GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

Application-agnostic Provenance Auditing for HPC Systems Using libc Interposition

---

Maxim Barnstorf

MatrNr: 26713171

First Supervisor: Prof. Dr. Julian Kunkel
Second Supervisor: Dr. Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

ISSN: 1612-6793

May 26, 2026

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Abstract

Reproducibility is fundamental to scientific integrity, and as computational experiments grow in complexity, systematically recording the data consumed, software invoked, and environment in which execution occurred becomes increasingly critical. This challenge is especially pronounced in High-Performance Computing environments, where workflows span multiple nodes, coordinated through job schedulers, and operate at a scale that makes manual documentation entirely infeasible.

Existing Provenance Auditing Tools were designed for general-purpose systems and fail to meet the combined requirements of HPC deployment, namely low overhead, transparency to the user, integration with job schedulers, applicability across a broad spectrum of computations, and compatibility with data lake storage for long-term artifact preservation. A gap analysis of thirty existing tools conducted in earlier work found that no publicly available system satisfies all of these criteria simultaneously.

This thesis presents the design and implementation of a Provenance Auditing Tool purpose-built for HPC. The tool uses `libc` interposition via `LD_PRELOAD` to transparently capture file access operations and process relationships across entire jobs without requiring modifications to user code or applications. Provenance data is stored in a database and made accessible through interfaces for querying prior jobs, generating graphical dependency overviews, and restoring earlier file states to enable rerunning of previous computations, with artifacts preserved in a data lake.

Validation demonstrates correct behavior for C and Python workloads including complex process trees, but also reveals that even minimal instrumentation causes reproducible crashes in highly concurrent applications, indicating that `libc` interposition cannot be assumed universally safe in production HPC environments. Performance evaluation identifies file descriptor path resolution as the dominant source of overhead, a bottleneck shared by comparable tools and addressable by delegating resolution to a background thread. The results motivate a transition toward modern kernel instrumentation as the most promising direction for future work. While kernel instrumentation has been used numerous times for provenance auditing, publicly available approaches were not developed with the specific constraints of HPC environments in mind. Modern mechanisms in this area offer universal applicability and do not interfere with host processes, and could be integrated directly into the processing and storage architecture developed in this thesis.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 5-10% of the entire text
- ☑ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Acknowledgements

I want to thank my supervisor, Lars Quentin, for his invaluable guidance on both the technical aspects and the writing of this thesis, which required a considerable time investment. This work would not have been possible without his support. I would also like to thank Prof. Dr. Julian Kunkel and Dr. Jonathan Decker for making this thesis possible, and Dr. Jonathan Decker for his advice.

Finally, I want to thank my family and my girlfriend for their patience and encouragement throughout this time.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

API Application Programming Interface

CPU Central Processing Unit

eBPF extended Berkeley Packet Filter

GPU Graphics Processing Unit

HPC High-Performance Computing

HTTP Hypertext Transfer Protocol

I/O Input/Output

JSON JavaScript Object Notation

JSONL JSON Lines

MPI Message Passing Interface

OS Operating System

PAT Provenance Auditing Tool

PID Process ID

PPID Parent Process ID

RDF Resource Description Framework

SLURM Simple Linux Utility for Resource Management

SQL Structured Query Language

SVG Scalable Vector Graphics

1 Introduction

This section introduces the thesis by explaining its motivation and purpose, as well as defining its structure and scope. Section 1.1 presents the motivation, detailing why provenance auditing is necessary and why it poses challenges in modern HPC environments. Section 1.2 specifies the goals that this thesis aims to achieve. Section 1.3 describes the contributions resulting from this research. Finally, Section 1.4 provides an outline of the remainder of the thesis.

1.1 Motivation

The ability to reproduce results obtained from scientific experiments is fundamental to the scientific method. To enable the reproducibility of previous experiments, a complete understanding of the conditions under which they were conducted is required. This includes the steps, parameters, and environmental factors that were involved. Scientists have traditionally used laboratory notebooks to document this information.

An increasing share of experimental work is conducted in the digital domain, where a laboratory notebook is insufficient to capture the full picture because software dependencies are more specific, numerous, and deeply interconnected than their physical counterparts, making it simply impractical to track all relevant information manually. Further, reproducing previous computations is becoming increasingly difficult as software stacks grow in complexity and computing hardware continues to advance. Modern data centers exploit ever-increasing thread counts to sustain progress in line with Moore's law [1], and scientists leverage this growing computational power to analyze larger datasets and run more sophisticated simulations than were previously possible in order to detect and predict patterns of greater subtlety and complexity [2]. The result is a growing number of interacting components and layers that no individual can fully comprehend. This effectively turns these computations into a black box.

