

Fidelis: Exploiting Task Similarity for Low-overhead Distributed Workflow Audit and Replay[Research]

Abstract

Reproducing distributed scientific workflows across heterogeneous clusters requires capturing the full execution context, including file access patterns and dependency environments, without approximation or data reduction. However, current provenance tracking approaches suffer from steep system-call interception overheads and fragmented logging across workers lacking a shared file system. These steep overheads are primarily due to treating each distributed task as an independent entity. We observe that distributed computations exhibit significant structural and environmental similarity across tasks, offering an opportunity to mitigate auditing overheads. Exploiting this insight, we present Fidelis, a scalable audit and replay framework for large scale distributed workflows. Fidelis minimizes auditing overhead through two key methods: execution context reuse that eliminates cold start latency and minimized context-switching overhead that eliminates redundant interception. Fidelis employs stable hash identifiers and content-addressable provenance graphs to decouple replay from the original execution environment, enabling cross platform workflow reproduction without worker-specific assumptions. We evaluate Fidelis on five real-world scientific computing applications and demonstrate up to 93% reduction in auditing overhead over plain execution, with replay time reduced by up to 93%, all while preserving full execution fidelity across heterogeneous clusters.

ACM Reference Format:

. 2026. Fidelis: Exploiting Task Similarity for Low-overhead Distributed Workflow Audit and Replay[Research]. In . ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Scientific workflows are the backbone of modern computational research. Applications in genomics, climate modeling, particle physics, and machine learning routinely decompose into thousands of independent tasks that execute across heterogeneous clusters, each reading from and writing to shared or distributed storage. Reproducing such workflows *i.e.*, re-executing them in a different environment and obtaining the same results is a fundamental requirement of scientific practice. Reproduction demands that the complete execution context of every task be captured faithfully: the exact set of files read, the software dependencies loaded, the environment variables consulted. Without such a record, a workflow is not a scientific artifact but becomes an unrepeatable event.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Auditing is the process of capturing this execution context; replay is the process of reconstructing it in a new environment. Together, audit and replay form a closed loop of scientific accountability. Several systems have made progress on this loop. Application virtualization tools [2, 11] intercept system calls via `ptrace` to capture the dependencies and file accesses of a running process, producing a self-contained container that can be replayed elsewhere. These systems demonstrate that audit and replay can be made practical for individual, non-distributed workloads.

However, distributed scientific workflows are not individual workloads. A workflow executing across a cluster runs hundreds or thousands of tasks in parallel across many workers, none of which share a file system. If we were to audit each of these tasks, as it implies tracing each task, intercepting the task's system calls, copying accessed files into a per-task container, and logging provenance records to a local storage.

This incurs two prominent overheads: First, similar tasks are audited redundantly, especially environment dependencies. Thus tasks that run the same code and depend on the same libraries, are audited redundantly for the same software dependencies over and over (*see* Figure 1(left)). These identical dependencies are not only audited but are then also copied producing duplicate provenance records and duplicate copies of identical files across containers. Second, intercepting every system-call within a task requires the application to surrender control to an external tracer, context-switch to the kernel and back, and wait while the tracer copies accessed files; this happens at the entry and exit of every relevant system call for every task. In a workflow with thousands of tasks that share a common set of package imports, the overall cost of loading and auditing those packages is paid thousands of times.

Since distributed workflows exhibit pervasive structural similarity across tasks it is natural to use it to improve their execution. Indeed prior work [20] has exploited this repetition to speed up execution by distributing and retaining reusable function contexts across workers. Because the function is the same, its context *i.e.*, the libraries it imports, the models it loads is identical across all invocations. However, all current work requires users to manually specify and annotate which parts of the context are shared *i.e.*, the burden of identifying the common context falls entirely on the programmer. Consequently automated audit mechanisms like those in application virtualization tools cannot directly use the common context.

Our key insight is that the task signature, already computed for execution reuse, simultaneously defines audit equivalence classes, without any programmer annotation. This is true because tasks that share a signature load the same code and the same dependencies, and therefore access the same set of files will produce the same provenance record. In our evaluation, 60–70% of tasks in representative scientific workflows share equivalence classes, meaning their provenance record is known before they even execute.

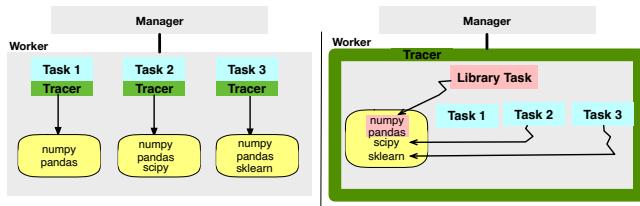


Figure 1: Distributed audit and replay necessitates per-task container creation (left). Fidelis (right) avoids it by creating audit equivalence classes termed ‘Library Tasks’, which different tasks use to amortize audit overhead. For replay fidelity, Task 2 and 3 are still audited in a light-weight manner to identify task-specific dependencies.

We present Fidelis, a scalable audit-and-replay middleware for distributed scientific workflows that treats audit equivalence classes as a first-class resource. In Fidelis, tasks that share a signature form a common ‘library task’, and tasks whose signature is a subset of an existing library task’s signature, share an execution context and therefore share a provenance record. Thus rather than auditing every such task independently, Fidelis audits once at a *library* task level producing a common dependency manifest that covers all ‘similar’ tasks dispatched to that worker. This eliminates the redundant per-task audit overhead. However, tasks may still dynamically load dependencies at runtime that were not captured by the signature. To ensure that no dependency is ever missed—and that replayed workflows do not fail—every task runs a light-weight interposition layer at the worker level that watches only for dynamic loads. In the common case, nothing new is observed and the overhead is negligible. If it does, per-task interposition, and not interception, handles the dynamic exception correctly, giving Fidelis both efficiency and completeness (see Figure 1 (right)).

Realizing Fidelis requires addressing three challenges. The first is an optimization challenge: context reuse must preserve both completeness and attribution. Sharing a single audited context across many tasks risks missing dependencies that a task loads dynamically beyond its static signature, and risks misattributing provenance—crediting a dependency to the library task rather than to the specific task that accessed it. Fidelis addresses this by combining library-task-level interposition, which audits the common context once, with per-task interposition that watches for dynamic loads and correctly attributes them to the task that triggered them.

The second is a performance challenge: capturing provenance must not block task execution. Rather than using `ptrace` to attach an external tracer, which requires a privilege-mode transition on every system call entry and exit, Fidelis interposes a shared library within the task’s own address space, keeping all provenance capture in user mode. An asynchronous materialization protocol further decouples container construction from task execution: accessed files are recorded to an in-memory buffer and flushed in batches, so the task is never blocked waiting for container writes.

Finally, the third is a correctness challenge: replay requires that provenance records be matchable to tasks across different runs and environments. Fidelis addresses this through stable task identifiers where in a normalization procedure strips nondeterministic

components from the task payload, such as temporary file names, worker-specific paths, and framework-generated identifiers, and hashes the result to produce a fingerprint that is stable across workflow runs. Combined with content-addressable provenance graphs, this design ensures that replay is decoupled from the original execution environment and remains portable across clusters that share no file system, or software installation.

Fidelis is scoped to the audit and replay problem: its goal is to capture a faithful, low-overhead provenance record of what each task accessed, and to replay that task in a new environment using the captured context. A related but independent question is whether the per-task and per-library-task containers produced by Fidelis can be merged into a single globally consistent container representing the full workflow dependency closure. Fidelis does not solve this problem, as it involves dependency constraint resolution across distributed, incomplete views—a research challenge in its own right. We note, however, that Fidelis’s content-addressable provenance graph are a necessary precondition for any such merge system: faithful recording must precede resolution.

Our contributions are as follows:

- We identify three properties: completeness, fidelity, and portability, that a correct and efficient audit-and-replay system for distributed workflows must satisfy, and show that naive per-task auditing satisfies none of them efficiently at scale.
- We introduce the key insight that the task signature, already computed for execution reuse, simultaneously defines audit equivalence classes. We exploit this coupling to audit once at the library-task level, eliminating redundant per-task audit overhead while preserving completeness and attribution.
- We eliminate context-switching overhead during audit by replacing `ptrace`-based interception with in-process system-call interposition, keeping all provenance capture in user mode. An asynchronous materialization protocol further decouples container construction from task execution.
- We introduce stable task identifiers, hash-based fingerprints produced by normalizing nondeterministic components of the task payload that remain invariant across workflow runs and execution environments, enabling portable replay on clusters.
- We implement these contributions in Fidelis, a functioning end-to-end prototype for distributed workflow audit and replay. We evaluate Fidelis on five real-world distributed scientific workflows and show that Fidelis reduces workflow audit time by 84–93% over plain execution and 90–97% over base audit for homogeneous workflows, with stable and consistent performance under compute-intensive workloads at scale.

