Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

StateScope: Verifying Training-Source Isolation in Mutable Machine Learning Models

SIQI LI

dunklerort1026@gmail.com

Research Article

Keywords:

Posted Date: November 24th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11822983/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

StateScope: Verifying Training-Source Isolation in Mutable Machine Learning Models

SIQI LI[1*]

[1]Changchun University, Changchun, China
ORCID: 0009-0008-1806-3823.

Corresponding author(s). E-mail(s): dunklerort1026@gmail.com;

Abstract

A model updated on permitted data can still retain coefficients, cached features, or trees constructed from excluded rows. Pipeline provenance and input-overlap checks leave unresolved whether native solvers consumed earlier state and whether recorded evidence still applies to the current model. We present StateScope, a runtime auditor for scikit-learn that connects reviewed native operation contracts to versioned source evidence. It validates state for prediction or continued training and maintains conservative source bounds when evidence is incomplete. Evaluation covered 196 independent native execution histories, yielding 2,092 derived audit queries. Under the stated trust assumptions, the Full-path rule made zero false determinate decisions. It resolved 97.7% of queries without injected faults or omissions. Six public workflows demonstrate supported single-process verification and identify unresolved associations across workers and scripts. With default full discovery, a product-review classification workflow took **4.59**× native runtime, while a customer-churn workflow used **6.94**× native peak working set. Registered-only discovery reduced cold audit-context entry from **8.423** to **0.098** seconds on an incremental-learning fixture. In the product-review workflow, it lowered the full-process runtime ratio from **3.00**× to **1.17**×. Decisions and supporting evidence matched full discovery on the tested paths.

Keywords: machine learning software; software quality assurance; training state; data provenance; runtime verification; state validation

1 Introduction

Data-use restrictions and held-out evaluation sets require a way to check whether excluded rows participated in constructing the model selected for use. This check becomes harder when models are updated in place: the final training batch need not reveal sources retained in earlier fitted components.

An incremental stochastic gradient descent (SGD) update on permitted rows can retain coefficients fitted on excluded data. Cached features can carry an earlier scaler's sources even after that scaler is refitted, and a warm-start forest keeps the sources of its old trees (Kapoor and Narayanan, 2023; Kaufman et al., 2012). ML engineering studies describe how reuse across pipeline stages complicates these checks (Amershi et al., 2019; Sculley et al., 2015). These cases raise a software quality question: how can historical training evidence be connected to the current fitted state of a mutable model?

Operator-level tools such as `mlinspect` and `ArgusEyes` expose dataflow and row overlap (Grafberger et al., 2022; Schelter et al., 2023, 2024). A pipeline edge alone, however, leaves open whether a native Cython solver consumed earlier parameters. Pipeline lineage also leaves open which fields must match for prediction and which additional state matters for continued training. When evidence is missing on one branch, a provenance trace without conservative source bounds cannot distinguish the remaining uncertainty from exposure verified along another branch.

StateScope addresses these gaps through native observation, purpose-specific state validation, and lower and upper source bounds (Green et al., 2007; Imieliński and Lipski, 1984; Schwartz et al., 2010). Its policy records construction participation: a zero-weight batch remains a source if the native solver consumes it and advances its state. The paper makes three contributions:

1. **Native source verification.** We verify historical training sources for mutable estimators by observing actual native consumption, fitted-state replacement, and links between recorded versions.
2. **Purpose-specific state validation.** We connect live-state checks for prediction and continued training to conditional guarantees for isolation and exposure, with complementary rules for incomplete evidence.
3. **Evaluation for offline auditing.** We evaluate the method against independent native histories, erroneous-edge controls, six public workflows, and staged resource measurements that characterize its use in budgeted offline audits.

Paper organization.

Sections 2 and 3 define the objective and method. Section 4 evaluates the checks and their resource costs. Sections 5–8 cover implications, threats, related work, and conclusions.

2 Motivating Example and Audit Objective

Suppose an SGD classifier first runs `partial_fit` on excluded rows `e` and then updates on permitted rows `t`. The second call starts from the earlier coefficients, so

the construction history includes both batches. A fresh non-warm `fit` on permitted raw inputs can replace those coefficients. If it instead uses features from an earlier scaler, the source of those features remains relevant.

Listing 1 illustrates the distinction between fitted state and its evidence. After the two updates the audit confirms exposure. The fresh fit confirms isolation. A new context sees the same object but abstains until it verifies the saved association. Authentic application identities keep the exclusion policy stable across row reordering or changes of representation.

Listing 1 Training, parameter replacement, and verified history recovery in StateScope.

```
from tempfile import TemporaryDirectory
from sklearn.datasets import make_classification
from sklearn.linear_model import SGDClassifier
from statescope import Audit, EvidenceArchive

X, y = make_classification(60, 4, n_redundant=0, random_state=7)
t, e, yt, ye = X[:40], X[40:], y[:40], y[40:]
excluded = e
# The v0.1.2 API uses 'assessment' for the excluded rows.
query = dict(assessment=excluded, purpose='prediction')

with TemporaryDirectory(dir='.') as path:
    archive = EvidenceArchive(path)
    with Audit(boundary='All fitting is observed') as a:
        for name, rows in [('t', t), ('e', e)]:
            ids = [f'{name}:{i}' for i in range(len(rows))]
            a.bind_members(rows, ids, source='trusted_raw')
        model = SGDClassifier(random_state=7)
        model.partial_fit(e, ye, classes=[0, 1])
        model.partial_fit(t, yt)
        retained = a.check(model, **query)['decision']
        model.fit(t, yt)
        replaced = a.check(model, **query)['decision']
        handle = archive.export(a, model)

    with Audit(boundary='Same observed object') as a:
        a.bind_members(e, [f'e:{i}' for i in range(20)])
        missing = a.check(model, **query)['decision']
        recovered = a.check_with_recovery(
            model, archive=archive, handle=handle, **query
        )['decision']

print(retained, replaced, missing, recovered)
```

2.1 Scope and Trust Assumptions

StateScope audits whether excluded rows participated in construction of the selected fitted version. The audit depends on three premises.

A1 (Authentic inputs and source declarations). The caller supplies truthful row identities through `bind_members`. An input declared `trusted_raw` has no fitted upstream producer. For a derived input, the audit requires verified producer evidence or a conservative bound on its possible sources.
A2 (Faithful operation contracts). Reviewed `scikit-learn` contracts identify the direct inputs actually consumed, the prior state used, and the dependencies removed by verified replacement. Relevant training and transfers occur within the observed `Audit` boundary. Unverified dependencies remain unresolved.

A3 (Valid association at query time). The live model matches the observed version on the identity and state required for its intended use. Content checks can miss an unobserved edit followed by restoration of identical values.

3 StateScope: Design and Method

StateScope observes training operations, records which earlier parameters, matrices, and fitted components they used, and verifies those links against the current model (Figure 1).

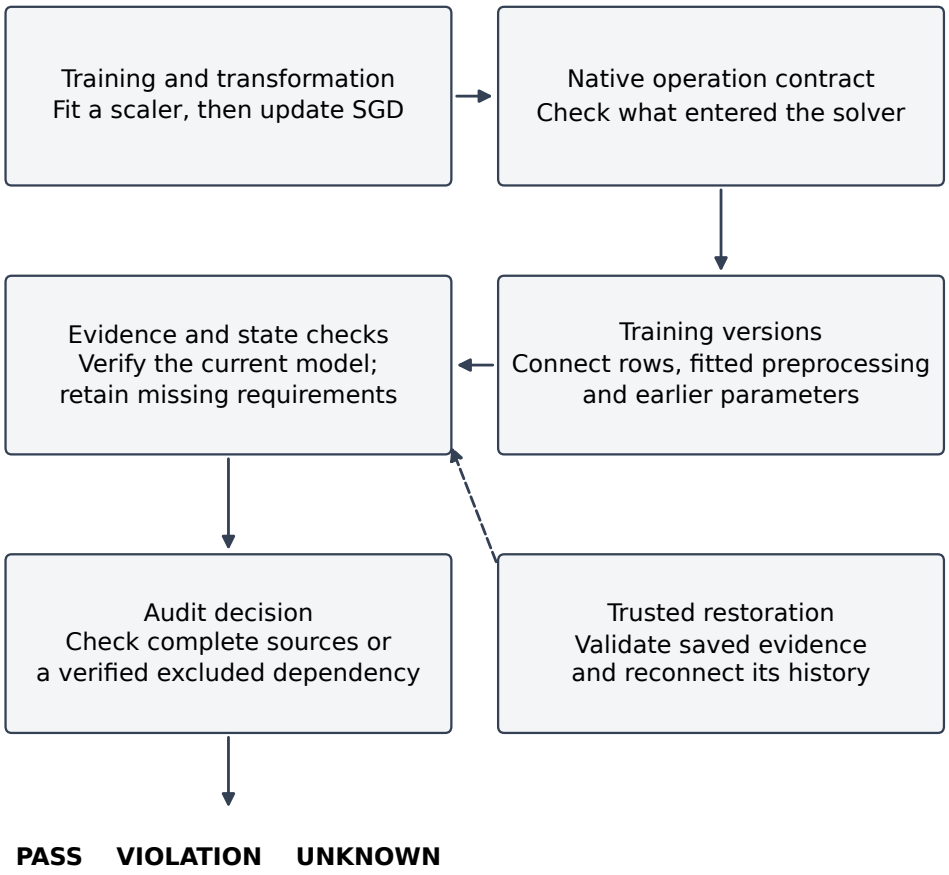

Fig. 1 Overview of StateScope from observed training to an audit of the current fitted model. Solid arrows follow runtime evidence used for an online decision; the dashed arrow reconnects validated checkpoint evidence via `EvidenceArchive`.

3.1 Training Sources and Operation Contracts

Table 1 Summary of formal notation.

Symbol	Meaning
o	Native construction operation.
$D(o)$	Identities of rows used directly by o.
$A(o)$	Earlier fitted versions whose state o uses.
$s,\ s'$	Fitted version and version produced by o, respectively.
$S(s)$	True set of construction sources for version s.
$L(s),\ U(s)$	Confirmed lower and conservative upper source bounds for s.
E	Identities of the excluded rows.
$\top$	Open upper bound with no verified finite restriction.

A native operation contract specifies the inputs and earlier state used to construct a version and how the observer identifies the result. Under the whole-batch policy, every row participates when the solver uses the batch. Python argument receipt and shape validation alone do not establish use.