Take, for instance, the recent advancements in machine learning algorithms, which can identify subtle patterns within massive datasets and have already enabled numerous valuable discoveries that were previously not thought possible. Such sophisticated computations typically involve extremely large datasets, often distributed across millions of files in diverse formats to maximize accuracy, combined with pipelines involving numerous steps such as data formatting, preprocessing, analysis, and evaluation. Distinct software tools with specific version dependencies are required for each step, further increasing the complexity of reproducing the computational workflow [3]. Even small changes in the input data can completely alter the resulting model, leaving no clear indication of how the input differed or how to recover the original result.

To address this, a process called provenance auditing can be applied, which aims to systematically record the dependencies of a given computational result, effectively serving as a digital equivalent of the laboratory notebook. PATs have been developed to automate this process, removing the burden of manual documentation. However, achieving complete reproducibility for digital computations remains nearly impossible in practice, and PATs therefore typically focus on capturing the subset of information most relevant to their intended use case. The approaches taken vary widely, ranging from injecting API calls directly into application source code to providing workflow managers with built-in auditing capabilities to instrumenting the kernel itself. Each involves different trade-offs between

the granularity of the recorded provenance, the degree of integration required with the computational environment, and a range of other practical factors [4].

Since modern scientific computations are increasingly resource-intensive, a large proportion of them are carried out on HPC clusters. In addition to the general complexity of the computation itself, HPC environments introduce a unique set of obstacles that distinguish them from traditional computing systems. They are highly sensitive to the performance overhead that a PAT might introduce, and they involve sophisticated execution abstractions not commonly encountered elsewhere. These include job schedulers such as SLURM [5], concurrent execution distributed across multiple nodes, and containerized environments such as Apptainer [6], all of which complicate the task of provenance capture.

Despite the critical importance of provenance tracking for scientific reproducibility and accountability, provenance auditing, both manual and automatic, remains almost entirely absent from HPC environments. Very few PATs have been developed with the specific requirements of HPC systems in mind [7]. To drive broader adoption, provenance tracking must be made both easier to deploy and more effective in practice for system administrators and end users alike.

A PAT purpose-built for HPC could yield further benefits beyond improving the reproducibility and transparency of computational research. Such a tool could also provide valuable logging capabilities that support compliance with regulatory and institutional standards. This thesis aims to address this shortcoming by presenting a PAT specifically designed to operate within HPC environments, capable of overcoming the unique challenges these systems present while delivering meaningful provenance information with minimal impact on system performance.

1.2 Goals

The overarching goal of this thesis is to design and implement a PAT that can be broadly deployed within HPC environments to enable reproducibility of computational results. Several interrelated properties that address the technical, operational, and usability constraints inherent to HPC systems are required to achieve this.

HPC-native integration
To operate effectively in HPC environments, the PAT must integrate seamlessly with the existing HPC infrastructure. Integration with the workload manager responsible for job scheduling and resource allocation, such as SLURM, is necessary, as all compute resources are allocated through it. The PAT must be capable of understanding computations in the context of workload manager jobs, which involves recording provenance on a per-job basis, separating computations into their constituent job steps, and capturing user-level submission workflows. Provenance should be modeled at the level of HPC job abstractions so that the auditing aligns naturally with the compute infrastructure of the cluster.

Usability
Ease of adoption is a critical consideration for real-world deployment. A technically sound system that imposes significant workflow disruption will simply not be used. The PAT should therefore integrate transparently into existing research workflows, requiring minimal modifications to user code. Researchers should be able to benefit from the system

without rewriting applications, learning complex new interfaces, or recompiling code, so that even closed-source tooling remains fully supported.

This low-friction design philosophy is required for adoption in production HPC environments, where researchers are under pressure to deliver results and have little tolerance for tools that slow down their work. Only by meeting users where they are rather than demanding they adapt to the system could the PAT actually be used in the real world.