The rest of the paper is organized as follows. Section 2 gives background on audit and replay in a distributed manager-worker environment; Section 3 outlines properties and goals with Fidelis architecture, Section 4 describes the three essential Fidelis primitives; Section 5 shares experiments and their detailed analysis, Section 6 discusses these results, Section 7 gives relevant literature review, and Section 8 concludes with a summary of our contributions.

2 Background

Distributed Workflow and Manager-Worker Model. A distributed scientific workflow is organized as a directed acyclic graph (DAG) of *tasks*. Each task is a self-contained unit of computation: it consumes a set of input files, executes a function or program against them, and produces a set of output files. Tasks are coordinated by a *manager-worker* system. A central manager typically maintains the task graph and dispatches ready tasks, those whose inputs are available, to a pool of workers. Workers execute tasks independently, without sharing memory or a common file system, and return results to the manager upon completion. This architecture supports massive parallelism: hundreds or thousands of tasks may execute concurrently across heterogeneous clusters spanning many physical machines. The manager treats each task as an opaque unit of work, scheduling it to whichever worker is available without knowledge of the task's internal structure.

Execution Contexts. The *execution context* of a task comprises all software state that must be initialized before application code runs: the language runtime, dynamically linked shared libraries, and loaded packages. In the Python setting this initialization proceeds through five distinct stages for every task: shell interpretation and forking, dynamic linking, Python runtime initialization, code compilation, and package loading. In a distributed workflow, the user provides dependency specifications to the manager, which stages the required files and packages to the worker prior to execution. The worker then instantiates the execution context independently, traversing these staged dependencies to initialize the runtime environment before the task runs.

Lineage Auditing and Replay. *Auditing* is the process of capturing, during task execution, a complete record and copy of every file accessed—data or library loaded by this task. This record also termed the task's *lineage* is sufficient to reconstruct the execution context on a different system, enabling *replay*: re-executing the task in that new environment and producing the same result.

Both audit and replay is implemented through system-call interception [11]. Every interaction between a running process and the operating system: opening a file, reading its contents, forking a child process, executing a binary, loading a shared library, is mediated by a system call. A ptrace-based audit framework attaches to the target process as an external tracer and intercepts these calls, recording them as a sequential structured provenance log. The lineage captured during a single task's execution is expressed as λ , defined recursively over the sequence of system calls issued by that task:

$$\lambda_i = (\lambda_{i-1}, h_i, \delta_i) \quad (1)$$

where i indexes system call steps within the task's execution, $\lambda_0 = \emptyset$ is the empty record at task start, h_i is a content hash of the task code executing at step i , and δ_i is the system event issued at step i together with the content hash of any external file accessed. The lineage λ_n after n system calls thus forms a cumulative chain that uniquely characterizes the complete execution history of the task i.e., both the code that ran and the data it consumed.

In addition to logging metadata, the audit framework copies the content of every accessed file into an isolated *container*—a self-contained package bundling the task's dependencies, data inputs,

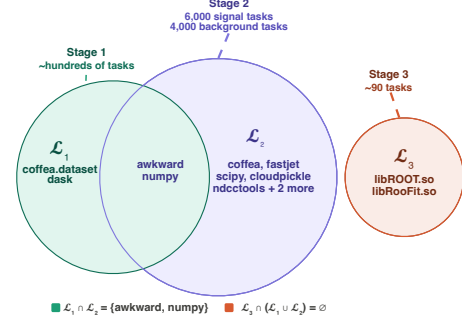


Figure 2: CMS Jet Shared Execution Contexts in different stages

and provenance log. This container is the unit of replay: given a container, a replay framework redirects all file accesses to the containerized copies, re-executing the task in a faithfully reproduced environment.

Establishing that two executions produced equivalent provenance requires resolving three subtleties. First, process identifiers are non-deterministic: the same logical process receives a different PID on each run, so lineage comparison must abstract PIDs to their logical roles. Second, concurrent processes issue system calls in an interleaved total order that may differ across runs even when the underlying partial order of events is identical; lineage equality must therefore be established over partial orders, not total orders. Third, external file content must be compared by content hash, not by path, since the same path may refer to different content across environments.

2.1 Motivating Example: A CMS Jet Substructure Analysis

We consider a real distributed scientific workflow from experimental high-energy physics. The workflow implements a jet substructure analysis on proton-proton collision data recorded by the CMS detector at CERN [6]. The analysis is orchestrated by a TaskVine manager using a Dask front-end, dispatching tasks across a heterogeneous cluster. The workflow comprises of three stages; the first two are drawn directly from the available open-source code, and the third is a representative completion that a full CMS analysis would typically include [1].

Stage 1: Preprocessing. The manager dispatches one preprocessing task per sample signal category to scan the input ROOT files and partition them into 50,000-event chunks, recording chunk boundaries in a metadata file. Every preprocessing task is structurally identical: each initializes the same Python runtime and loads the library set:

$$\mathcal{L}_1 = \{\text{coffea.dataset_tools}, \text{dask}, \text{awkward}, \text{numpy}\},$$

Each task scans ROOT file metadata via sequential POSIX read system calls, and writes chunk boundary records to a shared `samples_ready.json`. All tasks in this stage exhibit identical environmental requirements and form a single homogeneous execution context class.

Stage 2: Iterative Analysis. Once preprocessing completes, the manager dispatches thousands of parallel analysis tasks, one per 50,000-event chunk per across multiple signal and background datasets. Every analysis task loads the richer library set:

$$\mathcal{L}_2 = \{\text{coffea, dask_awkward, awkward, numpy, fastjet, scipy, cloudpickle, ndcctools.taskvine, variable_functions}\}.$$

Bootstrapping $\mathcal{L}_2$ requires thousands of independent `openat`, `stat`, and `mmap` system calls per task instance as the Python interpreter traverses site-packages on a shared network file system. While all tasks share the baseline $\mathcal{L}_2$ context, the workflow branches into divergent execution profiles based on the underlying physics data being processed. Tasks processing *signal* samples dynamically import truth-matching modules to isolate simulated Higgs boson decay products (`pdgId` = 25). Conversely, tasks processing *electroweak background* samples bypass these modules, executing an entirely different code path that evaluates background vector bosons (`|pdgId|` $\in$ {23, 24}).

Stage 3: Statistical Combination and Fitting. The final stage aggregates the columnar Parquet files generated in Stage 2 to extract the physical signal significance. This stage utilizes the standard CMS combine tool—a high-performance, compiled C++ binary that dynamically links against massive native HEP libraries (`libROOT.so` and `libRootFit.so`).

$$\mathcal{L}_3 = \{\text{libROOT.so, libRootFit.so}\}$$

The execution context of Stage 3 does not share runtime components, interpreters, or libraries with $\mathcal{L}_1$ or $\mathcal{L}_2$. Its system-call footprint is dominated by heavy memory-mapped I/O (`mmap`) and multi-threaded synchronization primitives (`futex`). Furthermore, Stage 3 tasks produces worker-specific paths and framework-generated identifiers in their provenance records.

2.2 Strawman Solution and its Limitations

A natural approach to distributed workflow auditing is to extend existing single-node audit-and-replay tools directly to the manager-worker setting. In this strawman architecture, the manager dispatches each task to a worker as usual. Each worker attaches a `ptrace`-based tracer to its local task, intercepts every system call at entry and exit, writes out a container containing all intercepted files along with a provenance log, and returns this bundle to the manager upon completion. Applying this strawman auditing to the CMS workflow reveals three inefficiencies:

Redundant auditing. In Stage 1, every preprocessing task loads the identical library set $\mathcal{L}_1$, yet the strawman independently initializes the runtime and intercepts thousands of `openat`, `stat`, and `mmap` calls for each task. The same failure occurs at far greater scale in Stage 2: with 10,000 tasks sharing $\mathcal{L}_2$, the full bootstrapping cost is paid 10,000 times and 10,000 duplicate copies of identical files are written into separate containers.