Let $D(o)$ be the row identities directly used in the construction operation o, and let $A(o)$ be the earlier fitted versions whose state it actually uses. If the operation produces version s', the version's construction sources are

$$S(s') = D(o) \cup \bigcup_{u \in A(o)} S(u). \tag{1}$$

For example, an SGD update can use the previous coefficient array as well as the current batch. A fresh fit can replace the array, yet still use features generated by an earlier fitted transformer. Table 2 describes the checks that distinguish these cases.

Table 2 Checking which sources an SGD update retains in `SGDClassifier.partial_fit`. Each earlier dependency must satisfy its own contract.

Contract element	Evidence required by StateScope runtime observer
Direct input $D(o)$	Authentic ordered row identities for the batch entering native training.
Configuration	Binary `float64`, L2, integer seed, and fitted intercept; no averaging, warm start, or early stopping. Section 4.2 specifies the reviewed loss, step-size, and weight combinations.
Earlier state $A(o)$	Recorded incoming coefficient object reference and verified native solver use of that state.
Completed construction	Successful native solver entry and return; a zero gradient update does not erase batch participation.
Resulting version s'	Returned fitted estimator object, incremented generation counter, and learned parameter references.
Intended use	Prediction scoring fields or continued-training optimization state, as modeled for the configuration.
Replacement or gap	A complete raw-input `fit` cuts old estimator parameters; missing origin or unverified consumption remains unresolved.

When a coefficient vector initializes a new fit, its donor version's construction sources follow the import. This policy tracks participation at version granularity. A discarded donor contributes no dependency. Object references and logical generations distinguish successive versions of the same estimator, including the version that produced a cached matrix before a transformer was refitted.

3.2 State Validation for Prediction and Continued Training

StateScope records source use during native training and rechecks the selected version at query time. Both purposes share the construction history. Prediction compares the fields read by the reviewed scorer. Continued training also compares settings and state read by the next reviewed operation. Table 3 lists the exemptions for SGD and Boosting. Object and version association remain required even when two arrays have equal values.

Table 3 Model state fields required by contract-specified native execution paths for `prediction` versus `continued training`. A prediction exemption preserves all object identity, generation, and remaining parameter checks.

Estimator and state fields	Prediction	Continued training
SGD: `coef_`, `intercept_`, `classes_`, feature metadata (class labels for classifiers)	Compared	Compared
SGD: update counter (`t_`), `alpha`	Exempted on modeled paths	Compared
Boosting: `init_`, `estimators_` (trees), `learning_rate`	Compared	Compared
Boosting: internal RNG state (`_rng`)	Exempted on modeled paths	Compared

Before a fit, entry checks record what state can carry forward. At query time the auditor checks the object selected for use. Validating the returned array cannot reconstruct an unverified entry state. For example, an SGD counter change can preserve a checked prediction path while invalidating continued training. A path-level exposure witness links an excluded row to the live fitted version through verified uses. The Full-path rule in Section 3.4 can rely on this witness when actual coefficient use and the current association are independently verified.

3.3 Tracking Sources with Incomplete Evidence

Evidence about an input or an earlier fitted version can be missing on one branch of a model's history while another branch remains verifiable. In Figure 2, SGD updates on permitted rows B whose features come from a scaler with an unverified fitting history. The same update uses coefficients from a fit on excluded rows E. The coefficients and the features must be checked separately.

At each recorded version s, the evidence supports a set $L(s)$ of confirmed sources and a conservative range $U(s)$ of possible sources. The design invariant is

$$L(s) \subseteq S(s) \subseteq U(s). \tag{2}$$

An open upper bound ($U(s) = \top$) represents an unresolved relevant producer or dependency without a sound finite bound. Missing evidence adds no confirmed identities. A verified replacement removes only the dependency that ceased to be used. In Figure 2, a new fit may replace coefficients and still consume cached features from the earlier scaler. The missing scaler history leaves the upper bound open, while a verified coefficient route can support exposure if its association with the current model holds.

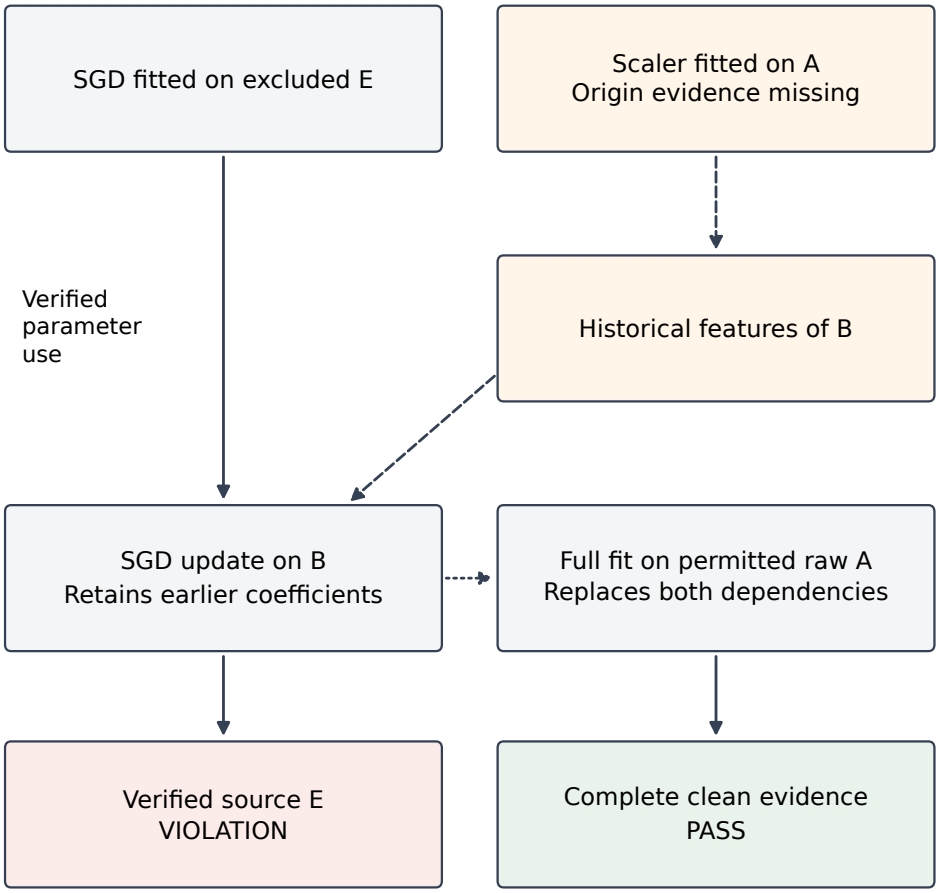

Fig. 2 Running example of an SGD model version s' with two consumed dependencies: a verified parameter retention route from excluded set E (solid arrows) and a feature transformation route whose scaler origin is missing (dashed arrows). A subsequent non-warm `fit` on permitted raw inputs A (dotted arrow to s'') severs both historical dependencies. If evidence for the coefficient route is also missing, the unresolved scaler origin leaves the decision at `UNKNOWN`.

3.4 Audit Decisions and Conditional Guarantees

Under Premises A1–A3 (Section 2.1), StateScope checks the fitted version selected for prediction or continued training against the excluded set E. The four rules below share the same operation evidence and participation policy. They differ in how they use an accepted audit log, verified field transitions, and independently verified input or dependency paths.

Whole-record (WR): the accepted audit log. A confirmed excluded identity in the log's lower bound establishes `VIOLATION`, even when the source upper bound is open. A valid, finite upper bound disjoint from E establishes `PASS`. If required policy or state checks fail, or the log supports neither decision, the rule returns `UNKNOWN`.

Field-aware Whole-record (FWR): verified field transitions. This rule also accepts independently verified transitions of supported configuration or optimizer fields, such as recorded `set_params` updates. It applies the same intended-use checks before deciding from the resulting audit log.

Direct-only (DO): a check of final-fit inputs. When the accepted audit log cannot establish exposure, this rule falls back to verifying the direct inputs consumed by the selected final fit.

Full-path (FP): surviving historical dependencies. This rule extends the direct check by following verified uses of earlier parameters, cached features, or fitted components. A validated path-level exposure witness can establish `VIOLATION` while evidence on an unrelated branch remains unresolved.

These rules are complementary. Verified field transitions can recover a decision unavailable to a path check, while a surviving historical dependency can confirm exposure when whole-record evidence is insufficient.

Conditional Soundness Argument

Base case and inductive preservation.

For an initial full fit on authenticated raw rows with no fitted predecessors, the participating identities establish $L(s) = S(s) = U(s)$. Order subsequent versions by creation, leaving cyclic or unverified dependencies unresolved. Under A1 and A2, the construction sources in Eq. (1) include the current direct inputs and sources of consumed state. Verified inputs and confirmed lower bounds of actually consumed predecessors therefore add only members of $S(s')$ to $L(s')$. An unresolved branch supplies no confirmed identities. The upper bound covers every relevant predecessor until its use can be ruled out, for instance by verified replacement. An unresolved producer without a sound finite bound leaves the upper bound open. These steps preserve Eq. (2).

Consequences for the four audit rules.

With A3, `PASS` requires complete coverage and a finite $U(s)$ disjoint from E. Equation (2) therefore excludes E from the true sources. A `VIOLATION` requires a verified excluded member of $L(s)$, or a separately validated chain from an authentic origin through each actual use to the version bound under A3. Either route establishes a member of $S(s)$ even when another branch remains unresolved. The rules abstain when neither form of evidence is sufficient.

Necessity of A2.

An erroneous edge can insert a non-participating source into $L(s)$. Section 4.2.5 isolates this failure in an injected version graph. The example shows why a graph path must be backed by evidence of actual state use.

Algorithm 1 summarizes the checks.

1. Select the recorded version for the model, exclusion policy, and intended use. Validate supported field transitions for Field-aware Whole-record.
2. Compute source bounds and check their applicability to the current model. Return `VIOLATION` for a confirmed excluded source in the accepted audit log.
3. For Direct-only or Full-path, check a direct excluded origin. Full-path also checks verified uses of earlier state. Return `VIOLATION` for a valid witness.
4. Return `PASS` if current association and complete coverage establish a finite upper bound disjoint from the exclusion set. Otherwise return `UNKNOWN`.

Algorithm 1: Auditing the fitted model selected for use

Trusted Restoration

`EvidenceArchive` validates the saved association against a live object or a supported file restore. Checks cover recorded structure, contents, and compatible contracts within a trusted archive and execution environment. Observed copies may establish a new association. Equal-valued arrays, shallow copies, and ordinary serialization alone do not supply one. An alias can invalidate a binding when detected. A change followed by restoration between observations can violate the observation premise.