Performance efficiency
HPC systems are highly sensitive to performance degradation, as even small overheads can translate into significant costs at scale. Consequently, the performance penalty introduced by the PAT must be minimal and carefully evaluated. The design should prioritize high performance to ensure that provenance collection does not meaningfully interfere with computational throughput or resource utilization.

Reproducibility of Inputs
The goal of this PAT is to enable reproducibility of the computations performed. While complete reproducibility of computations is realistically nearly impossible to achieve, capturing all input files, including scripts and executables, provides the most important data points required for reproducibility. By systematically recording these dependencies together with relevant execution metadata, the essential context required to reconstruct the computation could be preserved, although smaller factors such as hardware variations or nondeterministic behavior could still limit reproducibility somewhat.

Meaningful outputs
Finally, the collected provenance information must be transformed into outputs that are meaningful and practically useful for researchers. This includes generating visual representations of data dependencies and process interactions, enabling efficient querying and retrieval of prior artifacts, and restoring intermediate states within a job.

1.3 Contribution

The contributions of this thesis are as follows:

- **Analysis of existing PATs:** A taxonomy and gap analysis of existing PATs with respect to their applicability in HPC environments
- **SLURM compatible:** The design of an HPC-native provenance auditing workflow integrated with the SLURM workload manager
- **PAT uses libc interposition:** The implementation of an `LD_PRELOAD`-based injector leveraging `libc` interposition for automated auditing
- **Provenance Data Management:** The development of a backend system for processing, interpreting, and persistently storing provenance data
- **Visualizing Provenance Data:** The design of analysis software that generates artifact and process dependency graphs for executed jobs
- **Provenance Artifact Storage:** The implementation of automated staging and recovery of job-related artifacts via a data lake architecture

- **Verification Benchmarks:** The design of a verification test suite to evaluate the capabilities and limitations of the proposed PAT
- **Performance Evaluation:** A performance evaluation of the injector overhead using fio [8] and perf [9]

1.4 Outline

Section 2 provides the background necessary to understand the PAT and the environment in which it operates, covering data provenance and provenance auditing, HPC architecture and job scheduling with SLURM, data lake storage, and `libc` dynamic linking and library preloading. Section 3 defines the requirements a PAT must satisfy for practical HPC deployment, surveys thirty existing PATs against these criteria, and concludes with a gap analysis motivating the need for a new tool. Section 4 presents the design and implementation of the PAT, detailing the selection of `libc` interposition as the auditing method, the individual components and their design rationale, and how they interact. Section 5 evaluates the correctness and coverage of the PAT through a set of targeted benchmarks spanning synthetic hook tests across multiple programming languages, fork and exec propagation, concurrent execution, and real-world HPC workloads. Section 6 evaluates the performance overhead introduced by the injector using fio and perf, identifying file descriptor path resolution as the dominant source of overhead. Section 7 discusses the applicability and performance limitations identified by the benchmarks and outlines kernel instrumentation as a more promising alternative approach. Finally, Section 8 concludes the thesis and identifies directions for future work.

2 Background

This section provides the necessary background to understand how the PAT operates and the environment in which it functions. Section 2.1 explains provenance auditing, including how it can be performed, what can be audited, and the potential benefits and drawbacks. Section 2.2 outlines the structure of HPC systems and details how compute resources are allocated to user jobs. Section 2.3 provides an overview of data storage in HPC systems, explaining how data lakes function, how data is organized within them, and how it is stored. Section 2.4 introduces `libc` *and details how library preloading can be used to dynamically modify program behavior. Finally, Section 2.5 provides a conclusion of this section.*

2.1 Data Provenance and Provenance auditing

2.1.1 Data provenance

In computer science, provenance denotes the complete set of information describing how the components of a computation interact to produce a specific result. This includes the data consumed, the programs executed, and the contextual environment in which execution occurs. Access to such information supports the reproducibility of prior computations. The main application targeted in this thesis is to allow researchers to build on existing computations and understand how changes to code, parameters, or other dependencies affect the resulting output. It also helps reduce reliance on implicit trust. In addition, systematic provenance capture can assist in diagnosing errors.

From a practical perspective, provenance capture requires recording input data, invoked executables and their versions, the values of the environment variables, and relevant aspects of the execution context. The latter may include kernel versions and linked system libraries. Complete reproducibility is therefore difficult to achieve. In practice, provenance auditing systems focus on capturing the most critical and accessible provenance data, such as input files, executed binaries, and environment variables, to provide reproducibility.