Interception overhead. `ptrace` incurs two privilege-mode transitions per system call, one at entry and one at exit. Bootstrapping $\mathcal{L}_2$ alone triggers thousands of such transitions per task; across 10,000 tasks this overhead accumulates to tens of millions of transitions.

Stage 3 compounds this further: the combine tool’s `mmap` and `futex`-heavy footprint makes `ptrace`-based interception prohibitive on an already compute-intensive stage.

Unstable task identity. In Stage 2, the 6,000 signal and 4,000 background tasks share the same static signature $\mathcal{L}_2$ but diverge at runtime through data-driven dynamic imports. The strawman inherits non-deterministic scheduler identifiers, worker-specific paths, and framework-generated process identifiers without normalization, making it impossible to match a task submission to its previously captured container across runs. Stage 3 compounds this: the combine binary embeds worker-specific paths in its provenance records that, without normalization, cannot be matched to their environments during replay on a different cluster.

These three inefficiencies share a common root cause: the strawman treats each task as a fully independent entity, ignoring the structural similarity that distributed workflows exhibit across tasks. A correct and efficient audit-and-replay system must exploit this structure systematically, identifying equivalent tasks automatically, sharing their audited contexts, and producing provenance records that remain stable and matchable across runs and environments. We formalize these requirements in the following section and describe Fidelis architecture.

3 Fidelis Architecture

Before describing Fidelis’s design, we state precisely what it must achieve. The correctness properties define what a faithful audit-and-replay system must guarantee; the efficiency goals define the overhead constraints under which it must operate.

3.1 Properties and Design Goals

Properties. We define a correct distributed audit and replay as one that satisfies the following properties:

- **P1: Completeness.** The audit of a task must capture every file access, dynamic library load, environment variable read, and side effect that participates in task execution. No dependency or accessed artifact may be approximated or omitted. A replay missing any element of the original execution context may produce incorrect or irreproducible results.
- **P2: Fidelity.** Replaying a task’s captured container must produce the same provenance record as the original execution.¹
- **P3: Portability.** A task’s container must be replayable on any worker that can execute it, without requiring any element of the original execution environment to be present. The container must not embed worker-specific file system paths, scheduler-generated identifiers, or assumptions about pre-installed software. Portability makes no claim about hardware compatibility; it requires only that the captured environment be self-contained and free of implicit dependencies on the original worker.

Design Goals. Satisfying P1–P3 is necessary but not sufficient for a practical middleware system. A correct audit that treats every task as an independent audit target imposes overhead that grows linearly with task count. Correctness and efficiency must therefore be achieved simultaneously. We state three efficiency goals:

¹If the underlying computation is non-deterministic, such as floating-point variance in deep learning libraries, the provenance record must remain semantically equivalent, if not bit-wise identical.

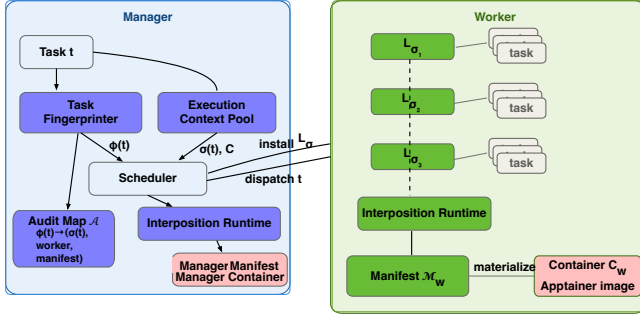


Figure 3: Fidelis Architecture

- **G1: Sub-linear overhead scaling.** The total auditing overhead must grow sub-linearly with the number of tasks in the same equivalence class. Costs attributable to the shared execution context must be paid once per equivalence class, not once per task.
- **G2: Low per-task interception cost.** Per-task overhead of intercepting and recording system calls must be minimized. In particular, privilege mode transitions induced by external tracing mechanisms and blocking of task executions for synchronous container I/O must be decoupled and eliminated from the task's critical path.
- **G3: Stable task identity.** Every logical task must receive a stable, portable *fingerprint* that remains invariant across different workflow runs and across execution environments. This fingerprint must be computable without relying on scheduler-specific identifiers, worker-specific paths, or process identifiers all of which are non-deterministic. The fingerprint must serve as the key that maps a task submission to the container in which it was previously executed, enabling correct retrieval.

3.2 Fidelis Architecture

Figure 3 shows Fidelis's three architectural components: (i) the Execution Context Pool, (ii) the Task Fingerprinter and Audit Map, and (iii) the In-process Interposition Runtime.

Task Fingerprinter and Audit Map (Manager). The Task Fingerprinter is co-located with the workflow manager and processes each task at submission time, before it is dispatched to a worker. It normalizes the task payload by replacing scheduler-generated keys, framework-assigned file handles, and worker-specific paths with stable canonical placeholders, and by substituting upstream task key references with the stable fingerprints of those upstream tasks. The normalized payload is hashed with SHA-256 to produce a stable fingerprint $\phi(t)$ that is identical across runs for the same logical computation. The fingerprinting component is responsible for the routing decision only; it determines which worker environment t requires and dispatches t accordingly.

The Audit Map $\mathcal{A}$ is maintained by the fingerprinter and updated incrementally as tasks complete. For each completed task, the fingerprinter records:

$$\mathcal{A}[\phi(t)] = (\sigma(t), ; \text{worker}, ; \text{manifest_path})$$

where $\sigma(t)$ is the module signature of task t , *worker* is the hostname of the worker that executed t , and *manifest_path* is the location of the dependency manifest $\mathcal{M}_w$ captured on that worker. $\mathcal{A}$ is flushed to the manager as a JSON file when the workflow finishes. During replay, the manager reads $\mathcal{A}$ and routes each incoming task to the worker whose captured environment matches its fingerprint. The details of fingerprint computation are described in Section 4.2.

Execution Context Pool (Worker). The Execution Context Pool maintains a cache C of persistent *library tasks* on each worker. A library task L_σ is a long-running process keyed by module signature σ , a frozenset of package names describing a set of software dependencies that has pre-loaded all modules in σ into its address space and persists over the course of the workflow. The set of all tasks dispatched to L_σ forms an *audit equivalence class* $\mathcal{E}\sigma$: tasks in $\mathcal{E}\sigma$ share an execution context and share a provenance record. The three concepts are thus distinct: $\sigma(t)$ is a property of the task describing its dependencies; L_σ is the persistent process that pre-loads those dependencies; and $\mathcal{E}\sigma$ is the set of tasks that reuse L_σ .

When the manager receives a task t , it extracts $\sigma(t)$ and consults C . If a library task L_σ exists in C with $\sigma(t) = \sigma$ or $\sigma(t) \subseteq \sigma$, the task is dispatched to that worker. Otherwise a new library task $L_{\sigma(t)}$ is installed on an available worker and registered in C . Each task executes as a child process forked from $L_{\sigma(t)}$, inheriting the pre-loaded address space without repeating initialization. During replay, workers are launched inside C_w , which already contains the pre-loaded library tasks. Tasks reuse the captured execution context without any reinstallation. The details of module signature extraction and library task management are described in Section 4.1.

In-process Interposition Runtime (Manager and Worker). The In-process Interposition Runtime is a shared library loaded into the manager and each worker process via LD_PRELOAD. Rather than attaching an external ptrace-based tracer, the library shadows standard glibc file access functions within each process's own virtual address space, recording accessed paths to a dependency manifest without copying any files. On the manager, the interposition runtime captures manager-side dependencies only. These are recorded to the manager manifest $\mathcal{M}_m$ and materialized into a manager container C_m . On each worker, the interposition runtime captures all file accesses by the library task L_σ and its child tasks, recording them to the worker manifest $\mathcal{M}_w$. Each manifest entry records the type of the accessed artifact, its absolute path, and the library task L_σ currently active on that worker, the library task to which the dependency is attributed. This attribution ensures that during replay, each task executes in an environment containing exactly the dependencies its equivalence class accessed during audit, and no others.