3.5 Supported Operations

StateScope targets `scikit-learn` 1.8.0. Its `Audit` context temporarily wraps reviewed methods and selected native entry points, then removes the wrappers at exit. It observes the actual input and returned objects and distinguishes nested calls so an internal fit does not create a second account of the same construction. For each operation, the contract specifies which evidence can establish a new fitted version. The runtime keeps the relevant references, ordered member identities, structured state summaries, and dependencies from the operations it observes. It compares object identities, observed versions, types, and content digests when required by the contract. The digest tests whether supported contents match a prior observation. Authentic row identities and trusted observation supply the source association. The checks cover different fields for prediction and continued training (Table 3) and carry forward gaps for dependencies they cannot verify. Holding intermediate objects and evidence entries allows checking a consumed feature matrix or retained tree later, and can increase memory use. We establish contracts by reviewing which inputs and state the native implementation reads and by testing interventions on that execution. Table 4 separates those capabilities from the operator test suite, which evaluates 61 fixed operation profiles. The exposure and isolation results appear in Section 4.2.

Table 4 Functional support for reviewed profiles. C denotes verified support under the stated conditions. U denotes unverified support, including operations for which no applicable check is established. N/A indicates a capability that does not apply. Cont. denotes continued training.

Group	Sources	WR	Path witness	Prediction	Cont.	Restore
SGD classifier	C	C	C	C	C	C
SGD regressor	C	C	C	C	C	U
Gradient boosting classifier	C	C	C	C	C	C
Logistic regression	C	C	U	C	C	C
Random forest classifier	C	C	C	C	C	C
StandardScaler	C	C	C	N/A	C	C
TfidfVectorizer	C	C	C	N/A	C	C
HashingVectorizer	C	C	U	N/A	N/A	U
Other registered operations	39/54	U	U	U	U	U

The table specifies the state validation capabilities for the reviewed configurations. For the SGD classifier, Table 2 describes the path-level check of coefficient use under the settings tested in Section 4.2. The boosting check requires verified growth and retention of the earlier trees. The StandardScaler check follows a recorded copy of the transformed output to its actual consumer. For HashingVectorizer, Whole-record verification concerns the transformation, since it has no fitted source state. Other registered operations are included in the operator test suite and are unverified for the other checks unless individually reviewed. SGD regression paths are established for reviewed single-target squared-error updates. TF–IDF paths require reviewed inputs and a recorded, unchanged `float64` sparse output consumed by a downstream fit. Forest paths require unweighted binary growth, bootstrap disabled, and a fixed integer seed. The runtime checks the retained tree objects, their contents, and the class mapping. The restoration column covers supported audit logs. Cross-process restoration of path-level exposure witnesses in these three profiles is unverified.

Feature hashing learns neither a vocabulary nor IDF weights (`scikit-learn` Developers, n.d.). For the reviewed HashingVectorizer configuration, the runtime checks ordered document identities and the actual sparse output. Custom adapters, including row-preserving feature callbacks evaluated in Section 4.4, require individually reviewed contracts. A transform whose configuration or behavior is outside these checks can leave a source requirement unresolved.

4 Evaluation

4.1 Evaluation Design

Table 5 maps the four research questions to their evidence, scale, and evaluation sections.

Table 5 Evaluation overview: research questions, evidence, scale, and locations.

RQ	Evaluation design
RQ1	**Contract and graph soundness**. Operator test suite, SGD and purpose controls, parameter imports, bounded DAGs, and edge injections *Scale:* 61 profiles, 14 SGD configurations, 16 purpose conditions, 43,216 DAG configurations, 660 graph checks, including 20 wrong-edge injections, 108 base histories *Location:* 4.2, Tables 6–8
RQ2	**Incomplete-evidence auditing**. Native-reference replay with four rules and Explicit-Graph *Scale:* 196 independent histories, 2,092 derived audit queries *Location:* 4.3, Tables 9–10
RQ3	**Public workflows and tools**. Reviews, Churn, Taxi, Reuters, Dask, STREAMLINE, and four external tools *Scale:* Six public workflows and 36 common controlled cases *Location:* 4.4, Tables 11–14
RQ4	**Runtime, memory, and scaling**. End-to-End, Staged Scaling, and Cold-Start Optimization benchmarks *Scale:* 210 workflow runs, up to 262,144 rows or 1,000 updates, and 75 cold-start runs *Location:* 4.5, Tables 15–18

StateScope (v0.1.2) runs with scikit-learn 1.8.0 and Python 3.13.9. Independent native executions log row identities, solver inputs, retained components, predictions, and fitted arrays. Native labels stay outside the auditor. Matched evidence changes hold the model fixed, while fit or state interventions are repeated in the reference arm.

Each independent history trains a model along a fixed sequence. Uses, checkpoints, and evidence views from that run form derived audit queries (Runeson and Höst, 2009). We score each rule from Section 3.4 against the reference and report determinate coverage separately from false decisions. Explicit-Graph receives equivalent checked facts. External tools are assessed on tasks their interfaces express. The performance environment manifest reports Windows 11 (build 26200), an Intel Core i5-13500H (12 physical cores, 16 logical processors), and 16 GB of RAM. The archived run paths are on the system volume, which currently uses an NVMe SSD.

4.2 Reliability of Training-Source Verification

4.2.1 Operation Profile Coverage and SGD Configuration Sweep

The contract tests distinguish evidence of exposure from the complete source coverage needed to establish isolation. Across 61 operation profiles, the operator suite tests fresh exposure, retained exposure, and permitted training. For both prediction and continued training, it confirms exposure on 46 profiles in each exposed schedule and isolation on 44 profiles in the clean schedule. Two clean boosting refits return `UNKNOWN` because nested transformations lack producer coverage. The supported count therefore depends on the audit question.

The SGDClassifier sweep examines configuration changes within the reviewed native boundary. It varies sample or class weights, loss, and learning rate across 14 distinct configurations. In the retained-state cases, the native solver uses the earlier coefficient array. These observations apply to the exact checked binary, non-averaged configurations. The native incremental API rejects balanced class weights and early stopping. The component experiments in Section 4.3.2 use the same operation boundaries.

4.2.2 Native Construction Boundaries and Purpose-Aware Interventions

Construction participation can persist without a numerical change in the tested parameters. In the zero-gradient and all-zero-weight controls (Table 6), the native solver uses the supplied batches and advances its counter while leaving the fitted coefficients and intercept unchanged. Solver entry and batch mapping establish participation; the counter advance corroborates the call. A prediction check can omit that counter on a reviewed scoring path while retaining the batch in the construction history.

The boosting controls show why state rebuilt outside tree growth also needs a source contract. With zero growth, the classifier retains its trees but reconstructs class labels from the new target vector; the regressor has no such label step. In the API-valid relabelling test, predicted class labels change while the old trees remain. The current contract verifies tree-growth provenance but leaves the source link for rebuilt class labels unresolved, so both zero-growth checks return `UNKNOWN`. The SGD contract, in contrast, verifies consumption of the zero-weight batch and supports `VIOLATION` even when the coefficients stay fixed. Verified boosting growth on excluded rows also establishes exposure.

Table 6 Native construction boundary test results. Each query tests whether an excluded second batch is detected; `prediction` and `continued training` queries agree across all cases.

Operation	Native behavior	Contract interpretation	Decision
SGD, zero gradient	Native solver consumes batch; `coef_` and `intercept_` unchanged; counter `t_` advances.	Batch participates in construction.	`VIOLATION`
SGD, zero sample weights	Native solver receives zero weights; counter `t_` advances without coefficient change.	Whole-batch participation policy applies.	`VIOLATION`
Boosting classifier, no new trees	Trees remain. The native call reads the current labels and rebuilds `classes_`.	Current labels rebuild class state. The contract leaves their source link unresolved.	`UNKNOWN`
Boosting regressor, no new trees	Old trees remain. The observations leave new fitted state from the targets unestablished.	Contract condition unestablished.	`UNKNOWN`
Boosting classifier, tree growth	New stage trees constructed using supplied batch and residuals of earlier ensemble.	New batch and prior ensemble history retained.	`VIOLATION`

State validation must also match the intended use. Editing the SGD step counter or `alpha`, or the boosting random-number-generator state, leaves all tested dense and compressed sparse row prediction and decision-function outputs unchanged but alters continued training. The reviewed prediction profiles accept these states, while continued-training checks return `UNKNOWN`. Edits to coefficients, intercepts, or tree thresholds invalidate association for both purposes.

The comparison in Table 7 isolates these association decisions. Four estimator/source templates combine SGD or boosting with permitted or excluded training rows. Sixteen derived conditions cover unchanged states and edits to counters, `alpha`, random state, coefficients, intercepts, and tree values. Native controls compare the predictor and a fixed next update before each audit. Prediction-only compares prediction fields, Whole-state compares complete state, and Purpose-aware selects fields for the intended use. Accepting an association still requires separate source evidence before a source decision can be made.

Purpose-aware avoids the two errors made by the fixed comparison strategies in these controls. Whole-state rejects six prediction states whose edited counters, `alpha`, or random state preserve the predictor. Prediction-only accepts the same six conditions for continued training even though the next update changes. The public auditor preserves determinate source judgments for valid associations and abstains after edits to relevant state, with no false `PASS` or `VIOLATION` in these controls.

Table 7 State-association ablation, 16 derived conditions per purpose. Cont. denotes continued training. Errors refer to reviewed read paths and paired native execution.

Purpose	Comparison	Correct	False acceptance	Unnecessary rejection	Accepted
prediction	Prediction-only	16	0	0	10
prediction	Whole-state	10	0	6	4
prediction	Purpose-aware	16	0	0	10
Cont.	Prediction-only	10	6	0	10
Cont.	Whole-state	16	0	0	4
Cont.	Purpose-aware	16	0	0	4

4.2.3 Parameter Imports and Bounded Specification Conformance

Consumed donor state determines which history follows a parameter import. The checks in Table 8 trace the donor associated with each array used by the solver. Importing a clean coefficient while leaving an exposed intercept unused gives `PASS`; consuming both gives `VIOLATION`. A coefficient from a mixed donor carries the donor version's full history. An unobserved equal-valued copy yields `UNKNOWN` because its origin is unverified.

The graph tests separately check propagation and decision logic given operation facts. An independent index-based interpreter evaluates all 75 ordered directed acyclic graphs with one to four versions. An immediate predecessor can supply retained receiver state; other edges represent earlier feature use or parameter import. The configurations vary the excluded origin, evidence gaps, initial input mapping, selected version, and current association. Across 43,216 configurations, reachability, source bounds, and verified-path existence agree with the reference. The selected version is one of the last two where applicable, and the excluded origin is one version or none.

