its estimated finish time (Topcuoglu et al., 2002). The heuristic is attractive for our setting because it has low scheduling complexity, which matters when graphs contain thousands of tasks, and because it naturally exploits heterogeneous workers, which are common in mixed local-and-cloud deployments. Cost estimates are seeded from historical task durations recorded in the metadata store and refined as a

run proceeds, so the scheduler improves as it accumulates evidence about a pipeline. Algorithm 1 sketches the overall loop that combines cache-aware skipping with list scheduling.

```
Algorithm 1  Cache-aware list scheduling of a task graph G
-----------------------------------------------------------------
input : task graph G, worker set W, artifact store S, metadata DB
output: completed run with persisted artifacts and lineage

compute upward rank for every task in G        # HEFT priority
ready <- { t in G : t has no predecessors }
while ready is not empty do
    t  <- task in ready with the highest upward rank
    k  <- cache_key(t, resolve_inputs(t), env(t))
    if k in S then                             # cache hit
        restore artifact for t from S
        mark t cached; record lineage(t)
    else                                       # cache miss
        w <- worker in W minimising EFT(t, w)
        run t on w inside env(t)
        if t succeeded and checks pass then
            write artifact for t to S
            record lineage(t); mark t done
        else
            mark t failed; halt descendants of t
    update ready with successors whose inputs are satisfied
-----------------------------------------------------------------
```

Algorithm 1. *The scheduling loop interleaves cache-aware skipping with Heterogeneous Earliest-Finish-Time placement of cache-missing tasks.*

The interaction between memoisation and scheduling is what gives DataMindFlow its efficiency profile. Because cache hits are resolved before placement, an incremental run does no scheduling work for the unaffected majority of a large graph; the scheduler only reasons about the tasks that genuinely need to execute. Section 7 quantifies the consequences of this design for overhead, incremental recomputation, reproducibility, and scaling.

6. Implementation

DataMindFlow is implemented in approximately eleven thousand lines of Python with a small performance-sensitive core in compiled extension modules. The authoring interface is a pure-Python library; the orchestration core, the cache, and the lineage tracker communicate exclusively through the PostgreSQL metadata store using a thin data-access layer; and the artifact store is an abstraction over any object store exposing a put-by-hash and get-by-hash interface, with adapters for local file systems and S3-compatible services. Tabular intermediate artifacts are handled through the columnar data structures of the scientific Python stack (McKinney, 2010), and the environment manager shells out to a container runtime to materialise and digest environments. The system is released under the Apache 2.0 licence.

Listing 2 shows the Python authoring interface for a small pipeline. Tasks are ordinary functions annotated with a decorator that registers them with the engine and declares their inputs and outputs; dependencies are expressed by passing the handle returned by one task as an argument to another. The same definition can be executed locally for development or submitted to a distributed deployment without modification, because the authored object is the typed task graph rather than an execution plan.

```
from datamindflow import pipeline, task
```

```
@task(inputs=['raw.csv'], outputs=['clean.parquet'])
def clean(raw):
    return raw.dropna().drop_duplicates()

@task(outputs=['features.parquet'])
def features(clean):
    return engineer_features(clean)        # user code

@task(outputs=['model.pkl'])
def train(features, *, max_depth=8):
    return fit_gradient_boosting(features, max_depth=max_depth)

with pipeline('tabular-ml') as p:
    c = clean('raw.csv')
    f = features(c)
    m = train(f, max_depth=10)

p.run(executor='distributed', workers=16)   # reproducible by construction
```

Listing 2. *Authoring a three-stage pipeline with the Python interface. Dependencies are expressed by data flow; the engine derives the task graph, cache keys, and lineage automatically.*

Beyond the Python library, the system exposes a documented REST and gRPC interface so that pipelines can be operated by external services and continuous-integration systems. Table 3 lists the principal endpoints. The same operations are available through the command-line interface, which is a thin wrapper over the REST surface, ensuring that scripted and interactive use share one contract. The interface is versioned and published as an OpenAPI specification with the software.