2.1.2 Provenance Auditing

Provenance auditing refers to the systematic acquisition of provenance data. In many cases, this task is still performed manually by users. This approach is both error-prone and labor-intensive. Dependencies can easily be overlooked, required artifacts may not be archived or archived correctly, and documentation effort scales poorly with mounting workflow complexity. Automated auditing mechanisms integrated into computational infrastructures therefore represent a more reliable alternative.

2.1.3 Types of Provenance

Since complete reproducibility is rarely attainable, PATs differ in scope and granularity. Many systems capture only a subset of potentially relevant information, tailored to specific objectives. Auditing can occur at different abstraction levels within a computation, resulting in completely different provenance data.

Existing PATs target diverse layers of the computational stack. Some intercept and interpret SQL queries. Others integrate with workflow managers. Still others require manual instrumentation within application code to provide semantically rich context.

Additional approaches rely on intercepting specific libraries to access domain-relevant information. These methodological differences reflect the heterogeneity of provenance requirements [4]. Broadly, provenance scope can be categorized as follows:

- **Data-level provenance:** Capturing accessed data artifacts, such as files or database queries, from which dependencies can be inferred.
- **Process-level provenance:** Monitoring internal data flow and states within processes to enable fine-grained analysis and debugging.
- **Workflow-level provenance:** Auditing the execution of workflows and the processes started within them.
- **System-level provenance:** Observing low-level system interactions, including library calls or kernel events, to obtain comprehensive coverage across processes and their descendants.

While the list of these categories includes the majority of established systems, other domain-specific methods exist [7].

Benefits and Use Cases Provenance auditing provides several distinct benefits. At fine granularity, intra-process monitoring enables inspection of internal program behavior, including state transitions and data dependencies. In such cases, PATs function similarly to typical debugging instruments which help with error detection and performance optimization.

At a broader level, provenance enhances transparency and accountability in scientific research. All data and computational steps underlying published findings can be traced and audited, which can provide validation and reproducibility. Further, detailed provenance records aid follow-up studies by clarifying methodological decisions and identifying potential sources of divergence. It could also provide the audit trail required for compliance purposes, which is often necessary when working with sensitive data.

Finally, systematic provenance capture helps ensure that computational experiments can be reproduced reliably. Research workflows most commonly involve iterative modification of prior experiments. Automated recording prevents the loss of critical configuration details over time and removes the risks associated with undocumented changes.

Limitations Every PAT operates within a constrained scope. Approaches that provide deep visibility into a single process may lack awareness of external interactions, while system-level monitoring captures inter-process activity but lacks insight into internal program state. Applicability is often restricted by language, runtime, or workflows. More universal approaches typically trade fine-grained, internal process information for broader coverage across multiple processes.

Drawbacks Provenance auditing inevitably introduces overhead. Performance penalties and storage requirements vary substantially across methods. Given the high computational and energy costs of large-scale scientific workloads, excessive runtime overhead can render systematic auditing economically infeasible. Storage demands also increase when complete input and output datasets must be archived to ensure reproducibility.

User overhead constitutes an additional concern. Approaches that require manual instrumentation or configuration impose cognitive and operational burdens. This will stunt adoption and create the opportunity for errors. Transparency is therefore a most critical property for provenance systems intended for scalable and widespread deployment.

2.2 HPC Usage and Compute Workflow

HPC systems combine many compute nodes into a coordinated environment to provide large-scale computational power. The PAT presented in this thesis is designed to function in HPC environments. This section only covers the aspects of HPC required to understand how and where the PAT operates.

2.2.1 HPC Architecture

An HPC system consists of clusters, which in turn consist of many interconnected nodes. There are two types of nodes: login nodes, which represent the entry point of a cluster, where users can access the system, compile code, manage files, and run jobs, and compute nodes, where the actual computations are performed and which are optimized to handle high-performance workloads. Depending on the computational needs, clusters may provide both GPU and CPU nodes.

Further, there is a shared parallel file system functioning as the storage. This storage system provides data access across all nodes. And finally, there is a low-latency network that connects the nodes. With this distributed architecture, users can run jobs across several nodes in parallel.

2.2.2 SLURM Workload Manager

Resource allocation and job scheduling on the cluster are managed by SLURM [5]. SLURM is responsible for allocating resources to jobs queued by users.