Table 8 Parameter import evaluation under version-level construction sources. All contract-based variants and both audit purposes agree across these scenarios.

Native Parameter Selection Scenario	Relevant Construction Origin Tracked	Audit Decision
Clean `coef_` imported; excluded donor component unused	Clean donor version and new training batch	`PASS`
Clean `coef_` and excluded `intercept_` both imported and used	Both donor version histories	`VIOLATION`
`coef_` imported from a hybrid-origin (clean + excluded) fitted version	Whole recorded donor version ancestry	`VIOLATION`
Equal-valued unobserved `coef_` array copy assigned in memory	Origin unverified (no registered version link)	`UNKNOWN`

This bounded conformance check covers graphs with $|V| \leq 4$ under trusted operation facts. The native tests examine how those facts are obtained. A separate Explicit-Graph replay covers six controlled Churn and Taxi histories with competing

dependencies, evidence omission, and clean replacement. The implementations agree on 86 derived checkpoint, purpose, and restoration observations. Their traversal code is independent, while the native operation contracts and state comparisons are shared.

4.2.4 Complete Histories and Producer Coverage Controls

Complete audit logs provide a common baseline on which all four rules agree with the native reference. The 108 independently constructed base histories exercise feature production, estimator reuse, and permitted training. For each of two fixed seeds, permitted preprocessing and model-training batches are separated from excluded rows and independent behavioral probes. Reviews uses cached TF–IDF; Churn uses scaling with incremental SGD or warm-start forests; Taxi uses scaling and incremental regression; Dask uses feature hashing and incremental SGD.

The 84 derived conditions then add a refit, remove evidence, or intervene on the final model. A failed current association leads to abstention. The paired queries contain no false isolation or exposure decisions. Agreement with complete logs checks the operation and source handling shared by the rules.

Producer controls test a different boundary: transformed row identities do not identify the data used to fit their producer. When an external scaler has fitted on excluded rows, supplying only the transformed rows' identities leaves that origin unresolved. Trusted restoration reconnects the producer and supports the decision for its recorded origin. A separate verified exposure route can remain usable while another producer is unverified. All 60 matrix and archive controls, covering rebinding, aliases, copies, and missing evidence, meet their expected outcomes under the declared trust conditions. Falsely declaring the same derived features raw leads to `PASS` despite the excluded scaler fit, demonstrating the dependence on truthful source declarations.

4.2.5 Erroneous Version-Edge Injection

An incorrect consumption edge can invalidate an otherwise consistent graph calculation. The suite contains 660 structured and generated graph checks, including 20 wrong-edge injections. None of the other 640 checks produces a false determinate decision. In the injections, a false edge adds a source outside the reference history to $L(s)$, causing 5 false `VIOLATION` decisions.

The minimal counterexample has 2 nodes: a selected version with permitted sources and a separate version with an excluded source. A fabricated dependency transfers the excluded source into the selected version's lower bound. The test supplies an invalid consumption link and illustrates the role of Premise A2. These finite checks establish conformance on the enumerated facts; Section 3.4 provides the conditional argument.

Takeaway (RQ1).

The native tests support the recorded construction decisions on reviewed paths when producer and state-association evidence is available. The edge injections identify a concrete failure when operation facts violate A2.

4.3 Audit Decisions under Incomplete Evidence

4.3.1 When Historical Dependencies Matter

A model can still use earlier state after the auditor can no longer validate the complete audit log. In H2 and H3, the next SGD update receives earlier coefficients even though another entry field has changed. In H7 the imported coefficients connect the receiver of the array to the version that produced it. In H8 an independently verified scaler produced features the learner used, while an unrelated input requirement remains unresolved. The selected version and its purpose checks still hold on each of these four paths. H1 in Table 9 illustrates a different case because the final update supplies the evidence and Direct-only succeeds.

Table 9 Representative histories with incomplete evidence. P, V, and U denote PASS, VIOLATION, and UNKNOWN. The two purpose queries agree within each history. WR, FWR, DO, and FP are defined in Section 3.4.

ID	Evidence condition	Source	WR	FWR	DO	FP
H1	Excluded final update after prior replacement	Exposed	U	U	V	V
H2	Changed counter before update	Exposed	U	U	U	V
H3	Changed intercept before update	Exposed	U	U	U	V
H4	Unobserved clean replacement	Clean	U	U	U	U
H5	Edited coefficients before update	Exposed	U	U	U	U
H6	Verified clean raw-input refit	Clean	P	P	P	P
H7	Imported coefficient followed by update	Exposed	U	U	U	V
H8	Excluded scaler history with unrelated missing evidence	Exposed	U	U	U	V
H9	Changed random state before tree growth	Exposed	U	U	U	U
H10	Unobserved fitted-state replacement	Clean	U	U	U	U
H11	No new trees and rebuilt class labels	Exposed	U	U	U	U
H12	Missing input identity on final update	Exposed	U	U	U	U

A path still requires current state and source evidence. The edited coefficients in H5 and missing rows in H12 block confirmation. The unobserved replacements in H4 and H10 lack a trustworthy association. H6 verifies a raw-input refit and ends the earlier dependencies. Field-aware Whole-record and Whole-record agree under the shared evidence. An independently observed H2 counter change can restore Whole-record confirmation.

4.3.2 Native Reference, Abstention, and Rule Complementarity

Full-path returned no incorrect determinate decision in the native-reference corpus. The evaluation covered 196 independent native execution histories and 2,092 derived queries across purposes, checkpoints, and evidence views. Its determinate coverage was 42.2%; Table 10A reports the decision breakdown. Given equivalent checked facts, Explicit-Graph agreed on all 2,092 queries. This comparison checks decision consistency using shared native contracts and state comparisons. Obtaining validated native evidence and binding it to the current model remain separate engineering requirements.

Table 10 Rule outcomes, nested evaluation subsets without injections, and causes of abstention. All panels use the same native-reference query corpus. EG denotes Explicit-Graph.

A. Native-reference decisions

Rule	PASS	VIOLATION	UNKNOWN	False determinate
WR	350	422	1,320	0
FWR	360	430	1,302	0
DO	350	523	1,219	0
FP	350	533	1,209	0
EG	350	533	1,209	0

B. Coverage without injections

Subset	Histories	Determinate / total	UNKNOWN	Coverage
All unperturbed queries	76	346/354	8	97.7%
Queries on unperturbed terminal models	76	150/152	2	98.7%
Supported terminal models with complete producer coverage	73	146/146	0	100%

C. Causes of FP abstention

Cause	Queries	Share of UNKNOWN
Controlled fault or omission injection only	896	74.1%
Injection and capability limit	298	24.6%
Capability limit only	15	1.2%

Decision shares and producer coverage. For Full-path, `PASS`, `VIOLATION`, and `UNKNOWN` represent 16.7%, 25.5%, and 57.8% of all queries, respectively. The 6 unperturbed terminal queries outside complete producer coverage come from 3 Boosting histories with open upper bounds. Verified lower bounds support 4 exposure decisions. The remaining 2 terminal queries and 6 additional policy rechecks return `UNKNOWN`, despite valid terminal associations, because no excluded source is confirmed.

The high overall abstention rate mainly reflects controlled evidence disruptions in this corpus. Fault or omission injections contributed to 1,194 of the 1,209 abstentions (98.8%; Table 10C). Without injections, determinate coverage was 97.7%. All 146 supported terminal queries with complete producer coverage were determinate. Table 10B distinguishes this subset from the broader sets of unperturbed terminal models and unperturbed queries. Evidence omission was the primary cause for 573 queries, including 301 with a matched determinate answer before omission.

The rules complement one another by recovering different queries. Full-path confirmed 111 more exposure queries than Whole-record and 10 more than Direct-only. In 6 queries from 2 independent histories, the other three audit rules abstained. Of those queries, 5 revisited a version already confirmed exposed. Across all tested conditions within each history, the number of independent histories whose exposure was confirmed only by Full-path was 0. Field-aware Whole-record recovered 10 isolation decisions where Full-path abstained by validating supported configuration transitions.

Takeaway (RQ2).

Whole-record evidence resolves many complete cases. Verified configuration changes can restore isolation, and independently verified paths can establish exposure when whole-record checks abstain. The benefit of each rule depends on which evidence remains available.

4.4 Evaluation on Public ML Workflows

4.4.1 Workflow Suite and Supported End-to-End Pipelines

The public workflows test whether the auditor can connect application-side training to a selected fitted model. Table 11 records the evidence obtained and the application glue needed to bind row identities and open an `Audit` context. Integration takes 6–18 lines once reusable contracts are available. Developing a custom contract, such as the Taxi date callback, is a separate library engineering task.

Table 11 Public workflow execution and application integration. Counts refer to the original application where available. The Dask line count refers to synchronous adaptation.

Workflow	Rows	Native fits	Audit evidence or boundary	Glue (LOC)
Reviews	8,510	3	TF–IDF and selected fitted classifiers.	18
Churn	7,043	410	Model selection and verified fitted models. The 410 calls include 328 preprocessing fits.	18
Taxi	7,252 train	1 pipeline	Original nested date callback, supported by a reviewed row-preserving contract.	6
Dask	11,314 train	25 updates	Native worker updates. Final client association unresolved in distributed execution.	18
Reuters	17,872 train	20 updates	Sequential training, retained coefficients and selected model.	14
STREAMLINE	132	—	Six fold identity checks. Saved components cannot be associated with their earlier fits.	—

The supported single-process cases connect fitted components to their construction histories. Reviews links selected TF–IDF classifiers to training rows. Churn establishes isolation for live queries. An ordinary `joblib` load leaves the model association unverified; a trusted restore re-establishes it for both uses. Controlled branches separately test cached-feature reuse on Reviews and retained trees on Churn. In the original Reuters program, all rules agree under clean and exposed policies after 20 sequential SGD updates.

The Taxi case shows why overlap checks must follow trip identities through a join. The fare regressor (Schelter et al., 2023) joins 7,252 training trips to a spatial lookup. A modern join changes the positions of 7,251 training trips, while identity matching preserves the reference overlap. The inclusive and strict temporal splits share 177 and zero training trips, respectively (Table 12).

Table 12 Audit decisions on the NYC Taxi regression pipeline (7,252 training trips) across three pipeline integration modes under the inclusive temporal split (177 shared train/test trips; ground truth: exposed) and strict temporal split (0 shared trips; ground truth: isolated). Decisions hold across both purposes and all contract-based variants.