Container materialization is deferred entirely off the task's critical path: during execution, the interposition runtime records dependencies to the manifest only. After the worker completes its workload, $\mathcal{M}_w$ is used to build an Apptainer container image C_w containing all dependencies captured during audit. Similarly, $\mathcal{M}_m$ is materialized into C_m . During replay, the manager is launched inside C_m and each worker is launched inside C_w . Each task executes in a filtered view $C_w[\mathcal{E}\sigma]$ of the worker container, containing only the dependencies attributed to its equivalence class, constructed from the attribution entries in $\mathcal{M}_w$. No system call interposition

Algorithm 1 Fidelis Audit Workflow

Require: Task t , audit map $\mathcal{A}$, context pool C , manifests $\mathcal{M}_w, \mathcal{M}_m$

Ensure: $\mathcal{A}[\phi(t)]$ recorded; C_w, C_m materialized post-workflow

- 1: ▷ Manager, at task submission
- 2: $\phi(t) \leftarrow \text{FINGERPRINT}(t)$
- 3: $\sigma(t) \leftarrow \text{EXTRACTSIGNATURE}(t)$
- 4: **if** $L_{\sigma(t)} \notin C$ **then**
- 5: $\text{INSTALLLIBRARYTASK}(\sigma(t)); C \leftarrow C \cup L_{\sigma(t)}$
- 6: **end if**
- 7: **dispatch** t to worker hosting $L_{\sigma(t)}$
- 8: ▷ Worker, during task execution
- 9: $\tau \leftarrow \text{FORK}(L_{\sigma(t)})$
- 10: $\text{EXECUTE}(\tau, t)$ ▷ interposition records to $\mathcal{M}_w$, attributed to $L_{\sigma(t)}$
- 11: ▷ Manager, at task completion
- 12: $\mathcal{A}[\phi(t)] \leftarrow (\sigma(t), \text{worker}, \text{manifest_path})$
- 13: ▷ Post-workflow
- 14: $C_w \leftarrow \text{MATERIALIZE}(\mathcal{M}_w); C_m \leftarrow \text{MATERIALIZE}(\mathcal{M}_m)$

is required at replay time: the containers provide complete filesystem isolation natively. The details of the interposition mechanism, deduplication scheme, and materialization protocol are described in Section 4.3.

3.3 Distributed Audit and Replay

We explicitly describe the the temporal ordering and handoffs between manager and worker in audit and repeat workflows in a distributed environment below.

3.3.1 Audit Workflow. When the workflow is launched, the interposition library is loaded into the manager and each worker process via LD_PRELOAD, activating file access recording for the entire execution. When the manager receives a task t , the fingerprinter computes its stable fingerprint $\phi(t)$ at submission time and extracts its module signature $\sigma(t)$. If a library task $L_{\sigma(t)}$ already exists in C , the task is dispatched to that worker as a function call; otherwise a new library task is installed first. The task executes as a child of $L_{\sigma(t)}$, reusing its pre-loaded execution context. The interposition library on the worker records all file accesses from the library task and its child tasks to the shared manifest $\mathcal{M}_w$, with each entry attributed to the active library task $L_{\sigma(t)}$. Upon task completion, the manager records $\mathcal{A}[\phi(t)] = (\sigma(t), \text{worker}, \text{manifest_path})$ in the audit map. When the workflow finishes, $\mathcal{A}$ is flushed to disk. Concurrently, the manager's interposition runtime records manager-side dependencies to $\mathcal{M}_m$. After the workflow completes, each worker materializes $\mathcal{M}_w$ into an Apptainer container image C_w , and the manager materializes $\mathcal{M}_m$ into C_m , producing self-contained replay environments for both.

3.3.2 Replay Workflow. During replay, the manager is launched inside C_m and each worker is launched inside C_w , the Apptainer containers produced during the original audit. Workers advertise a feature label identifying their captured environment. The manager reads $\mathcal{A}$ from disk. For each incoming task t' , the fingerprinter

Algorithm 2 Fidelis Replay Workflow

Require: Task t' , audit map $\mathcal{A}$, manifest $\mathcal{M}_w$

Ensure: t' executes inside $C_w[\mathcal{E}\sigma(t')]$

- 1: ▷ Worker startup, once per equivalence class
- 2: **for all** $L_\sigma \in C$ **do**
- 3: $C_w[\mathcal{E}\sigma] \leftarrow \text{FILTEREDVIEW}(\mathcal{M}_w, L_\sigma)$
- 4: **end for**
- 5: ▷ Manager, running inside C_m
- 6: $\phi(t') \leftarrow \text{FINGERPRINT}(t')$
- 7: $(\sigma(t'), \text{worker}, \text{manifest_path}) \leftarrow \mathcal{A}[\phi(t')]$
- 8: **dispatch** t' to worker advertising $\mathcal{E}\sigma(t')$
- 9: ▷ Worker, running inside $C_w[\mathcal{E}\sigma(t')]$
- 10: $\text{EXECUTE}(t')$ ▷ no interposition; container provides isolation

recomputes $\phi(t')$ using the same normalization procedure. Because $\phi(t')$ is stable across runs, the audit map lookup returns $(\sigma(t'), \text{worker}, \text{manifest_path})$. The manager pins t' to the worker advertising the matching feature label and dispatches it. On the worker, the filtered view $C_w[\mathcal{E}\sigma(t')]$ is constructed from $\mathcal{M}_w$ by selecting all manifest entries attributed to $L_{\sigma(t')}$. The task executes inside $C_w[\mathcal{E}\sigma(t')]$, which provides complete filesystem isolation using the dependencies captured during audit. No system call interposition is required at replay time; the containers redirect all file accesses to the captured environment transparently.

4 Fidelis Primitives

4.1 Execution Context Reuse

As established in Section 2, initializing a Python execution context is expensive and the five initialization stages are repeated identically for every task. In a distributed workflow where hundreds or thousands of tasks share the same software dependencies, this cost dominates. Fidelis eliminates this redundancy by identifying tasks that share a common execution context and reusing that context across all such tasks.

Fidelis automatically determines task similarity without programmer annotation through static analysis of the task's function object. When the manager receives a task, it scans the task's bytecode using Python's `dis` module, identifying `LOAD_GLOBAL` opcodes whose names resolve to imported modules in the function's global namespace and `IMPORT_NAME` opcodes that correspond to explicit in-body imports. The union of these forms the task's *module signature*, a frozenset of package names $\sigma(t)$ that fully describes the static software dependency set of task t . Static analysis is bounded in callable traversal depth: modules imported inside deeply nested callables may not appear in $\sigma(t)$. Such modules are loaded by the child task on demand at runtime and are captured by the per-task interposition layer described in Section 4.3. In Fidelis, no dependency is missed.

- (1) **Exact match:** If $\exists L_\sigma \in C$ such that $\sigma = \sigma(t)$, the task is dispatched to L_σ .
- (2) **Subset match:** If $\exists L_\sigma \in C$ such that $\sigma(t) \subseteq \sigma$, the task is dispatched to L_σ . The extra pre-loaded modules in $\sigma \setminus \sigma(t)$ are inert with respect to t : they cannot affect t 's execution since t does not reference them.

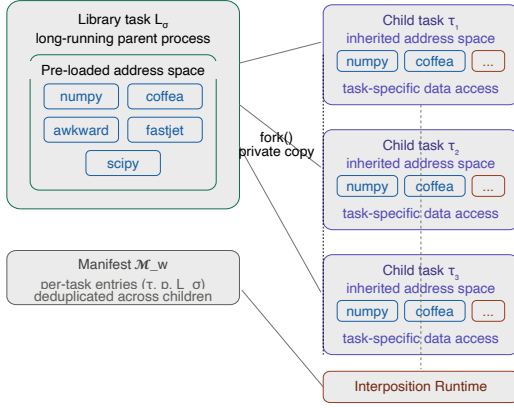


Figure 4: Library task as a parent process with forked children, the shared address space, and per-child independent execution.

- (3) **No match:** If neither condition is satisfied, the manager creates a new library task $L_{\sigma(t)}$, installs it on an available worker, and registers it in C .

The set of all tasks dispatched to a given library task L_{σ} forms an *audit equivalence class* $\mathcal{E}_{\sigma}$ — tasks in $\mathcal{E}_{\sigma}$ share an execution context and, as established in Section 2, share a provenance record.