SLURM jobs are most commonly submitted as bash job scripts that load necessary dependencies through `module load` and launch the target application, though other files such as Python scripts can be submitted directly as well. Submitted jobs enter a queue and are scheduled according to resource availability and priority policies. Once resources become available, the job is executed on the compute nodes assigned by SLURM.

2.2.3 MPI

When a computation is distributed across multiple nodes, the processes running on each node must be able to communicate and coordinate with one another. MPI [10] is the standard communication model used for this in HPC environments.

2.3 Data Lakes

A data lake is a storage system that retains data in its raw format regardless of structure, accommodating files of arbitrary type and layout [11]. Unlike data warehouses, which are analytical database systems requiring data to be transformed and loaded into a predefined schema before it can be queried [12], a data lake applies structure only at read time. This schema-on-read design makes data lakes suitable for storing heterogeneous collections of files without requiring any upfront knowledge of how they will be used. A data catalogue

is typically used alongside a data lake to make its contents discoverable. It maintains metadata records that describe what is stored, where it is located, and how it relates to other assets. This provides the indexing layer that raw file storage alone cannot provide. The increasing use of large-scale data processing, such as machine learning workloads in HPC environments, makes this flexibility necessary.

Data is stored in the data lake through a staging process. This process may involve cleaning, validation, and normalization. Importantly, staging is also the step during which datasets are registered in the data catalogue. During staging, the catalogue is provided with both manually curated and automatically generated metadata, which is used to register and link the dataset to its physical storage location. This registration process is commonly referred to as indexing.

2.4 Libc, Dynamic Linking, and Library Preloading

One of the primary ways programs execute low-level system operations in Linux is through functions in the C standard library `glibc`. The library is attached to the processes through dynamic linking [13].

2.4.1 System Calls

A system call is the mechanism by which a user-space program requests a service from the operating system kernel. Operations such as reading a file, spawning a process, or allocating memory cannot be performed directly by user-space code, as they require privileged access to hardware or kernel internals. Instead, the program issues a system call, which transfers control to the kernel, which performs the requested operation and returns the result.

2.4.2 The C Standard Library libc

The C standard library provides low-level functions for operations such as memory management using malloc and free, file and I/O operations, process control, string manipulation, and interaction with the operating system through system call wrappers.

Applications written in C and many higher-level languages like Python use `libc` to interact with the operating system. `libc` is dynamically linked at runtime.

2.4.3 Dynamic Linking

With dynamic linking, shared libraries are loaded and linked to an executable at runtime rather than at compile time. The dynamic loader resolves symbol references and loads the required shared objects into the process address space. There are multiple reasons for using dynamic linking. It reduces the size of program binaries, since shared libraries do not need to be included in the executable itself. It is also more memory efficient, since many processes can share the same library instance. In addition, it provides flexibility because a library can be updated independently of the applications that use it.

When the program starts, the dynamic loader determines the dependencies of the application, locates the required shared libraries, resolves symbols, and performs relocation. Control is transferred to the entry point of the program after this process has been completed.

2.4.4 Environment Variables

Environment variables are key-value pairs associated with a process and supplied as part of its execution environment. They can be used to configure program behavior at runtime without modifying the executable itself. In Linux systems, child processes inherit the environment of their parent unless it is changed explicitly. This makes environment variables a convenient mechanism for propagating configuration through a process tree.

2.4.5 Library Preloading

Because `libc` is dynamically linked, its functions can be overridden at runtime without modifying or recompiling the target program. This is achieved through library preloading, which on Linux is enabled by setting the `LD_PRELOAD` environment variable to the path of a custom shared library. Libraries specified via `LD_PRELOAD` are loaded before all others, giving their functions priority over those of the standard library. An overriding function can perform additional work, such as logging, and then delegate to the original implementation. This makes it possible to instrument a program transparently without altering its behavior. Since child processes inherit environment variables from their parent, `LD_PRELOAD` propagates automatically through the process tree, extending instrumentation to all descendant processes automatically. Beyond provenance auditing, common applications of this mechanism include security [14] and sandboxing [15].

Since `libc` is the universal wrapper through which the vast majority of programs interact with the kernel, interposing its functions is the only user-space mechanism capable of intercepting system calls without directly auditing them. Programs written in C, C++, Python, and other languages route their I/O and process operations through `libc`, which makes preloading possible for them. Libc interposition cannot be used on programs that bypass `libc` and issue system calls directly, such as those written in Go, or on statically linked binaries for which dynamic linking is not used at all. Outside of these cases, `libc` interposition provides an effective interception mechanism with no equivalent in user space.