Pipeline Integration Mode	**Inclusive Temporal Split (177 shared trips; 2,912 test trips)**	**Strict Temporal Split (0 shared trips; 2,735 test trips)**	**Preserves Original Pipeline Topology?**
1. Original topology, standard capture (unmodeled `FunctionTransformer`)	`UNKNOWN`	`UNKNOWN`	Yes
2. Dates computed before capture (manual pipeline refactoring)	`VIOLATION`	`PASS`	No (requires code edit)
3. Original topology, contract-specified callback (StateScope restricted contract)	`VIOLATION`	`PASS`	**Yes**

A reviewed row-preserving date adapter supports the original nested callback without changing its features, coefficients, intercepts, or predictions in the same environment. The adapter rejects thirteen invalid entry controls. A learned-coefficient edit makes the rules abstain for both purposes. The ArgusEyes comparison addresses direct train–test overlap on the two splits and agrees with the independent trip counts. A separate controlled branch tests retained coefficients in fare regression.

4.4.2 Unresolved Execution Boundaries

Worker-side observations leave a gap when they cannot be associated with the final client model. The distributed Dask notebook, run with a string-metadata compatibility setting, records 25 native worker updates scheduled through the Dask interfaces (Dask Developers, n.d.; Dask-ML Developers, n.d.). The final client association remains unverified. Independent distributed runs also differ in fitted values, preventing an exact paired comparison between those executions.

Persistence across scripts creates a similar association problem in STREAMLINE. The 11-stage binary survival-prediction workflow (Urbanowicz et al., 2023) produces six fold views of the same 132 rows, each with disjoint training and test sets of 88 and 44 rows. Custom imputation maps, PCA wrappers, and 18 base models are saved across stage scripts. The evidence establishes fold membership but does not reconnect the saved components to their earlier fits, leaving the end-to-end audit unresolved.

4.4.3 Sensitivity to Post-Training Evidence Omission

Removing an essential evidence link can block a decision even when training is held fixed. The matched experiments omit selected evidence entries while preserving the model and source policy. They use synchronous Dask, the Reuters training loop, cached Reviews features after a refit, and growth of a no-bootstrap Churn forest.

Table 13 Decisions after training is frozen. The paired results agree for prediction and continued training. The clean histories give `PASS` with complete evidence. EG denotes Explicit-Graph.

Evidence condition	WR and DO	FP and EG
Exposed, complete audit log or unrelated evidence entry omitted	VIOLATION	VIOLATION
Exposed, evidence omitted for the origin, middle, or final fit	UNKNOWN	UNKNOWN
Clean, unrelated evidence entry omitted	UNKNOWN	UNKNOWN
Verified clean replacement	PASS	PASS

None of the four matched settings in Table 13 yields an exposure decision from one rule while the others abstain. Missing evidence for an origin, an actual use, or the final state can block confirmation. Counts of verified transfers describe the extent of the recorded history; the remaining usable links determine the decision.

4.4.4 Scope Comparison with External Tools

External tools are compared on shared tasks that their interfaces can express. On the common Taxi, Reviews, and Churn benchmark, the four StateScope rules agree on 36 controlled cases. For each purpose they identify 12 clean and 18 exposed cases, and abstain on six essential producer-evidence omissions. Table 14 reports the corresponding tool tasks. Direct row overlap covers the selected fit's inputs, while earlier coefficients or preprocessing can carry excluded sources into a disjoint final batch.

The `noWorkflow` trials record script and package dependencies (Pimentel et al., 2017). Notebook trials exercise ordinary training, excluded preprocessing, retained SGD state, and stale downstream state. Static analyzers such as Yang et al. (2022) target pre-execution abstract-syntax-tree dataflow; Table 19 places those documented capabilities alongside the runtime checks. The ArgusEyes execution uses a supported scikit-learn release with unused Keras hooks omitted. The noWorkflow execution uses an import-path fallback for cross-volume scripts.

Table 14 Shared tasks and source-evidence observations. Tools have different outputs and are evaluated on separately stated tasks.

Tool	Task	Result
ArgusEyes	Direct train–test overlap in original Taxi topology	177 inclusive, zero strict
mlinspect	Direct row origins on common data	36 matched lineage results
noWorkflow	Execution dependencies in flat scripts	Four recorded scenario trials
NBSafety	Stale dependencies in ordered notebook cells	Rerun marked downstream uses stale

4.5 Runtime and Resource Costs

Measurement design.

The three benchmarks answer different cost questions. The End-to-End Benchmark measures post-startup workflow time, peak process working set, and separate uncached queries (Section 4.5.1). The Staged Scaling Benchmark separates lifecycle phases and varies batch size and update depth, with distinct encoding-only and file-persistence protocols (Section 4.5.2). The Cold-Start Optimization Benchmark includes launch and cold startup when comparing discovery modes (Section 4.5.3). Ratios are interpreted within their matched timing windows.

4.5.1 End-to-End Workflow Overhead

The end-to-end comparison measures the cost of capturing evidence during a fixed workflow. Table 15 includes 210 valid runs, with 15 fresh processes per mode. Mode order is randomized within blocks, and synchronous Dask uses a fixed partition order. Outcomes, predictions, and fitted arrays are checked against the reference after timing. Peak memory denotes the process working set.

Table 15 End-to-End Benchmark. Time and peak working set are medians with interquartile ranges. Evidence is median serialized size. Ref. denotes the independent native reference observer. EG denotes Explicit-Graph. A dash indicates no retained audit evidence.

15 independent fresh processes per mode; values are median (IQR).

Reviews

Mode	Time (s)	Peak (MiB)	Evidence (MiB)
Bare native	2.19 (0.19)	342.96 (0.49)	—
Native + ref.	11.34 (1.79)	343.16 (0.44)	—
EG + ref.	11.75 (1.35)	349.58 (0.55)	0.78
StateScope	10.18 (1.09)	739.17 (0.44)	0.95
StateScope + ref.	19.31 (1.21)	739.14 (0.40)	0.95

Churn

Mode	Time (s)	Peak (MiB)	Evidence (MiB)
Bare native	33.02 (5.97)	213.66 (0.52)	—
Native + ref.	61.32 (8.22)	683.69 (1.85)	—
EG + ref.	65.36 (6.71)	1,500.42 (1.10)	127.04
StateScope	78.48 (12.58)	1,483.43 (1.87)	102.07
StateScope + ref.	161.87 (25.69)	1,967.60 (0.52)	102.07

Dask (sync.)

Mode	Time (s)	Peak (MiB)	Evidence (MiB)
Bare native	7.47 (1.81)	298.16 (0.44)	—
Native + ref.	7.55 (1.53)	381.92 (1.27)	—
StateScope	16.77 (2.84)	755.55 (1.16)	2.36
StateScope + ref.	16.85 (2.81)	756.33 (1.68)	2.36

The within-block median runtime and peak-memory multipliers vary by workload. On Reviews they are 4.59× and 2.16× bare native, respectively. On Churn they are 2.35× and 6.94×. On synchronous Dask they are 2.26× and 2.53×.

Churn's separate marginal peak medians rise from 213.66 MiB to 1,483.43 MiB. Retained fitted objects, features, identity maps, and evidence entries contribute to the auditor's live state. Figure 3 shows the placement and dispersion of all measured modes on a common bare-native scale.

Uncached queries isolate the decision step from training and capture. Table 16 compares rules on the same fixed evidence and model within each captured context.

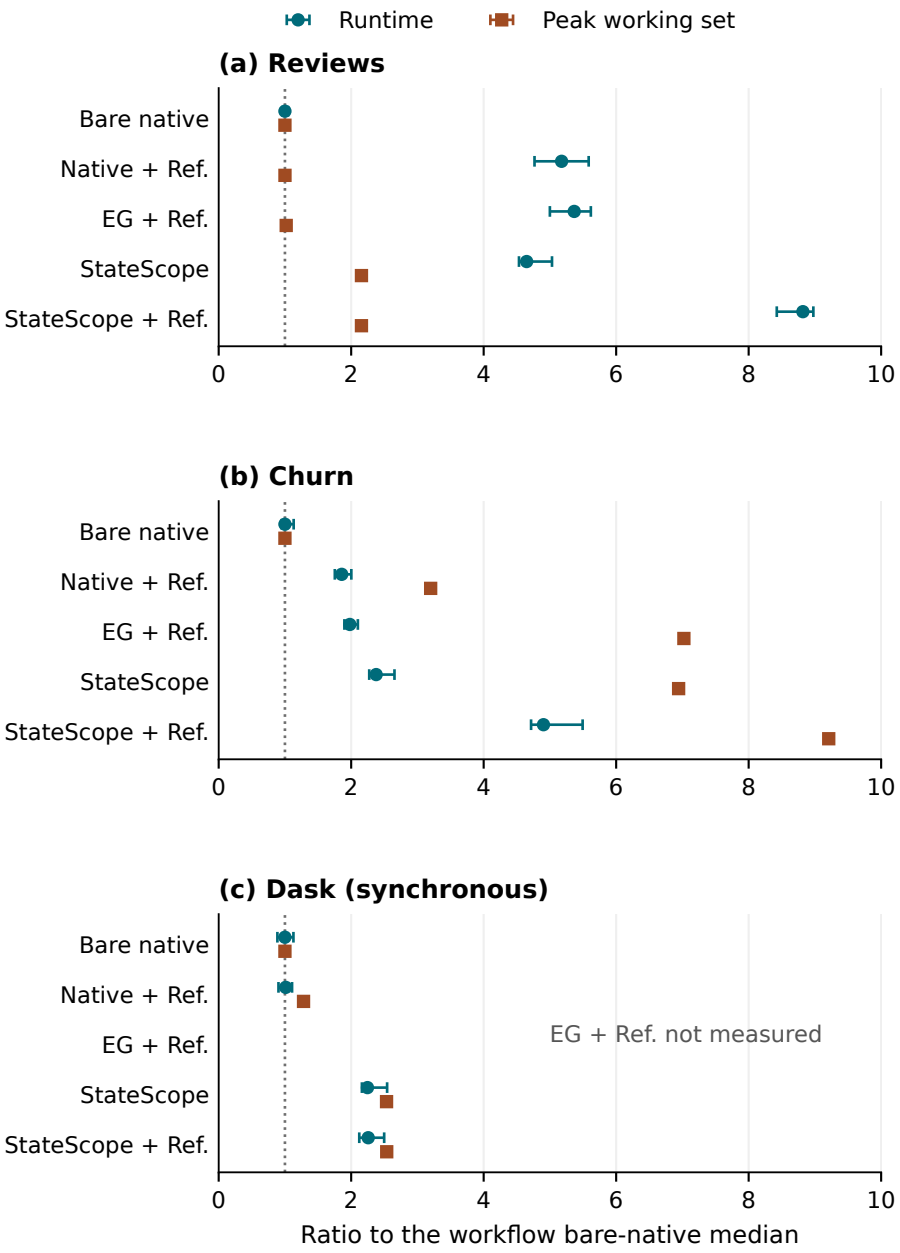

Fig. 3 End-to-end time and peak process working set relative to the bare-native median for each workflow; each panel uses 15 processes per mode. Markers and bars show mode medians and interquartile ranges on that common scale. EG denotes Explicit-Graph, and Ref. denotes the independent native reference observer. The text reports within-block median ratios.