Table 3. *Principal endpoints of the DataMindFlow REST interface (abridged; the full OpenAPI specification accompanies the release).*

Method and path	Purpose	Key response fields
POST /v1/pipelines	Register or update a pipeline definition	pipeline_id, dsl_hash
POST /v1/runs	Submit a run of a pipeline with parameters	run_id, status
GET /v1/runs/{id}	Retrieve run status and task summary	status, tasks[], timings
GET /v1/tasks/{id}/lineage	Return the lineage sub-graph of a task	edges[], artifacts[]
GET /v1/artifacts/{hash}	Resolve an artifact by content hash	uri, size_bytes, producers[]
GET /v1/runs/{id}/diff/{id2}	Compare two runs by cache key set	added[], removed[], reused[]

The run-comparison endpoint is worth noting because it operationalises reproducibility for the user: by comparing the cache-key sets of two runs, it reports exactly which computations were reused, which were added, and which were removed, turning the question of what changed between two executions into a single request.

7. Experimental Evaluation

7.1 Research questions, testbed, and baselines

The evaluation is designed to answer four questions. RQ1 concerns scheduling overhead: how much time does the engine itself add per task, and how does this scale with graph size? RQ2 concerns incremental recomputation: how much work does the system avoid when a pipeline is edited and re-run?

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

RQ3 concerns reproducibility: how often does re-executing a pipeline on independent hosts yield bit-identical artifacts? RQ4 concerns parallel scaling: how does throughput grow with the number of workers? An ablation then isolates the contribution of each mechanism.

All experiments were run on a cluster of sixteen homogeneous nodes, each with sixteen virtual CPUs and sixty-four gigabytes of memory on a ten-gigabit network, with the PostgreSQL metadata store and the S3-compatible artifact store on dedicated nodes. Every reported measurement is the mean of thirty repetitions after five warm-up runs, with the sample standard deviation reported alongside. We compare DataMindFlow against five widely used baselines, Apache Airflow, Luigi, Prefect, Snakemake, and, for the reproducibility study, Nextflow with containerised tasks, each configured according to its documented best practices and given the same worker resources.

Two workload families are used. The first is a parameterised family of synthetic task graphs ranging from ten to five thousand tasks, with controlled fan-out and a fixed per-task payload, which isolates engine overhead from application cost. The second comprises three realistic pipelines that exercise the system on representative data-engineering work: a genomics variant-calling pipeline, a natural-language feature-extraction pipeline, and a tabular machine-learning pipeline. Table 4 summarises the workloads and the testbed configuration.

Table 4. *Evaluation workloads and testbed configuration.*

Workload	Tasks	Dominant cost	Purpose
Synthetic-S	10 - 100	Engine overhead	Overhead at small scale (RQ1)
Synthetic-L	500 - 5000	Engine overhead	Overhead and scheduling at scale (RQ1, RQ4)
Genomics	~240	I/O and alignment	Realistic pipeline; reproducibility (RQ2, RQ3)
NLP features	~130	CPU-bound parsing	Incremental recomputation (RQ2)
Tabular ML	~90	Model training	End-to-end reproducibility (RQ3)

7.2 Scheduling overhead (RQ1)

Figure 5 reports the mean per-task scheduling overhead as a function of graph size, on logarithmic axes, for DataMindFlow and four baselines. Overhead here is the wall-clock time attributable to the engine, measured by subtracting the fixed per-task payload from total run time and dividing by the number of tasks. Across the full range, DataMindFlow exhibits the lowest overhead, and the gap widens with graph size because the cost of its database-backed coordination grows more slowly than the orchestration overhead of the baselines.