Child Task Execution and Reproducibility Guarantee. Figure 4 illustrates the execution model. Each task dispatched to L_{σ} is forked as a child process that inherits the parent’s pre-loaded address space as a private copy. The pre-loaded modules constitute read-only shared state i.e., no child can observe the mutable state of another, and state mutations by one child are not visible to any sibling or to the parent. Two properties together ensure that context reuse does not compromise reproducibility. First, child tasks are independent by construction: fork semantics guarantee each child receives its own address space copy, preventing cross-task state contamination. Second, the interposition layer tracks lineage per child task independently: every file access and library load issued by a child, whether pre-loaded by the parent or loaded on demand, is recorded as a separate entry (τ, p, L_{σ}) in M_w , attributed to the active library task and deduplicated across concurrent children. Together these properties ensure that the provenance record of each task in $\mathcal{E}_{\sigma}$ faithfully reflects that task’s individual execution, satisfying P1 and P2.

Reusing an execution context across tasks in $\mathcal{E}_{\sigma}$ eliminates cold-start overhead, but introduces a new requirement: each task must be assigned a stable identity invariant across workflow runs, so that during replay it can be correctly matched to the container capturing its execution environment. We describe how Fidelis constructs this identity in Section 4.2.

4.2 Stable Task Fingerprinting

Serialized task payloads contain four categories of non-deterministic content that cause logically identical computations to produce different fingerprints across runs:

- **Scheduler-generated task keys:** Run-varying UUIDs embedded in task identifiers, such as Dask optimizer keys (*subgraph_callable-62c7ee78*) or zip-task keys (*zip-51bff51e*).
- **Framework-assigned file handles:** Temporary identifiers assigned by a manager to stage task inputs and outputs, such as *w_6929c2a4* and *f_3e2a1b4c*.
- **Worker-specific paths:** Absolute file paths that embed worker hostnames, mount points, or scratch directory locations that differ across workers and runs.
- **Upstream task references:** Payload fields that reference the run-varying key of an upstream task as part of the task’s input specification.

Figure 5 illustrates categories 1–3: Tasks A and B perform identical computations but their raw payloads differ in scheduler keys, file handles, and paths. Without normalization, hashing $\rho(t)$ directly violates G3. Fidelis computes a normalized payload $\hat{\rho}(t)$ by applying two transformations to task payload $\rho(t)$:

- (1) **Token substitution:** Scheduler-generated keys, framework-assigned file handles, and worker-specific paths are replaced with stable canonical placeholders, applied to function definitions, input arguments, and framework-specific metadata fields.
- (2) **Upstream reference resolution:** Each upstream task key reference in task payload is replaced with $\phi(t')$, the already-computed stable fingerprint of the upstream task t' , anchoring it to the semantic content of the full dependency chain.

Upstream reference resolution requires that $\phi(t')$ is available before $\hat{\rho}(t)$ is computed. Fidelis enforces this by reconstructing the workflow DAG from task payloads as tasks arrive and buffering each task until the fingerprints of all its upstream dependencies have been computed. Fingerprints are thus computed in topological order, independently of manager’s dispatch order. The normalized payload is serialized and hashed with SHA-256:

$$\phi(t) = \text{SHA-256}(\hat{\rho}(t)) \quad (2)$$

As Figure 5 shows, after normalization Tasks A and B produce identical representations and thus the same fingerprint.

For any two executions of the same logical workflow, task t receives fingerprint $\phi(t)$ if and only if it performs the same computation with the same upstream dependency chain. This invariant holds because: (i) token substitution removes all run-varying identifiers from the local payload; and (ii) upstream reference resolution anchors each fingerprint recursively to the fingerprints of its upstream tasks, which by induction satisfy the same invariant. The base case is tasks with no upstream dependencies, whose normalized payloads are fully determined by their function definitions and input arguments.

The fingerprint $\phi(t)$ and the terminal lineage $\lambda_n(t)$ are complementary artifacts. $\phi(t)$ is computed before execution from the normalized task payload and identifies the logical computation. $\lambda_n(t)$ is accumulated during execution and records the complete system call history of the task. Together they form the complete audit record: $\phi(t)$ is the key under which $\lambda_n(t)$ is stored and retrieved during replay.

Fidelis maintains a persistent audit map $\mathcal{A}$, updated at task completion:

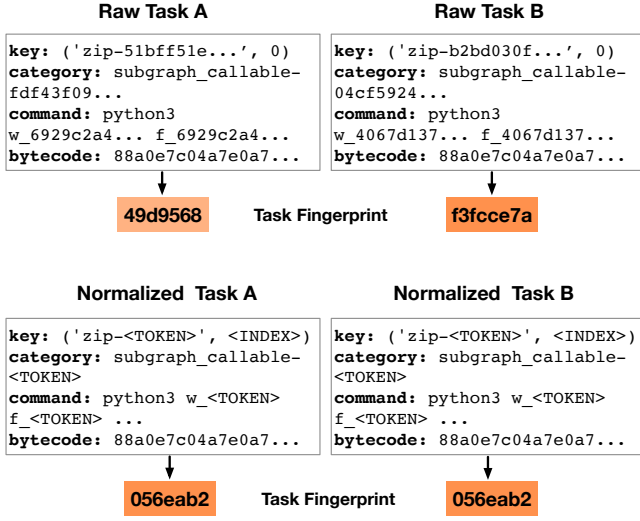


Figure 5: Stable fingerprints of a task computed through SHA-256. Tasks A and B in their raw form contain non-determinism. After normalization, they produce identical task fingerprints which are stable across executions.

$$\mathcal{A}[\phi(t)] = (\sigma(t), ; \text{worker}, ; \text{manifest_path}) \quad (3)$$

where $\sigma(t)$ identifies the equivalence class $\mathcal{E}_{\sigma(t)}$, *worker* is the hostname of the worker that executed t , and *manifest_path* is the filesystem path of $\mathcal{M}_w$ on that worker. $\mathcal{A}$ is flushed to disk as a JSON file when the workflow finishes.

During replay, Fidelis recomputes $\phi(t)$ for each incoming task using the same normalization procedure and queries $\mathcal{A}$. The manager retrieves $\sigma(t)$ and routes t to a worker advertising the feature label of $\mathcal{E}_{\sigma(t)}$. The fingerprinting component handles the routing decision only — construction of the filtered view $C_w[\mathcal{E}_{\sigma}]$ in which t executes is handled by the materialization component. This decouples replay routing entirely from scheduler state, worker-specific paths, and process identifiers, satisfying G3 and P3.

4.3 System Call Interposition and Asynchronous Materialization

Fidelis replaces ptrace-based interception with in-process system call interposition, eliminating privilege-mode transitions from the task’s critical path. A shared library loaded via LD_PRELOAD shadows standard glibc file access functions within the process’s own virtual address space. When the application calls a standard library function, the dynamic linker routes the call to the interposition library first, which records the necessary metadata and passes control to the real function. All interception occurs through ordinary function pointer jumps inside the process. The library intercepts four categories of calls, file accesses, filesystem metadata, shared library loads, and directory navigation, together capturing every dependency a task touches during execution.

LD_PRELOAD interposition can only observe accesses that flow through standard library routing. Three gaps exist and are addressed

explicitly. Libraries loaded before the interposition library’s constructor runs are captured by walking the dynamic linker’s in-memory link map at startup. Files opened by the dynamic linker during early process startup, such as locale archives, timezone files, the linker’s own cache, bypass standard routing and are recorded by a companion launcher binary that explicitly captures these known paths before launching the target process. Finally, Fidelis interposes both standard and versioned variants of each relevant function (e.g., `__xstat` and `stat`), ensuring completeness across kernel versions and glibc builds. Together these mitigations satisfy P1 despite the inherent boundary of user-space interposition.

Fidelis maintains a single manifest $\mathcal{M}_w$ per worker, shared across all tasks. Each entry is a triple $e = (\tau, p, L_{\sigma})$ where $\tau \in F, D, L$ is the entry type (regular file, directory, or symbolic link), p is the absolute path, and L_{σ} is the library task currently active on the worker, the library task to which the dependency is attributed. Symbolic link entries additionally record the link target. $\mathcal{M}_w$ is a set: a path is recorded only once, eliminating redundant logging of dependencies shared across library tasks or loaded dynamically by child tasks. The attribution field L_{σ} satisfies P2: it records which equivalence class first introduced each dependency, enabling per-task provenance records to reflect only the dependencies attributable to that class.

To prevent redundant writes across concurrent processes, Fidelis uses a three-level deduplication scheme. A thread-local last-seen cache handles repeated accesses within a thread with no locking cost. A per-process in-memory hash table covers paths seen earlier in the same process. A cross-process shared memory table is shared among the library task L_{σ} and all its concurrent child tasks: a path present at this level is never written to $\mathcal{M}_w$, ensuring each dependency is recorded exactly once without file-level locks on the manifest. Before writing any entry, the interposition library checks all three levels in order.