Table 16 Uncached query time in milliseconds, median with interquartile range in parentheses. Each result combines 30 measurements from each of five independently captured contexts. All rules check the same evidence and model within each context. Cont. denotes continued training.

Dependency	Use	WR	FWR	DO	FP
SGD classification coefficients	Prediction	0.90 (0.24)	1.08 (0.32)	1.01 (0.39)	1.08 (0.38)
	Cont.	0.85 (0.19)	1.01 (0.27)	0.97 (0.37)	0.95 (0.31)
SGD regression coefficients	Prediction	0.83 (0.21)	1.01 (0.25)	0.95 (0.17)	0.94 (0.24)
	Cont.	0.80 (0.13)	0.95 (0.28)	0.91 (0.22)	0.89 (0.25)
Historical TF–IDF features	Prediction	1.23 (0.51)	1.56 (0.50)	1.40 (0.51)	1.39 (0.44)
	Cont.	1.15 (0.35)	1.44 (0.57)	1.30 (0.37)	1.32 (0.46)
Retained forest trees	Prediction	3.20 (0.60)	4.25 (0.88)	3.65 (0.76)	3.60 (0.75)
	Cont.	3.12 (0.73)	4.27 (0.65)	3.62 (0.68)	3.57 (0.66)

4.5.2 Staged Latency and Scaling Behavior

Context entry dominates the short audited tasks, while subsequent costs grow with batch size and history depth. The encoding-only sweep uses 75 valid paired blocks and counts serialized JSON evidence bytes. A separate protocol adds a buffered file write and close, using 45 fresh-process pairs across 9 settings. Table 17 selects three persistence settings with 5 pairs each; T_3 is the dominant phase in these settings.

Table 17 Staged Scaling Benchmark for SGD with file persistence, median times in milliseconds. The audited total extends from process launch through context release.

Phase / measurement	256 × 5	262,144 × 5	256 × 1,000
T_1: Launch and imports	2,516.01	2,504.66	2,466.05
T_2: Data preparation	0.43	313.95	58.99
T_3: Audit context entry	8,729.43	8,069.48	7,850.36
T_4: Bare-native training	5.72	335.41	562.86
T_5: Audited training and capture	23.18	936.84	2,032.00
T_6: Query	3.20	1,767.41	357.64
T_7: Encode, write and close	2.10	1,429.73	469.03
Audited total	11,500.71	15,015.91	13,655.07

The audited total excludes T_4, measured in a separate process. It includes query preparation and context release. Separate phase medians need not sum to the median total. All values are milliseconds; five matched pairs per setting.

After entry, larger batches increase query and serialization time, while deeper histories increase training and capture costs. In the persistence protocol, the median query takes 1,767.41 ms for 262,144 rows and 5 updates. It takes 357.64 ms for 256 rows and 1,000 updates. Figures 4 and 5 show phase composition and resource trends under the separate encoding-only protocol.

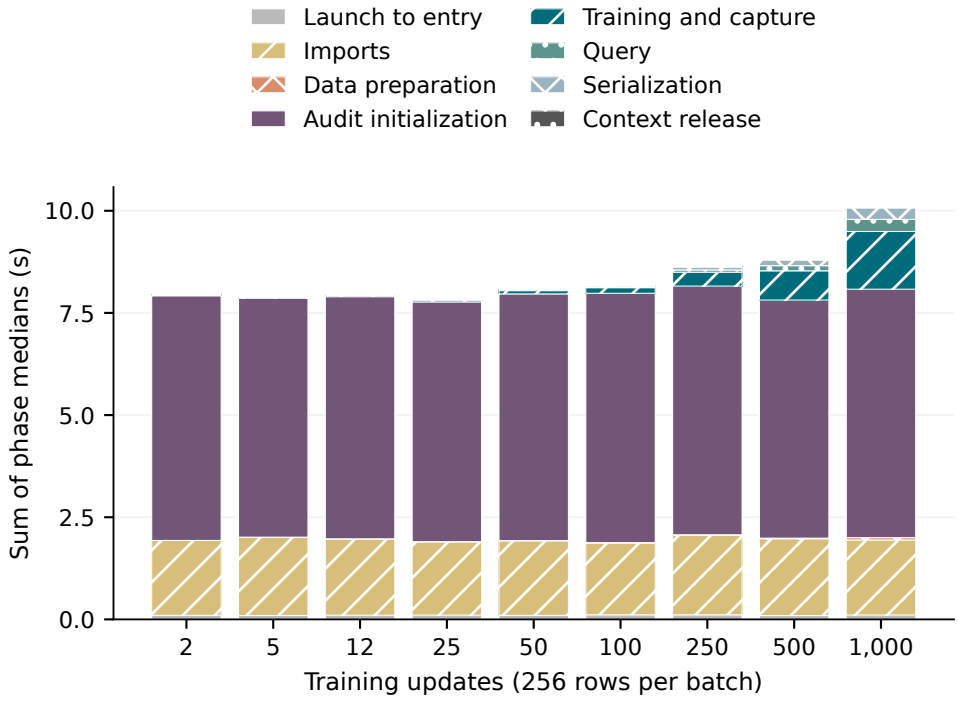

Fig. 4 Phase times by update depth at a fixed batch size in the encoding-only scaling sweep. Stack heights sum separate phase medians. Serialization covers JSON encoding and byte counting. Five complete blocks contribute to each setting; patterns distinguish lifecycle phases.

Paired training differences separate audited capture from bare-native fitting within each persistence block. The median $T_5 - T_4$ difference is 17.31 ms at 256 rows and 5 updates. It is 604.48 ms at 262,144 rows and 5 updates. At 256 rows and 1,000 updates, it reaches 1,365.65 ms (IQR 144.57 ms).

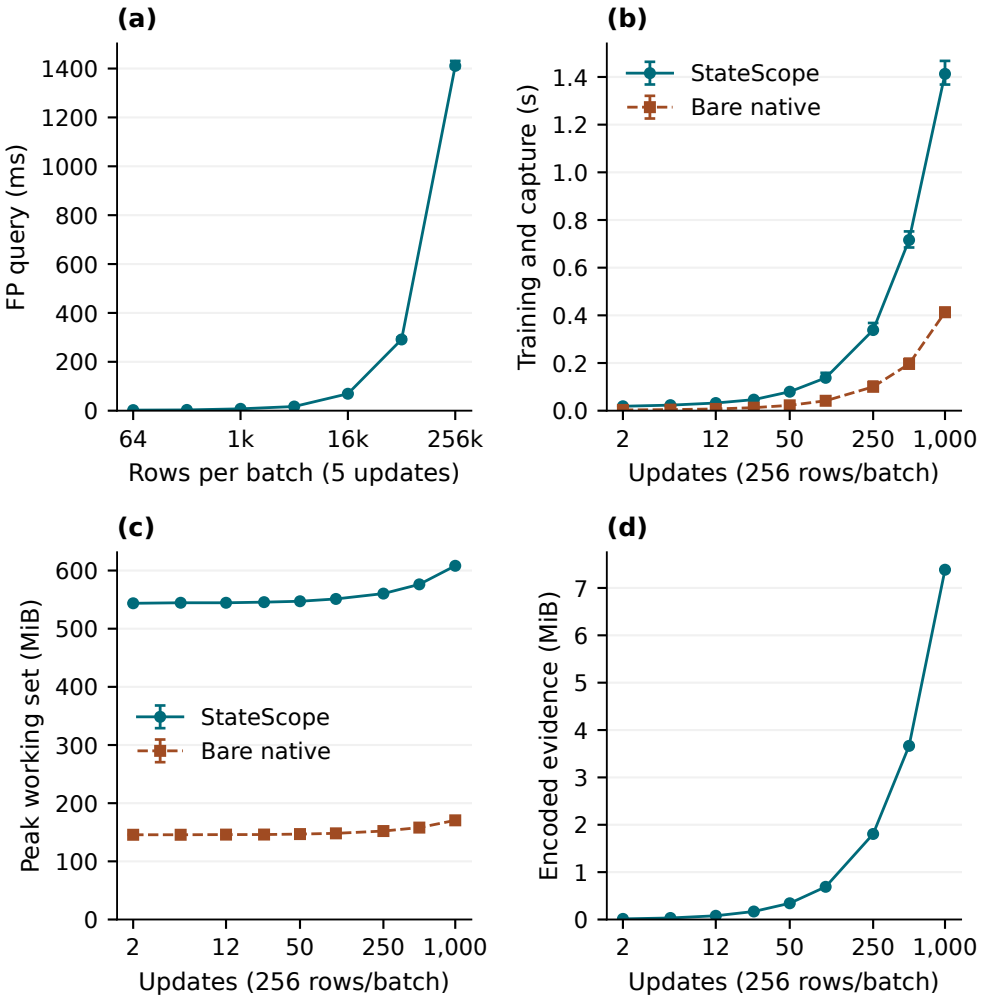

Fig. 5 SGD scaling in the encoding-only sweep. (a) Full-path prediction query latency by batch size at a fixed update depth. (b) Audited training and capture versus bare-native training by update depth. (c) Peak process working set and (d) encoded evidence volume by update depth. Markers and bars denote medians and IQRs across five complete blocks per setting. Horizontal axes use logarithmic scales.

4.5.3 Cold-Start Root-Cause Diagnosis and Mitigation

Profiling the SGD fixture locates most cold context-entry time in estimator discovery. The median `all_estimators()` call takes 9.452 s, within an instrumented entry median of 9.633 s. Discovery imports libraries unused by the fixture: median import times are 7.774 s for PyTorch and 0.970 s for `dask.array`. Wrapper installation takes a median 4.74 ms across 309 calls per run. Separate uninstrumented probes measure cold entry at 8.492 s and warm re-entry at about 79 ms.