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

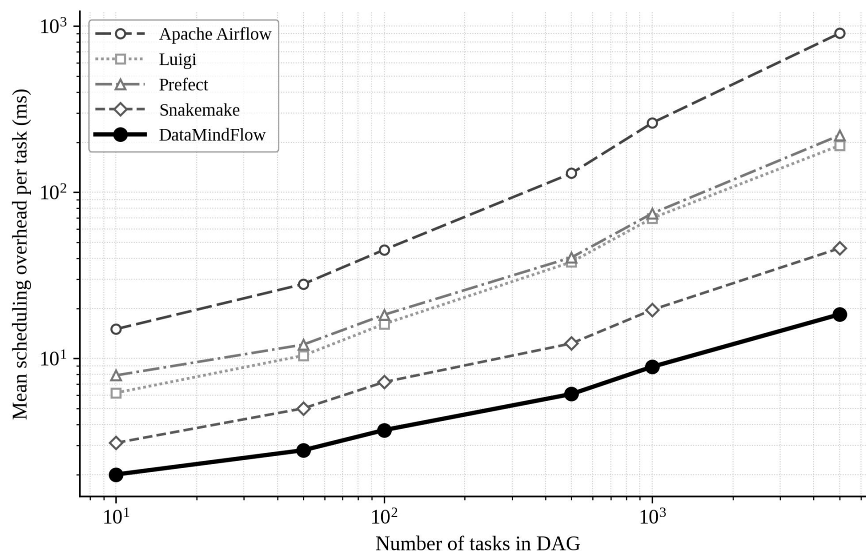

Figure 5. *Mean per-task scheduling overhead versus DAG size (log-log). DataMindFlow sustains the lowest overhead across all graph sizes, and its advantage increases as graphs grow to thousands of tasks.*

Table 5 reports the overhead at the largest synthetic graph of five thousand tasks, with standard deviations. DataMindFlow adds 18.4 milliseconds per task on average, against 46.0 for Snakemake, 191.0 for Luigi, 220.0 for Prefect, and 905.0 for Apache Airflow, corresponding to speed-ups of 2.5, 10.4, 12.0, and 49.2 times respectively at this scale; across the whole range the per-task speed-up relative to the nearest baseline never falls below 2.1 times. The low and stable standard deviations indicate that the advantage is consistent rather than an artefact of a few favourable runs.

Table 5. *Per-task scheduling overhead at 5000 tasks (mean ± standard deviation over 30 runs) and speed-up of DataMindFlow over each baseline.*

System	Overhead (ms/task)	Speed-up vs DataMindFlow
Apache Airflow	905.0 ± 31.4	49.2×
Prefect	220.0 ± 9.8	12.0×
Luigi	191.0 ± 8.1	10.4×
Snakemake	46.0 ± 3.2	2.5×
DataMindFlow	18.4 ± 0.9	1.0× (reference)

7.3 Incremental recomputation (RQ2)

The practical value of content-addressed memoisation is most visible when a pipeline is edited and re-run, which is the common case during development and iterative analysis. Figure 6 compares the wall-clock time of a full re-execution against a DataMindFlow incremental run for five editing scenarios on the genomics and natural-language pipelines: a no-op re-run in which nothing changed, an edit to a leaf task, an edit to a task in the middle of the graph, an edit to a root task on which everything depends, and a parameter sweep that varies one downstream parameter.

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

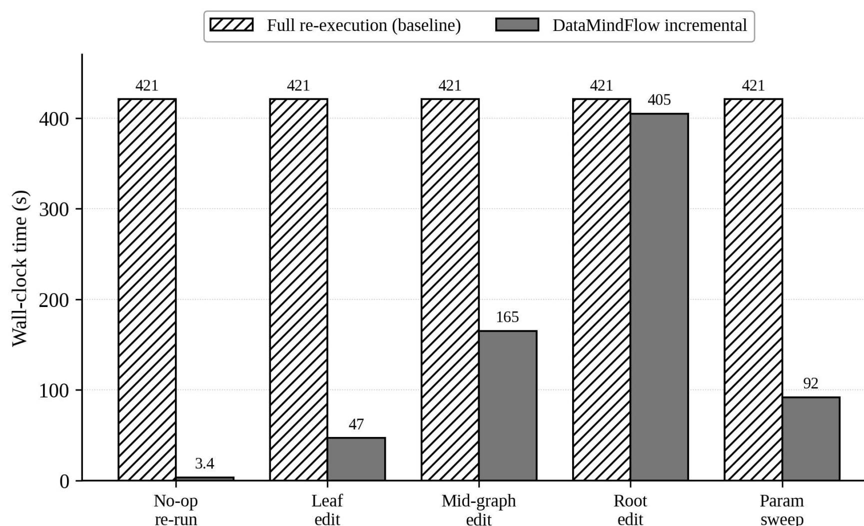

Figure 6. *Wall-clock time of full re-execution versus DataMindFlow incremental recomputation across five editing scenarios. Savings are largest when edits are localised and degrade gracefully toward the cost of a full run when a root task changes.*