During execution, the interposition library records paths to $\mathcal{M}_w$ only and no file copying occurs on the task’s critical path. Materialization is deferred to after the worker completes, or performed incrementally during idle periods. It proceeds in three passes: symbolic links first (shallowest-first, to ensure link targets exist before links are reconstructed), then directories, then regular files. An additional pass captures shared library version symlinks invisible to interposition hooks. Apptainer then builds the materialized directory into C_w . Since $\mathcal{M}_w$ is continuously persisted to disk, it remains available to reconstruct C_w if the worker crashes mid-execution, satisfying P2.

At worker startup during replay, Fidelis constructs one filtered view per equivalence class:

$$\mathcal{M}_w[\mathcal{E}_{\sigma}] = (\tau, p, L_{\sigma'}) \in \mathcal{M}_w \mid L_{\sigma'} = L_{\sigma}$$

and materializes it as $C_w[\mathcal{E}_{\sigma}]$ via symlinks or bind mounts from C_w . Each task executes inside its class’s filtered view, seeing exactly the dependencies its equivalence class accessed during audit and no others. Since $C_w[\mathcal{E}_{\sigma}]$ is a lightweight projection of C_w , no additional file copying is required at replay time, satisfying P2 and P3.

Table 1: Characteristics and description of the five benchmark workflows.

Workflow	Max Files	Max Tasks	Conda Size	Container Size	Description
MapReduce	16,384	24,577	2.2 GB	80 MB	Counts word frequencies by executing Map tasks over input files, combining intermediate outputs via Combine tasks, and finalizing results with a Reduce task.
CTrend	3,000	3,509	1.67 GB	181 MB	Ingests raw temperature data, applies data cleaning tasks, computes anomalies, and aggregates trends through statistical analysis.
DConv	15	15,360	1.50 GB	145 MB	Segments a large input image into uniform spatial tiles, applies a convolution kernel to each tile independently, and stitches results into a single output image.
DV5	60	64	1.56 GB	1.8 GB	Parses raw particle collision data from CMS detector experiments, filters background noise, generates summary statistics, and combines intermediate results into a final particle collision distribution.
RAG	5,000	5,000	733.7 MB	96 MB	Processes text documents into chunks, builds a BM25 keyword-matching index, and routes incoming queries through the index to retrieve context blocks for LLM prompt completion.

5 Experiments and Analysis

Our example benchmarks, code and data for experimental analysis are available².

5.1 Experimental Setup

Our evaluation aims to answer the following research questions:

- (1) **Audit and replay overhead:** How much does Fidelis reduce end-to-end workflow execution time compared to base execution and strawman audit, and what speedup does replay achieve?
- (2) **Per-task efficiency:** How does Fidelis affect individual task execution time, and what fraction of that reduction is attributable to execution context reuse?
- (3) **System throughput:** How does Fidelis affect the number of tasks the system completes per second, and does this improvement hold for dissimilar tasks?
- (4) **Scalability:** Does Fidelis maintain performance stability as the number of tasks and the number of worker nodes increases?

We evaluate Fidelis on five real-world distributed workflows from diverse scientific domains. Table 1 describes the characteristics of each workflow. Our evaluation analyzes three primary performance metrics for both the audit and replay phases: (1) end-to-end workflow execution time, (2) average per-task execution time, and (3) system throughput measured in tasks per second.

We conducted all experiments on an AWS cluster of eight nodes within the same VPC, each running an Intel Xeon Platinum processor at 2.50 GHz with 2 vCPUs and approximately 7.7 GB of memory. Four node run Ubuntu 22.04 and hosts the TaskVine manager; while the remaining nodes run Ubuntu 20.04 and serve as dedicated workers. Each task requests 1 core, allowing up to 2 tasks to run in parallel per worker. Workers do not share a filesystem: each worker maintains its own local storage for dependency manifests and Apptainer container images [10], consistent with the fully distributed audit model described in Section 3.3.

We evaluate Fidelis across three scales defined by input size, which results in a proportional increase in the number of tasks:

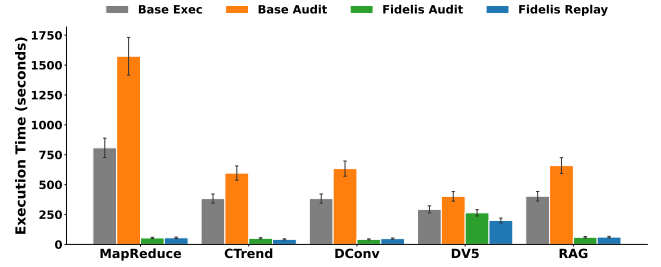


Figure 6: Comparison of total workflow execution time for all workflows.

small, medium, and large. For each scale and workflow, we measure four experimental conditions:

- (1) **Base Execution:** Workflow execution time without any audit capability, establishing the performance baseline.
- (2) **Base Audit:** Workflow execution time under the strawman audit approach — per-task ptrace-based interception with independent container materialization.
- (3) **Fidelis Audit:** End-to-end time to execute and audit the workflow using all Fidelis optimizations.
- (4) **Fidelis Replay:** End-to-end time to replay the audited workflow on the same cluster.

For each condition we measure:

- **Workflow execution time:** End-to-end time at the manager, from first task submission to last task completion. Includes worker connection time, scheduling latency, and intermediate communications. Excludes manager-side pre- and post-processing.
- **Average task execution time:** Time from task execution start to result return, averaged over all tasks in the workflow.
- **System throughput:** Total tasks divided by workflow execution time, measured in tasks per second.

To assess scalability, we additionally vary the number of active worker nodes from one to seven while holding input size fixed at the large scale, measuring how workflow execution time and system throughput scale with the number of workers for each experimental condition.

²<https://github.com/FidelisPro1/Fidelis>

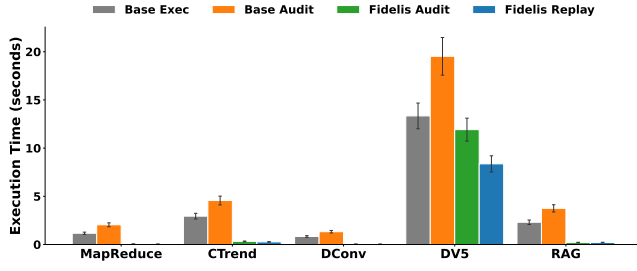


Figure 7: Comparison of average task execution time for all workflows.

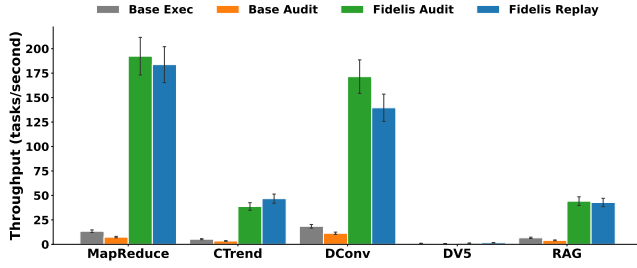


Figure 8: Comparison of system throughput in tasks executed per second for all workflows.

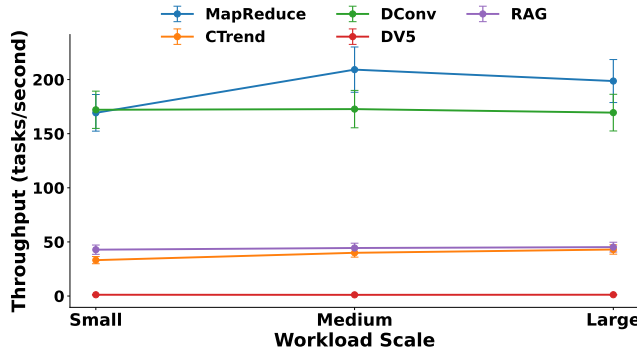
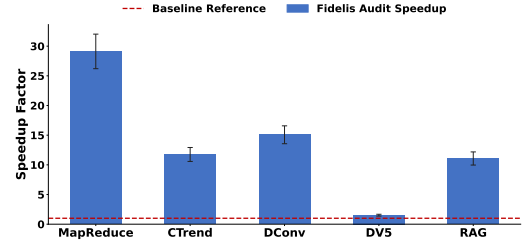


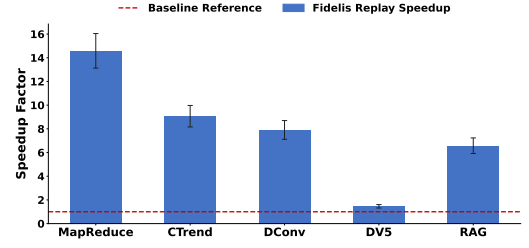
Figure 9: Comparison of system throughput for all workflows as their workload scale increases by increasing the number of distributed tasks.