Registered-only discovery avoids full estimator enumeration by installing hooks for operators already in the contract registry. The comparison uses Reviews and a 256-row, 5-update SGD fixture, with 15 randomized fresh-process blocks per workload. All 75 formal runs are valid, with no failures or reruns. Table 18 covers launch through task completion, including cold startup. The earlier Table 15 covers the post-startup workflow window, where Reviews has a 4.59× runtime ratio.

Table 18 Cold-Start Optimization Benchmark with 15 matched fresh-process blocks per workload. Times and ratios are medians. Reviews ratios divide elapsed time by bare-native time within each block.

Measurement	**Full discovery**	**Registered-only discovery**
Cold context entry, SGD (s)	8.423	0.098
Reviews runtime / bare native	3.00×	1.17×
Audit decisions and evidence entries	Reference	90/90 queries, 225/225 entries

The optimization reduces measured cost while preserving the tested evidence. On Reviews, the median paired runtime reduction is 62.85%, with a 95% block-bootstrap interval of 60.39–63.57%. Median peak working set falls from 739.06 to 352.41 MiB. Matched checks cover row identities, fitted-version associations, source bounds, and dependency witnesses.

5 Discussion

5.1 Selecting the Right Audit Rule in Practice

Choose the rule according to the evidence available for the intended use. For routine continuous-integration checks with complete logs, start with Whole-record. A verified upper bound can establish isolation after a clean refit. Field-aware Whole-record is appropriate when supported configuration changes are intentional and their transitions are independently verified. Under either rule, a failed state association calls for checking the observation boundary or recovering a trusted checkpoint.

For an investigation with incomplete logs, Direct-only can confirm exposure from the selected final fit's own inputs. Full-path can additionally follow earlier coefficients, cached features, or retained trees to the selected fitted version. Its benefit is specific to the query and the surviving evidence. Whole-record and Field-aware Whole-record remain the routine choices for complete histories; an unresolved source branch can prevent `PASS` under every rule.

5.2 Positioning Construction Participation Against Unlearning and Privacy

Construction participation is a deterministic execution property. It asks whether a row or an earlier fitted dependency entered the construction of the model chosen for use. Machine unlearning asks whether a removal procedure produces a model consistent

with a counterfactual trained without the designated rows (Bourtoule et al., 2021). Differential privacy bounds the change in an output distribution when individual inputs change (Dwork, 2006). Influence functions use a continuous, gradient-based approximation to estimate how a training perturbation affects a prediction (Koh and Liang, 2017).

These approaches answer different audit questions. StateScope can serve as a deterministic gate in a compliance pipeline by identifying a verified exposure, verifying isolation under its premises, or routing incomplete evidence for investigation. A later compliance step can separately assess the effectiveness of unlearning or the strength of a privacy guarantee.

5.3 Engineering Trade-offs in Contract Maintenance and Deployment

Native contracts require maintenance as scikit-learn changes solver signatures, retained fields, or nested call patterns. Automated comparison of Python and Cython abstract syntax trees could flag interface and read-set changes, propose candidate updates, and target the regression tests most likely to detect drift. Native observations would still be needed to confirm what an operation consumes and retains for a reviewed configuration.

Churn's peak working set and the single-process scope favor offline or near-offline CI audits, where teams can budget memory and schedule checks for selected model builds. Registered-only discovery reduces cold startup for short jobs with reviewed operations. A team adopting it should check that scheduled workflows use those operations and treat unresolved producer coverage as an audit result.

5.4 Bridging Distributed and Cross-Process Boundaries

Dask leaves a gap between worker-side training observations and the client-side fitted model. Signing a serialized task graph could bind task identities and declared dependencies across workers. A worker-side provenance envelope could carry input identities, observed operation facts, the fitted generation, and a digest of the output so the client can validate its association. STREAMLINE would need a similar envelope bound to each persisted component and verified at the point of loading. Both extensions would also need to establish that the observed worker or stage actually consumed the state named in the envelope.

6 Threats to Validity

Construct validity.

The audit tracks whether rows participated in constructing a fitted version. The native reference observes direct input use and constructed or retained state for each reviewed operation. A batch can participate without changing the tested outputs. Quantifying its effect on an individual prediction requires an influence analysis.

Internal validity.

Native comparisons use separate executions and check fitted state and predictions under the same training protocol. Premises A1–A3 (Section 2.1) define the conditions for interpreting the checks. Unsupported callbacks, invisible state changes, and unverified file recovery can leave gaps in the evidence. An unobserved change followed by restoration of the original contents can escape a digest comparison.

External validity.

The contracts cover specific scikit-learn profiles. We selected public examples to exercise different mechanisms for constructing and transferring fitted state. The distributed and cross-script runs leave some associations unresolved. The registered-only discovery comparison covers Reviews and synthetic SGD on one Windows host. The extra entry points installed by full discovery went unused in those runs, so equivalent coverage on other paths requires separate validation.

Conclusion validity.

Uses, checkpoints, and evidence views from one run are derived audit queries. Coverage on controlled subsets applies to their specified producer and state conditions. We measured performance on a single Windows workstation in fresh randomized blocks, checking paired outputs before accepting runs. Measurement suites ran separately under different timing scopes, and we observed variation in initialization and sub-millisecond per-update capture times across suites, so we interpret overhead multipliers within matched paired blocks.

7 Related Work

7.1 Data Provenance and Incomplete Information

Data provenance formalisms characterize why and where an output tuple was derived and how derivations depend on input sources (Buneman et al., 2001; Cheney et al., 2009; Herschel et al., 2017). Provenance semirings distinguish alternative derivation paths from joint dependencies required together (Green et al., 2007). Incomplete-information database semantics and abstract interpretation use sound lower and upper bounds to preserve certainty along verified paths when parallel origins remain unknown (Cousot and Cousot, 1977; Imieliński and Lipski, 1984). StateScope applies these provenance ideas to a fitted model that can change in place. The observer checks which earlier arrays, fitted components, and features the native operation actually used, and whether that evidence still applies to the selected model.

7.2 ML Lineage Tracking and Data Leakage Screening

ModelDB and MLflow log experiments and artifacts (Vartak et al., 2016; Zaharia et al., 2018). Pipeline provenance tools trace operations and intermediate artifacts (Qadeib Alban et al., 2026a, 2026b; Schlegel and Sattler, 2025). `mlinspect` extracts operator-level row lineage, and `ArgusEyes` screens intermediate artifacts for issues including overlap (Grafberger et al., 2022; Schelter et al., 2023, 2024). Chapman et

al. (2024) describe provenance templates for individual operators. Yang et al. (2022) screen notebook dataflow before execution. `noWorkflow` captures Python execution dependencies (Pimentel et al., 2017), and NBSafety tracks stale notebook state (Macke et al., 2021).

Table 19 reports scope by design. A dash in the binding column indicates tasks without audit row identities. Automatic lineage applies to the supported steps.

Table 19 Documented capabilities relevant to training-source auditing. Yes indicates direct support in the cited scope, partial indicates a related capability, and a dash indicates that row-level binding is outside the documented task. Origin means row-level origin. State means checked live estimator state. Bound means incomplete-evidence source bounds. Use means distinct checks for prediction and continued training.

A. Documented verification capabilities

Tool	Mode	Origin	State	Bound	Use
MLflow / ModelDB	Artifacts	No	No	No	No
noWorkflow	Python runtime	No	No	No	No
mlinspect	Operator DAG	Yes	No	No	No
ArgusEyes	Pipeline screen	Partial	No	No	No
NBSafety	Notebook state	No	Partial	No	No
Yang et al. (2022)	Static dataflow	No	No	No	No
MLProvLens	Artifact lineage	No	No	No	No
StateScope	Runtime	Yes	Yes	Yes	Yes

B. Binding requirements and scope

Tool	Row binding	Scope
MLflow / ModelDB	—	Model artifacts
noWorkflow	—	Python scripts
mlinspect	Automatic on supported steps	Supported pipeline ops.
ArgusEyes	Automatic on screened steps	Pipeline checks
NBSafety	—	Notebook dependencies
Yang et al. (2022)	—	Notebook code
MLProvLens	—	Pipeline artifacts
StateScope	Trusted identities	Reviewed scikit-learn profiles

7.3 Runtime Verification and ML Software Quality Assurance

Design by Contract (Meyer, 1992) and runtime verification (Bauer et al., 2011) check whether observed program executions satisfy formal specifications. In Python, frameworks such as `DynaPyt` provide dynamic instrumentation hooks (Eghbali and Pradel, 2022), while dynamic taint analysis tracks origin propagation across program variables (Schwartz et al., 2010). A Python-level hook at a call boundary can see the arguments,

but the hook alone cannot establish whether a compiled Cython or C solver consumed an array during a zero-gradient pass. Reviewed native contracts establish that consumption and identify which internal fields the supported prediction path leaves unread. For training-source isolation, construction participation can occur without a numerical change in predictions, as in the zero-gradient case. This distinguishes the audit objective from classical information-flow security (Sabelfeld and Myers, 2003). Notebook execution can leave a recorded fit out of step with the model still in memory (Macke et al., 2021; Pimentel et al., 2019). Optimizer counters and random states matter because their effects differ by use: they guide continued training while the reviewed predictors read other fields (Bottou, 2010).

Within the broader landscape of ML testing and lifecycle quality assurance (Ashmore et al., 2022; Breck et al., 2017; Martínez-Fernández et al., 2022; Paleyes et al., 2023; Riccio et al., 2020; Serban et al., 2020; Siebert et al., 2022; Zhang et al., 2022), StateScope checks training-source isolation for a specific fitted model, complementing statistical cross-validation controls (Cawley and Talbot, 2010; Varma and Simon, 2006) and behavioral privacy audits (Shokri et al., 2017).

8 Conclusion

StateScope verifies the training sources of mutable fitted versions by observing actual state consumption, replacement, and the state required for the intended use. A validated dependency can establish exposure with incomplete evidence elsewhere. Isolation requires a sound, complete source bound. Erroneous-edge controls make the dependence on faithful operation contracts explicit.

The native executions establish where verified historical dependencies and configuration transitions can resolve different queries. The public workflows demonstrate the checks on selected single-process models and identify where worker or persistence boundaries leave model associations unresolved. Staged measurements locate the main resource costs, and registered-only discovery reduces cold startup on the tested paths. The resulting deployment scope is resource-budgeted offline audits of reviewed operations.

Statements and Declarations

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Competing Interests

The author declares no financial or non-financial competing interests.

Author Contributions

SIQI LI is the sole author and is responsible for the research and manuscript.