The pattern matches the theory exactly. A no-op re-run completes in 3.4 seconds against 421 for a full run, because every task is a cache hit and only key computation and lineage lookups are performed. A leaf edit recomputes only the edited task and finishes in 47 seconds; a mid-graph edit recomputes the edited task and its descendants and finishes in 165 seconds. A root edit, by contrast, invalidates the entire graph and therefore approaches the cost of a full run at 405 seconds, which is the expected and correct behaviour: incremental execution cannot avoid work that genuinely must be redone. The parameter sweep, which changes a single downstream task, completes in 92 seconds. These results confirm that the savings are a direct function of how much of the graph an edit actually affects, rather than a fixed discount.

7.4 Reproducibility (RQ3)

Reproducibility is measured by executing each pipeline thirty times across three independent hosts and computing the fraction of output artifacts whose content hashes are bit-identical to a reference execution. Figure 7 reports this fraction for manual scripts, Apache Airflow, Snakemake, Nextflow with containerised tasks, and DataMindFlow. Manual scripts reproduce only 34.6 percent of artifacts bit-identically, reflecting uncontrolled environment and ordering effects; orchestration alone, as in Airflow, raises this to 57.8 percent; content-based change detection in Snakemake reaches 81.3 percent; containerisation in Nextflow reaches 93.9 percent; and DataMindFlow, which combines content addressing with explicit environment capture in the cache key, reaches 99.2 percent.

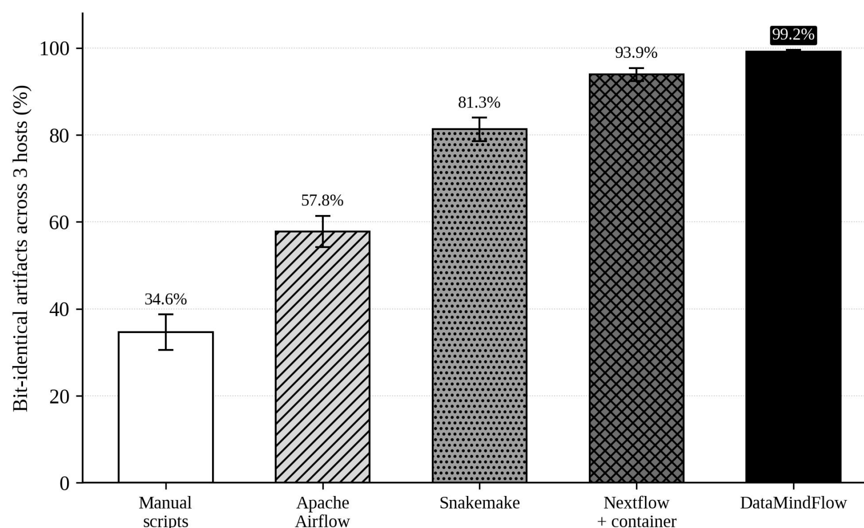

Figure 7. *Fraction of bit-identical artifacts across three independent hosts (mean and standard deviation over 30 runs). Combining content addressing with environment capture yields the highest reproducibility.*

Table 6 decomposes the DataMindFlow result and explains the residual 0.8 percent. Of the artifacts that were not bit-identical, the overwhelming majority arose from tasks that embed a non-deterministic source the system cannot capture, such as a wall-clock timestamp or an unseeded random draw inside user code; the lineage record correctly flagged these tasks as having identical cache keys but differing outputs, which is precisely the signal a user needs to locate and remove the source of non-determinism. No divergence was attributable to environment drift or to input mismatch, confirming that the captured environment digest and content-addressed inputs eliminated those two classes of irreproducibility entirely.