5.2 Results and Analysis

5.2.1 Q1: Audit and Replay Overhead. Figure 6 shows end-to-end workflow execution time across all five workflows and three scales. Fidelis significantly reduces audit time compared to both base execution and strawman audit. Across the four homogeneous workflows — MapReduce, CTrend, DConv, and RAG — Fidelis audit time is 84–93% lower than base execution and 90–97% lower than base audit. The gains are most pronounced in MapReduce, where all 24,577 tasks share a single module signature: the library task is installed once and reused for every task, eliminating cold-start overhead entirely after the first invocation. CTrend and RAG show consistent reductions of 85–87% over base execution, DConv shows



(a) Speedup factor for Fidelis audit over the baseline across all workflows.



(b) Speedup factor for Fidelis replay over the baseline across all workflows.

Figure 10: Speedup factor for audit and replay using Fidelis over the baseline across all workflows.

89%, all consistent across small, medium, and large scales. Figure 10a shows the audit speedup factor across all workflows; MapReduce achieves up to 28x, while CTrend, DConv, and RAG achieve 5–15x, consistent with their import-dominated task profiles. DV5 shows smaller reductions of 5–33% over base execution under audit and 30–55% under replay. Two factors explain this. First, the physics analysis functions import heavy packages — *awkward*, *fastjet*, *scipy*, *uproot*, and *coffea* — through deeply nested callables beyond the depth that Fidelis’s static bytecode analysis can reach, so these packages are not pre-loaded by the library task. Second, individual DV5 tasks run for 12–14 seconds processing 50,000-event chunks of particle collision data, meaning computation dominates task time and the relative benefit of eliminating initialization overhead is proportionally smaller. Despite these challenges, Fidelis still reduces DV5 audit time by 31–48% over base audit and replay time by 49–65% over base audit, confirming that the two-tier audit strategy provides meaningful gains even for compute-intensive heterogeneous workflows. Figure 10b shows the replay speedup factor. Fidelis replay reduces workflow time by 83–93% over base execution across homogeneous workflows, benefiting from pre-warmed library tasks already loaded from the audit run and the complete absence of audit recording overhead. MapReduce achieves up to 14x replay speedup, CTrend, DConv, and RAG achieve 5–12x.

5.2.2 Q2: Per-Task Efficiency. Figure 7 shows average task execution time across all workflows and conditions. The benefit of execution context reuse is most directly visible here. For MapReduce, average task time drops from 1.15–1.19 seconds under base execution to 0.05 seconds under Fidelis audit and 0.04 seconds under Fidelis replay, a reduction of approximately 96%. For DConv, average task time falls from 0.83 seconds to 0.04 seconds, a 95%

reduction. CTrend shows an 88–90% reduction, with average task time dropping from 2.83–3.13 seconds to 0.32–0.33 seconds. RAG shows a 90–91% reduction, with average task time dropping from 2.29–2.32 seconds to 0.20–0.22 seconds, reflecting that its text processing and indexing tasks are similarly import-dominated. These reductions directly reflect the elimination of shell startup, dynamic linking, Python runtime initialization, and package loading — costs paid only once by the library task rather than once per task. For DV5, the per-task reduction is a more modest 7–16% under audit and 35–41% under replay, consistent with its compute-bound task profile. Critically, per-task execution time is independent of cluster size: these reductions are architectural, stemming from the library task mechanism, and remain stable as the number of workers increases. The end-to-end workflow time reductions confirm this: MapReduce workflow time is 91–93% lower than base execution under replay, CTrend is 88–89% lower, DConv is 85–88% lower, and RAG is 83–85% lower, all consistent across scales.

5.2.3 Q3: System Throughput. Figure 8 shows system throughput in tasks per second across all workflows. Throughput increases substantially under Fidelis as a direct consequence of reduced per-task execution time. MapReduce throughput increases by 1173–1427% over base execution and 2029–2910% over base audit under Fidelis audit, and by 1045–1386% over base execution under replay. CTrend throughput increases by 581–680% over base execution and 905–1135% over base audit. DConv increases by 804–862% over base execution and 1402–1420% over base audit. RAG increases by 542–586% over base execution and 947–1016% over base audit. These gains scale with the ratio of cold-start time to computation time: workflows where initialization dominates show the largest throughput improvements. DV5 shows more modest gains of 6–49% over base execution under audit and 42–120% under replay, consistent with its compute-bound profile where throughput is bounded by physics computation rather than initialization overhead.

5.2.4 Q4: Scalability. Task-level scaling. Figure 9 shows Fidelis throughput as task count increases from small to large scale. Fidelis throughput does not degrade as scale increases — for most workflows it improves, confirming sub-linear growth in audit time with task count. For MapReduce, Fidelis audit throughput increases from 169.27 tasks per second at small scale to 198.63 tasks per second at large scale, even as the number of tasks grows 16x — meaning audit time grows only 13.5x for a 16x increase in tasks. CTrend shows a similar trend, with Fidelis audit throughput increasing from 33.16 to 42.98 tasks per second as tasks scale 4.5x. RAG throughput increases from 42.83 to 45.19 tasks per second as tasks scale 10x, with audit workflow time growing only 9.5x despite 10x more tasks. DConv throughput remains flat at approximately 170 tasks per second across all scales, reflecting that its tasks are fully optimized at small scale with no additional amortization to gain. DV5 throughput remains stable at 1.18–1.21 tasks per second across all scales, consistent with its compute-bound profile. Across all workflows, Fidelis throughput does not degrade as scale increases, satisfying G1.

Node-level scaling. We evaluate how Fidelis throughput scales as the number of active worker nodes increases from one to seven,

holding input size fixed at the large scale. For homogeneous workflows, throughput scales approximately linearly with the number of workers: approximately 28 tasks per second on one worker, 114 tasks per second on four workers, and 199 tasks per second on seven workers for MapReduce under Fidelis audit. This near-linear scaling confirms that Fidelis’s per-worker mechanisms — library task installation, manifest recording, and asynchronous materialization — introduce no cross-worker coordination overhead that would cause throughput to saturate. DV5 shows sub-linear node scaling, consistent with its small task count (64 tasks) relative to the number of workers, where scheduling overhead rather than execution overhead becomes the limiting factor. Base execution and base audit show similar linear scaling, confirming that Fidelis’s overhead model does not alter the scaling behavior of the underlying workflow system.

6 Discussion

Scope and Generality. Fidelis is currently implemented for Python workflows executing on TaskVine and Dask. The core insight that module signatures define audit equivalence classes generalizes to any language with inspectable bytecode or importable dependency graphs. Extending Fidelis to Julia or R would require replacing the Python dis-based signature extractor with an equivalent static analysis for those runtimes, while the library task, manifest, and fingerprinting mechanisms would remain unchanged. The fundamental assumption that makes the approach work is that the workflow system dispatches many invocations of the same function over different inputs a property that TaskVine [24], Dask [21], Parsl [3], and similar function-centric workflow systems share by design. Workflows where every task is structurally unique would see no benefit from equivalence-class-level auditing, though per-task interposition still eliminates ptrace overhead and asynchronous materialization still reduces blocking.

Limits of Static Analysis. The DV5 evaluation exposed the boundary of Fidelis’s static analysis: modules imported inside deeply nested callables or conditionally loaded at runtime do not appear in the static module signature. Per-task interposition catches these as dynamic loads, preserving completeness, but the missing modules must still be loaded per task, reducing the benefit of context reuse. The residual risk is not to correctness but to performance. Adaptive library task expansion, which observing dynamic loads across child tasks and absorbing frequently recurring ones into the library task’s pre-loaded state would address this incrementally, requiring the interposition layer to report dynamic loads back to the manager and trigger library task re-installation when accumulated imports exceed a threshold. We identify this as the most impactful near-term extension of Fidelis.