Data and Code Availability

StateScope v0.1.2 (LI, 2026) is available at https://github.com/FischotterDunklerort/statescope under the tag `StateScopev0.1.2` and is archived at https://doi.org/10.5281/zenodo.24416487. The software is licensed under Apache License 2.0. Supplementary Material S1 (Online Resource 1) supplies the replication package for review: per-query audit records, original timing observations, retained native-reference and public-workflow evidence, source-data manifests, and Python scripts for the reported summaries, figures, and tables. Its study-to-file index identifies the inputs and checks for each result and distinguishes recomputation of recorded observations from rerunning training workflows. Third-party datasets retain their original licenses and access conditions; their source locations are identified in the manuscript and package documentation.

Use of Generative Artificial Intelligence

ChatGPT 6.1 (OpenAI) assisted with translation, language polishing, revision of the manuscript structure, and summarizing reference literature. The author takes full responsibility for the accuracy of the manuscript, experimental results, analyses, figures, and scientific claims. Figures and tables were generated programmatically using Python from recorded data or the stated method definitions; no AI image-generation tool was used.

References

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software Engineering for Machine Learning: A Case Study. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 291–300. https://doi.org/10.1109/icse-seip.2019.00042

Ashmore, R., Calinescu, R., & Paterson, C. (2022). Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges. *ACM Computing Surveys*, 54(5), 1–39. https://doi.org/10.1145/3453444

Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4), Article 14. https://doi.org/10.1145/2000799.2000800

Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent. In Y. Lechevallier, & G. Saporta (Eds.), *Proceedings of COMPSTAT'2010* (pp. 177–186). Physica-Verlag HD. https://doi.org/10.1007/978-3-7908-2604-3_16

Bourtoule, L., Chandrasekaran, V., Choquette-Choo, C. A., Jia, H., Travers, A., Zhang, B., Lie, D., & Papernot, N. (2021). Machine Unlearning. *2021 IEEE Symposium on Security and Privacy*, 141–159. https://doi.org/10.1109/SP40001.2021.00019

Breck, E., Cai, S., Nielsen, E., Salib, M., & Sculley, D. (2017). The ML test score: A rubric for ML production readiness and technical debt reduction. *2017 IEEE*

International Conference on Big Data (Big Data), 1123–1132. https://doi.org/10.1109/bigdata.2017.8258038

Buneman, P., Khanna, S., & Tan, W. C. (2001). Why and Where: A Characterization of Data Provenance. In J. Van den Bussche, & V. Vianu (Eds.), *Database Theory — ICDT 2001* (Lecture Notes in Computer Science, vol. 1973, pp. 316–330). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-44503-x_20

Cawley, G. C., & Talbot, N. L. C. (2010). On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation. *Journal of Machine Learning Research*, 11, 2079–2107. https://jmlr.org/papers/v11/cawley10a.html

Chapman, A., Lauro, L., Missier, P., & Torlone, R. (2024). Supporting Better Insights of Data Science Pipelines with Fine-grained Provenance. *ACM Transactions on Database Systems*, 49(2), 1–42. https://doi.org/10.1145/3644385

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

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 238–252. https://doi.org/10.1145/512950.512973

Dask Developers. (n.d.). Text Vectorization Pipeline. *Dask Examples*. https://examples.dask.org/machine-learning/text-vectorization.html (Accessed 14 November 2026).

Dask-ML Developers. (n.d.). Incremental Learning. *Dask-ML documentation*. https://ml.dask.org/incremental.html (Accessed 14 November 2026).

Dwork, C. (2006). Differential Privacy. *33rd International Colloquium on Automata, Languages and Programming, part II*, 1–12. https://doi.org/10.1007/11787006_1

Eghbali, A., & Pradel, M. (2022). DynaPyt: A Dynamic Analysis Framework for Python. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 760–771. https://doi.org/10.1145/3540250.3549126

Grafberger, S., Groth, P., Stoyanovich, J., & Schelter, S. (2022). Data distribution debugging in machine learning pipelines. *The VLDB Journal*, 31(5), 1103–1126. https://doi.org/10.1007/s00778-021-00726-w

Green, T. J., Karvounarakis, G., & Tannen, V. (2007). Provenance semirings. *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 31–40. https://doi.org/10.1145/1265530.1265535

Herschel, M., Diestelkämper, R., & Ben Lahmar, H. (2017). A survey on provenance: What for? What form? What from?. *The VLDB Journal*, 26(6), 881–906. https://doi.org/10.1007/s00778-017-0486-1

Imieliński, T., & Lipski, W. (1984). Incomplete Information in Relational Databases. *Journal of the ACM*, 31(4), 761–791. https://doi.org/10.1145/1634.1886

Kapoor, S., & Narayanan, A. (2023). Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9), Article 100804. https://doi.org/10.1016/j.patter.2023.100804

Kaufman, S., Rosset, S., Perlich, C., & Stitelman, O. (2012). Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1–21. https://doi.org/10.1145/2382577.2382579

Koh, P. W., & Liang, P. (2017). Understanding Black-box Predictions via Influence Functions. *Proceedings of the 34th International Conference on Machine Learning, PMLR 70*, 1885–1894. https://proceedings.mlr.press/v70/koh17a.html

LI, S. (2026). *StateScope: Auditing Learned-State Provenance in scikit-learn Workflows* (Version 0.1.2) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.24416487

Macke, S., Gong, H., Lee, D. J. L., Head, A., Xin, D., & Parameswaran, A. (2021). Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment*, 14(6), 1093–1101. https://doi.org/10.14778/3447689.3447712

Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Vollmer, A. M., & Wagner, S. (2022). Software Engineering for AI-Based Systems: A Survey. *ACM Transactions on Software Engineering and Methodology*, 31(2), 1–59. https://doi.org/10.1145/3487043

Meyer, B. (1992). Applying Design by Contract. *Computer*, 25(10), 40–51. https://doi.org/10.1109/2.161279

Paleyes, A., Urma, R. G., & Lawrence, N. D. (2023). Challenges in Deploying Machine Learning: A Survey of Case Studies. *ACM Computing Surveys*, 55(6), 1–29. https://doi.org/10.1145/3533378

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2017). noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *Proceedings of the VLDB Endowment*, 10(12), 1841–1844. https://doi.org/10.14778/3137765.3137789

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019). A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 507–517. https://doi.org/10.1109/msr.2019.00077

Qadeib Alban, A., Belhajjame, K., & Grigori, D. (2026a). A W3C PROV–Aligned Metamodel for Tracing End-to-End Provenance in ML Pipelines. In T. Polacsek, M. Ruiz, J. Ralyté, & F. Ravat (Eds.), *Research Challenges in Information Science* (Lecture Notes in Business Information Processing, vol. 585, pp. 210–226). Springer, Cham. https://doi.org/10.1007/978-3-032-26836-5_13

Qadeib Alban, A., Belhajjame, K., & Grigori, D. (2026b). MLProvLens: Exploring End-to-End Provenance in ML Pipelines with a W3C PROV-Aligned Framework. In A. Mauri, L. Burgueño, & R. Tommasini (Eds.), *Web Engineering* (Lecture Notes in Computer Science, vol. 16625, pp. 333–337). Springer, Cham. https://doi.org/10.1007/978-3-032-29372-5_36

Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., & Tonella, P. (2020). Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25, 5193–5254. https://doi.org/10.1007/s10664-020-09881-0

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131–164. https://doi.org/10.1007/s10664-008-9102-8

Sabelfeld, A., & Myers, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Schelter, S., Grafberger, S., Guha, S., Karlaš, B., & Zhang, C. (2023). Proactively Screening Machine Learning Pipelines with ArgusEyes. *Companion of the 2023 International Conference on Management of Data*, 91–94. https://doi.org/10.1145/3555041.3589682

Schelter, S., Guha, S., & Grafberger, S. (2024). Automated Provenance-Based Screening of ML Data Preparation Pipelines. *Datenbank-Spektrum*, 24(3), 187–196. https://doi.org/10.1007/s13222-024-00483-4

Schlegel, M., & Sattler, K. U. (2025). Capturing end-to-end provenance for machine learning pipelines. *Information Systems*, 132, Article 102495. https://doi.org/10.1016/j.is.2024.102495

Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010). All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). *2010 IEEE Symposium on Security and Privacy*, 317–331. https://doi.org/10.1109/SP.2010.26

scikit-learn Developers. (n.d.). HashingVectorizer (version 1.8.0 documentation). Retrieved November 14, 2026, from https://scikit-learn.org/1.8/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., & Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems*, 28. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

Serban, A., van der Blom, K., Hoos, H., & Visser, J. (2020). Adoption and Effects of Software Engineering Best Practices in Machine Learning. *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–12. https://doi.org/10.1145/3382494.3410681

Shokri, R., Stronati, M., Song, C., & Shmatikov, V. (2017). Membership Inference Attacks Against Machine Learning Models. *2017 IEEE Symposium on Security and Privacy*, 3–18. https://doi.org/10.1109/SP.2017.41

Siebert, J., Joeckel, L., Heidrich, J., Trendowicz, A., Nakamichi, K., Ohashi, K., Namba, I., Yamamoto, R., & Aoyama, M. (2022). Construction of a quality model for machine learning systems. *Software Quality Journal*, 30, 307–335. https://doi.org/10.1007/s11219-021-09557-y

Urbanowicz, R. J., Zhang, R., Cui, Y., & Suri, P. (2023). STREAMLINE: A Simple, Transparent, End-To-End Automated Machine Learning Pipeline Facilitating Data Analysis and Algorithm Comparison. In L. Trujillo, S. M. Winkler, S. Silva, & W. Banzhaf (Eds.), *Genetic Programming Theory and Practice XIX* (pp. 201–231). Springer. https://doi.org/10.1007/978-981-19-8460-0_9

Varma, S., & Simon, R. (2006). Bias in error estimation when using cross-validation for model selection. *BMC Bioinformatics*, 7(1), Article 91. https://doi.org/10.1186/1471-2105-7-91

Vartak, M., Subramanyam, H., Lee, W. E., Viswanathan, S., Husnoo, S., Madden, S., & Zaharia, M. (2016). ModelDB: A system for machine learning model management. *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 1–3. https://doi.org/10.1145/2939502.2939516

Yang, C., Brower-Sinning, R. A., Lewis, G., & Kästner, C. (2022). Data Leakage in Notebooks: Static Detection and Better Processes. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–12. https://doi.org/10.1145/3551349.3556918

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Xie, F., & Zumar, C. (2018). Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41, 39–45. https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf

Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2022). Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering*, 48(1), 1–36. https://doi.org/10.1109/TSE.2019.2962027

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- ESM1.zip