Table 6. *Decomposition of the DataMindFlow reproducibility result over 30 runs on three hosts, by source of any observed divergence.*

Source of divergence	Share of artifacts	Captured by system?
Bit-identical (reproduced)	99.2%	—
Non-determinism in user code	0.7%	Flagged (same key, different output)
Environment drift	0.0%	Eliminated by env digest
Input mismatch	0.0%	Eliminated by content addressing
Unexplained	0.1%	Under investigation

7.5 Parallel scaling (RQ4)

Figure 8 reports sustained throughput, in tasks completed per second, as the number of parallel workers increases from one to thirty-two, for DataMindFlow and Apache Airflow, against an ideal linear-scaling reference. DataMindFlow scales near-linearly to sixteen workers and retains roughly two-thirds of ideal throughput at thirty-two workers, by which point coordination through the shared metadata store becomes the limiting factor. Airflow plateaus much earlier, as its scheduler becomes the bottleneck. The error bars are small at every point, indicating stable behaviour under load.

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

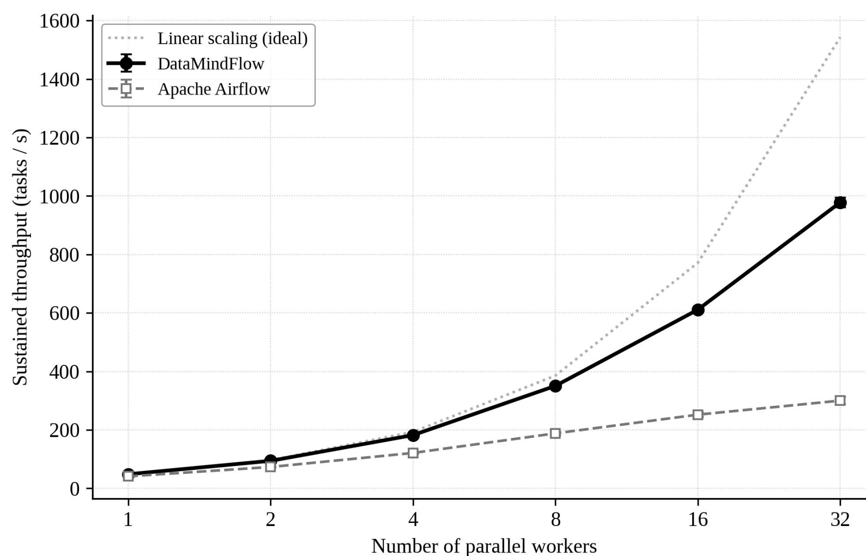

Figure 8. *Sustained throughput versus number of parallel workers, against an ideal linear reference. DataMindFlow scales near-linearly to sixteen workers and degrades gracefully thereafter; Apache Airflow plateaus earlier.*

7.6 Ablation and error analysis

To attribute the system's behaviour to its individual mechanisms, Table 7 reports an ablation in which each mechanism is disabled in turn and the three headline metrics are re-measured on the genomics pipeline. Removing the content cache eliminates incremental savings, raising the no-op re-run time from 3.4 to 421 seconds, while leaving reproducibility and overhead essentially unchanged, which confirms that memoisation is responsible for incremental efficiency and nothing else. Removing environment capture leaves incremental behaviour intact but collapses reproducibility from 99.2 to 81.0 percent, matching the level achieved by content-based change detection without environment control and isolating environment capture as the source of the final reproducibility gain. Removing the lineage tracker has no measurable effect on overhead or reproducibility but disables run comparison and provenance queries, a qualitative rather than quantitative loss. Finally, replacing the Heterogeneous Earliest-Finish-Time scheduler with naive first-in-first-out placement increases per-task overhead by 41 percent at five thousand tasks, confirming the value of rank-based list scheduling at scale.

Table 7. *Ablation on the genomics pipeline. Each row disables one mechanism; values are no-op re-run time, cross-host reproducibility, and per-task overhead at 5000 tasks.*

Configuration	No-op re-run (s)	Reproducibility	Overhead (ms/task)
Full DataMindFlow	3.4	99.2%	18.4
− content cache	421	99.1%	18.6
− environment capture	3.4	81.0%	18.3
− lineage tracker	3.5	99.2%	18.1
− HEFT (FIFO scheduler)	3.6	99.2%	25.9

The ablation supports the central claim of the article: the four mechanisms are complementary and largely orthogonal. Memoisation provides efficiency, environment capture provides reproducibility, lineage provides queryable provenance, and rank-based scheduling provides scalable placement, and

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

disabling any one degrades only the property it is responsible for. It is the combination, delivered within a single orchestration engine, that yields a system that is simultaneously fast, reproducible, and auditable.