Security and Trust. Fidelis uses LD_PRELOAD-based interposition, which operates in user mode without elevated privileges. However, LD_PRELOAD can be bypassed by applications that make direct system calls without going through glibc, or by statically linked binaries. Such bypasses would result in missed dependencies, violating completeness. In practice, scientific Python workflows rarely use direct system calls or static linking, making this a theoretical rather than operational concern for the target workload. For

workflows that include native binaries or statically linked components, Fidelis's companion launcher binary and link map scanning provide partial mitigation. Fidelis is designed for cooperative scientific workflows where tasks are trusted, not for adversarial execution environments.

7 Related Work

Distributed Workflow Management. Workflow management systems such as Pegasus [7], Nextflow [8], and Snakemake [17] coordinate task execution across distributed computational environments. Pegasus supports job clustering and data memoization to reduce redundant transfers, but requires users to manually specify task similarity and does not address audit or replay. TaskVine [24] and Dask [21] are explicitly designed to dispatch many invocations of the same function over different inputs, making structural task similarity pervasive by design. Ray [18] introduces stateful actor processes that maintain warm in-memory state across method invocations, reducing initialization overhead. However, Ray actors maintain mutable shared state across invocations, which complicates provenance attribution and makes per-task audit non-trivial. Fidelis differs from all these systems in that it treats task similarity as an audit resource rather than purely an execution resource, automatically discovering equivalence classes and coupling them to provenance capture.

Cold-Start Elimination and Context Reuse. Eliminating cold-start latency through context reuse has been extensively studied in serverless computing. SOCK [19] uses lightweight containers and import caching to reduce Python function startup. Catalyzer [9] achieves sub-millisecond startup through initialization-less booting via process snapshots. RainbowCake [25] proposes layer-wise container caching and sharing, allowing functions with overlapping dependencies to share lower container layers without a full cold start. HotSwap [15] enables inter-function container sharing by reusing containers across functions with compatible dependencies. All of these systems require users or runtime profilers to identify which functions share context, none automatically discovers equivalence classes from task code.

Phung et al. [20] bring context reuse to scientific workflows via TaskVine, formalizing three mechanisms: discovering function context (code, software dependencies, input data, and environment setup), distributing it to workers, and retaining it persistently across invocations. However, context discovery relies on user-specified setup functions: the paper explicitly notes that automatic context detection is out of scope. Fidelis directly addresses both gaps: it automatically extracts module signatures via static bytecode analysis without programmer annotation, and exploits the resulting equivalence classes for both execution efficiency and audit cost reduction simultaneously.

Provenance Capture and Workflow Reproducibility. Capturing workflow provenance for reproducibility has a long history. CDE [13] and ReproZip [5] use ptrace-based system call interception to capture the execution environment of individual programs into self-contained packages. Scinits [11] and its notebook extensions [2] package application dependencies into reusable units using the same interception approach. These tools demonstrate that audit and replay is practical for single-node workloads, but ptrace

overhead and per-task independence make them impractical at distributed scale. Landlord [23] addresses container proliferation in distributed workflows by coordinating shared software environments across workers, reducing storage overhead but requiring manual environment specification. ProvDeploy [14] integrates provenance capture into HPC workflow containerization but again requires users to configure container contents manually. DisProTrack [22] tracks provenance across distributed serverless applications but focuses on data flow rather than execution environment capture. Fidelis differs from all of these in three ways: it captures provenance at the library task level rather than the individual task level, automatically deduplicates shared dependencies via a per-worker manifest with attribution, and produces stable task fingerprints that enable correct replay routing without relying on scheduler state.

System Call Interposition. System call interposition via LD_PRELOAD has been used in security, sandboxing, and monitoring contexts for decades [12]. Recent work on low-level I/O monitoring for scientific workflows [4] uses ptrace with seccomp filters to reduce interception overhead to near-native levels for non-audited calls. Secure namespace-based kernel audit for containers [16] addresses container-level audit without per-process attachment. Fidelis takes a different approach: rather than filtering which calls are intercepted or attaching per-process tracers, it eliminates ptrace entirely in favor of LD_PRELOAD-based interposition that runs in user mode with no privilege-mode transitions. Coverage gaps inherent in user-space interposition — pre-loaded libraries, dynamic linker internals — are addressed through link map scanning at startup and a companion launcher binary, together satisfying P1 across all tested configurations.

8 Conclusion

We present Fidelis, a distributed workflow audit-and-replay middleware that exploits a previously unrecognized connection: the module signature already computed for execution reuse simultaneously defines audit equivalence classes. Fidelis audits once per equivalence class, dispatches similar tasks as forked children inheriting the pre-loaded address space, captures dependencies via in-process interposition with no privilege-mode transitions, and defers container materialization off the task's critical path. Stable task fingerprints computed by normalizing non-deterministic scheduler metadata and resolving upstream references in topological order enable correct replay routing across runs and clusters without relying on scheduler state.

Evaluated across five real-world scientific workflows, Fidelis reduces workflow audit time by 89% over base execution and achieves up to 28x speedup over base audit for homogeneous workflows, with per-task execution time dropping by 93% and system throughput improving by 542–1427% over base execution. Gains are consistent across scale. For the compute-intensive DV5 workflow, Fidelis still outperforms plain execution and base audit despite the limits of static analysis on deeply nested imports, confirming that the two-tier audit strategy is both efficient and complete.

AI-Generated Content Acknowledgment

Some portions of the text in the paper were rephrased with the assistance of AI language models to improve readability. The authors have verified all the content and take full responsibility for it.

References

- [1] 2024. The CMS statistical analysis and combination tool: COMBINE. *Computing and software for big science* 8, 1 (2024), 19.
- [2] Rashid Ahmad and Tanu Malik. 2022. Reproducible computational notebooks via scientific workflow interoperability. In *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE, 311–320.
- [3] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rajesh Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. 2019. Parsl: Pervasive parallel programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.
- [4] Gregory Bauer et al. 2024. Low-level I/O monitoring for scientific workflows. *arXiv preprint arXiv:2408.00411* (2024).
- [5] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational reproducibility with ease. In *Proceedings of the 2016 international conference on management of data*. 2085–2088.
- [6] CMS Collaboration. 2008. The CMS Experiment at the CERN LHC. *JINST* 3 (2008), S08004. doi:10.1088/1748-0221/3/08/S08004
- [7] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. In *Future Generation Computer Systems*, Vol. 46. Elsevier, 17–35.
- [8] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature biotechnology* 35, 4 (2017), 316–319.
- [9] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.
- [10] David Dykstra et al. 2024. *Apptainer: Application containers for Linux*. <https://apptainer.org>
- [11] Dai Fils, Tanu Malik, Mihir Bhatt, and Ravi Bhardwaj. 2017. SciUnits: Reusable research objects. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, 374–383.
- [12] Tal Garfinkel. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.
- [13] Philip J Guo and Dawson Engler. 2011. CDE: Using system call interposition to automatically create portable software packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.
- [14] Liliane Kunstmann, Débora Pina, Daniel de Oliveira, and Marta Mattoso. 2024. ProvDeploy: Provenance-oriented containerization of high performance computing scientific workflows. *arXiv preprint arXiv:2403.15324* (2024).
- [15] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 69–84.
- [16] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. 2021. Secure namespaced kernel audit for containers. In *Proceedings of the ACM Symposium on Cloud Computing*. 518–532.
- [17] Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. 2021. Sustainable data analysis with Snake-make. *F1000Research* 10 (2021).
- [18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.
- [19] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.
- [20] Thanh Son Phung, Colin Thomas, Logan Ward, Kyle Chard, and Douglas Thain. 2024. Accelerating function-centric applications by discovering, distributing, and retaining reusable context in workflow systems. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 1–13.
- [21] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, Vol. 130. 136.
- [22] Utkalika Satapathy, Rishabh Thakur, Subhrendu Chattopadhyay, and Sandip Chakraborty. 2023. DisProTrack: Distributed provenance tracking over serverless applications. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 1–10.
- [23] Tim Shaffer, Thanh Son Phung, Kyle Chard, and Douglas Thain. 2023. Landlord: Coordinating dynamic software environments to reduce container sprawl. *IEEE Transactions on Parallel and Distributed Systems* (2023).
- [24] Thanh Son Phung Slyvets et al. 2023. TaskVine: Managing In-Cluster Storage for High-Throughput Data-Intensive Workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.
- [25] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 335–350.