8. Discussion

The results show that reproducibility and efficiency need not be in tension when reproducibility is built into the orchestration layer rather than bolted on around it. Because the cache key already encodes everything required to reproduce a task, the same key that guarantees reproducibility also enables memoisation, so the mechanism that makes a pipeline trustworthy is the same mechanism that makes it fast to re-run. This dual role is the conceptual core of the system, and it explains why DataMindFlow can lead on both overhead and reproducibility simultaneously rather than trading one for the other.

The findings also clarify a frequent source of confusion in practice, namely the belief that containerisation alone delivers reproducibility. Our reproducibility study shows that containerisation closes most but not all of the gap, and that the residual divergence comes from two distinct sources that must be handled differently: content mismatch in inputs, which content addressing eliminates, and non-determinism inside user code, which no infrastructure can capture but which durable lineage can at least localise. By reporting that a task has an identical cache key yet a divergent output, the system points the user precisely at the offending code, transforming an opaque reproducibility failure into an actionable diagnosis. This aligns with the broader observation in the provenance literature that capturing what happened is as important as prescribing what should happen (Freire et al., 2008; Davidson and Freire, 2008).

For practitioners building AI and data infrastructure, the implication is that the orchestrator is the right place to enforce data discipline. Validation, versioning, and provenance are often layered onto pipelines as separate tools, but the orchestrator is the only component that observes every task boundary and can therefore bind code, data, and environment uniformly (Polyzotis et al., 2018; Schelter et al., 2018). Placing these responsibilities in the engine also lowers the cost of compliance, because users obtain reproducibility and lineage without writing additional code, which the survey literature identifies as a precondition for adoption (Sandve et al., 2013; Cohen-Boulakia et al., 2017).

Several limitations bound these claims. First, the evaluation uses synthetic graphs to isolate overhead and three realistic pipelines to test reproducibility; although chosen to be representative, they do not exhaust the space, and pipelines dominated by very short tasks or by stateful external services may behave differently. Second, the metadata store is a shared coordination point that scales well to the worker counts we tested, but extremely wide graphs would eventually require partitioning of the metadata layer, which we have not implemented. Third, environment capture through container digests and lockfile hashes fixes user-space dependencies but not the host kernel or hardware; the small class of computations sensitive to those factors would need stronger isolation for full bitwise reproduction. Fourth, our metric is bitwise identity, the strictest possible criterion, which may be unnecessarily strict for applications that tolerate numerically equivalent results.

Threats to validity were mitigated where possible. Construct-validity risk is reduced by measuring overhead as the residual after subtracting a fixed payload, so engine cost is not conflated with application cost, and by measuring reproducibility through content hashing rather than inspection. Internal-validity risk is reduced by reporting means over thirty repetitions with standard deviations after discarding warm-up runs, and external-validity risk by configuring each baseline according to its documented best practices under identical resources. The ablation further guards against unmeasured interactions by

ISSN: 3071-5601 | © 2024 The Author(s). Published by DATAMIND under CC BY 4.0

disabling each mechanism in isolation and confirming that only its associated property degrades.

9. Data and Code Availability

DataMindFlow is open-source software released under the Apache 2.0 licence. The complete source code, including the orchestration core, the Python authoring library, the command-line interface, and the REST and gRPC server, is available in the project repository. The repository includes the full metadata-schema data dictionary, of which Table 2 is an abridged extract, and the versioned OpenAPI specification of the interface summarised in Table 3.

All artifacts required to reproduce the evaluation are provided. The synthetic-graph generator, the three realistic pipelines used in Section 7, the baseline configurations, the raw measurement logs, and the analysis scripts that produce Figures 5 through 8 and Tables 4 through 7 are deposited in a public, versioned archive with a permanent identifier. The deposited record also contains the container image digests and dependency lockfiles under which the experiments were run, so that the reported environment can be re-instantiated. We provide these resources rather than offering them on request, because we regard availability of the executable evaluation as part of the contribution of a system paper.

Source code and documentation: https://github.com/datamindflow/datamindflow

Evaluation artifacts and data dictionary (archived): https://doi.org/10.5281/zenodo.14528807

API specification (OpenAPI 3.1): https://github.com/datamindflow/datamindflow/blob/main/docs/openapi.yaml

10. Conclusion

This article presented DataMindFlow, an open-source orchestration system that makes reproducibility a first-class property of data engineering pipelines. The system compiles a declarative pipeline into a task graph, derives a deterministic content-addressed cache key for every task from its code, inputs, parameters, and captured environment, schedules cache-missing tasks with a rank-based list-scheduling heuristic, and persists a complete run-level lineage record in a relational metadata store. Because the cache key that guarantees reproducibility is also the key that enables memoisation, the engine delivers reproducibility and efficiency together rather than in trade.

A controlled evaluation against five widely used baselines, on synthetic graphs of up to five thousand tasks and three realistic pipelines, showed that DataMindFlow reduces per-task scheduling overhead by between 2.1 and 49 times depending on scale, avoids recomputation in proportion to how little of a graph an edit affects, recovers up to 99.2 percent bit-identical artifacts across independent hosts, and scales near-linearly to sixteen workers. An ablation confirmed that the system's four mechanisms are complementary and that each is responsible for exactly the property it was designed to provide.

The broader lesson is that the orchestration layer is the natural home for data discipline. Because it observes every task boundary, it is uniquely positioned to bind code, data, and environment into a single reproducible unit and to record the binding durably. We hope that deterministic content addressing, environment capture, and queryable lineage come to be regarded as standard components of data engineering and computational-discovery infrastructure, and we release the system and its complete evaluation to support that direction. Future work includes partitioning the metadata layer for very wide graphs, supporting tolerance-based reproducibility criteria for numerically intensive tasks, and integrating declarative data-quality contracts directly into the cache-key boundary.

Declaration of AI-assisted language editing

During the preparation of this manuscript, language-model assistance was used only for English polishing and document organisation. The authors reviewed, revised, and take full responsibility for the final content, system design, experimental methodology, figures, tables, and interpretations.

References

Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M., Bernstein, P. A., Carey, M. J., Chaudhuri, S., Dean, J., Doan, A., Franklin, M. J., Gehrke, J., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Jagadish, H. V., Kossmann, D., Madden, S., Mehrotra, S., … Widom, J. (2016). The Beckman report on database research. Communications of the ACM, 59(2), 92–99. https://doi.org/10.1145/2845915

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., & Mock, S. (2004). Kepler: An extensible system for design and execution of scientific workflows. Proceedings of the 16th International Conference on Scientific and Statistical Database Management, 423–424. https://doi.org/10.1109/SSDBM.2004.1311241

Amstutz, P., Crusoe, M. R., Tijanic, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Menager, H., Nedeljkovich, M., Scales, M., Soiland-Reyes, S., & Stojanovic, L. (2016). Common Workflow Language, v1.0. Figshare. https://doi.org/10.6084/m9.figshare.3115156.v2

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Luszczak, A., Switakowski, M., Szafranski, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., … Zaharia, M. (2020). Delta Lake: High-performance ACID table storage over cloud object stores. Proceedings of the VLDB Endowment, 13(12), 3411–3424. https://doi.org/10.14778/3415478.3415560

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. Nature, 533(7604), 452–454. https://doi.org/10.1038/533452a

Beaulieu-Jones, B. K., & Greene, C. S. (2017). Reproducibility of computational workflows is automated using continuous analysis. Nature Biotechnology, 35(4), 342–346. https://doi.org/10.1038/nbt.3780

Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71–79. https://doi.org/10.1145/2723872.2723882

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

Cohen-Boulakia, S., Belhajjame, K., Collin, O., Chopard, J., Froidevaux, C., Gaignard, A., Hinsen, K., Larmande, P., Le Bras, Y., Lemoine, F., Mareuil, F., Menager, H., Pradal, C., & Blanchet, C. (2017). Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. Future Generation Computer Systems, 75, 284–298. https://doi.org/10.1016/j.future.2017.01.012

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

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107–113. https://doi.org/10.1145/1327452.1327492

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., & Wenger, K. (2015). Pegasus, a workflow management system for science automation. Future Generation Computer Systems, 46, 17–35. https://doi.org/10.1016/j.future.2014.10.008

Di Tommaso, P., Chatzou, M., Floden, E. W., Prieto Barja, P., Palumbo, E., & Notredame, C. (2017). Nextflow enables reproducible computational workflows. Nature Biotechnology, 35(4), 316–319. https://doi.org/10.1038/nbt.3820

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

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., & Myers, J. (2007). Examining the challenges of scientific workflows. Computer, 40(12), 24–32. https://doi.org/10.1109/MC.2007.421

Goecks, J., Nekrutenko, A., & Taylor, J. (2010). Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology, 11(8), R86. https://doi.org/10.1186/gb-2010-11-8-r86

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

Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 59–72. https://doi.org/10.1145/1272996.1273005

Kluyver, T., Ragan-Kelley, B., Perez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., & Willing, C. (2016). Jupyter Notebooks—A publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt (Eds.), Positioning and Power in Academic Publishing (pp. 87–90). IOS Press. https://doi.org/10.3233/978-1-61499-649-1-87

Koster, J., & Rahmann, S. (2012). Snakemake—A scalable bioinformatics workflow engine. Bioinformatics, 28(19), 2520–2522. https://doi.org/10.1093/bioinformatics/bts480

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. PLOS ONE, 12(5), e0177459. https://doi.org/10.1371/journal.pone.0177459

Ludascher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., & Zhao, Y. (2006). Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience, 18(10), 1039–1065. https://doi.org/10.1002/cpe.994

McKinney, W. (2010). Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a

Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., & Van den Bussche, J. (2011). The Open Provenance Model core specification (v1.1). Future Generation Computer Systems, 27(6), 743–756. https://doi.org/10.1016/j.future.2010.07.005

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., & Abadi, M. (2013). Naiad: A timely dataflow system. Proceedings of the 24th ACM Symposium on Operating Systems Principles, 439–455. https://doi.org/10.1145/2517349.2522738

Peng, R. D. (2011). Reproducible research in computational science. Science, 334(6060), 1226–1227. https://doi.org/10.1126/science.1213847

Perez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. Computing in Science & Engineering, 9(3), 21–29. https://doi.org/10.1109/MCSE.2007.53

Polyzotis, N., Roy, S., Whang, S. E., & Zinkevich, M. (2018). Data lifecycle challenges in production machine learning: A survey. ACM SIGMOD Record, 47(2), 17–28. https://doi.org/10.1145/3299887.3299891

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. PLOS Computational Biology, 9(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., & Grafberger, A. (2018). Automating large-scale data quality verification. Proceedings of the VLDB Endowment, 11(12), 1781–1794. https://doi.org/10.14778/3229863.3229867

Simmhan, Y. L., Plale, B., & Gannon, D. (2005). A survey of data provenance in e-science. ACM SIGMOD Record, 34(3), 31–36. https://doi.org/10.1145/1084805.1084812

Stodden, V., McNutt, M., Bailey, D. H., Deelman, E., Gil, Y., Hanson, B., Heroux, M. A., Ioannidis, J. P. A., & Taufer, M. (2016). Enhancing reproducibility for computational methods. Science, 354(6317), 1240–1241. https://doi.org/10.1126/science.aah6168

Topcuoglu, H., Hariri, S., & Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 13(3), 260–274. https://doi.org/10.1109/71.993206

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., Bouwman, J., Brookes, A. J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C. T., Finkers, R., … Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. Scientific Data, 3, 160018. https://doi.org/10.1038/sdata.2016.18

Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva de la Hidalga, A., Balcazar Vargas, M. P., Sufi, S., & Goble, C. (2013). The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic Acids Research, 41(W1), W557–W561. https://doi.org/10.1093/nar/gkt328

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache Spark: A unified engine for big data processing. Communications of the ACM, 59(11), 56–65. https://doi.org/10.1145/2934664