2.5 Conclusion

This section has established the foundational concepts underlying the PAT, covering data provenance auditing and its trade-offs, HPC architecture and job scheduling with SLURM, data lake and catalogue storage, and library preloading as a mechanism for transparent interception. These are all the relevant concepts required to understand the purpose and design of the PAT.

3 Related Work

Provenance auditing can be implemented in many ways, and a variety of PATs have been developed over the years to exploit these possibilities. This section reviews PATs from the academic literature by categorizing and evaluating their approaches relative to the goals defined earlier. Section 3.1 details the requirements for building a PAT that delivers the specified features. Section 3.2 outlines a taxonomy used to distinguish the approaches employed by different PATs to collect data. Section 3.3 summarizes the performance of various PATs against relevant criteria. Section 3.4 presents a gap analysis, illustrating why existing PATs cannot fully address the challenges of deploying provenance auditing at scale in HPC systems. Finally, Section 3.5 concludes the findings.

3.1 PAT Requirements

The requirements for a PAT completely depend on the intended use case. HPC environments have especially strict requirements due to their scale, performance sensitivity, and execution complexity. Systems not explicitly designed for HPC are therefore almost certainly unsuitable in practice. To enable deployment at scale within HPC infrastructures, a PAT should satisfy the following properties:

- **Low overhead:** HPC workloads are highly performance-sensitive. Many existing PATs introduce runtime overhead that is unacceptable in such environments, as performance optimization is often not their primary design objective. Overheads of 15–20% are not uncommon, even for established systems such as SPADEv2 [16]. For practical adoption in HPC, the overhead should ideally remain below 1%.

- **Transparency:** The intended use case assumes large-scale auditing in which users are not expected to modify their code or engage with the auditing process directly. Interaction with the PAT should therefore be minimal. Ideally, users invoke the PAT once at the start of a job and provide basic contextual information, with no further changes to their application logic required.

- **Adaptation to HPC execution abstractions:** Provenance tracking in HPC introduces challenges that are uncommon in regular systems. A central abstraction is the parallel execution of a single job across multiple compute nodes. Provenance must therefore be tracked on a per-job rather than per-node basis. Additionally, containers are widely used in HPC. Provenance capture in HPC is impossible without the explicit support of these abstractions.

- **Integration into Workload Managers:** In HPC environments, all relevant computations are initiated through a workload manager. Because of this, it is required to integrate the auditing mechanism with the workload manager in order to reliably identify the processes that belong to a specific job, even when execution spans multiple nodes.

Beyond HPC-specific constraints, a universally applicable PAT that supports reproducibility in scientific computing should satisfy additional requirements:

- **Wide range of applicability:** The PAT should support diverse scientific workloads, including different programming languages, libraries, and environments.

- **Granularity:** To enable reproducibility, the PAT must capture sufficiently detailed information, including all input data and operations performed by child processes spawned during job execution.
- **Integration with a data lake:** Capturing dependency information alone is not enough for long-term reproducibility. Relevant artifacts and metadata must be stored permanently, ideally within a data lake, to ensure reliable provenance and lasting records.

3.2 Provenance Auditing Methods

Multiple methodological approaches exist for collecting provenance data, each characterized by distinct trade-offs. The following taxonomy follows prior work [7]:

- **Compute platform–specific PATs**: Workflow systems or computational platforms in which provenance capture is integrated by design, including tools developed for environments such as Snakemake [17], Jupyter [18], or Hadoop [19].
- **API-based PATs**: Approaches requiring explicit insertion of auditing calls into application code.
- **Library wrappers**: Techniques that override existing library functions to incorporate logging while preserving original functionality. Coverage depends on the scope of the wrapped library.
- **Kernel instrumentation–based PATs**: Mechanisms that intercept system calls at the kernel boundary, using tools such as strace [20], SystemTap [21], or eBPF [22], which capture operations together with process identifiers.

3.2.1 Compute Platform-Specific PATs

Compute platform-specific PATs integrate provenance auditing directly into existing compute platforms, most commonly workflow management systems. Many workflow managers have been explicitly designed with provenance capture as a core objective. In addition, this category includes PATs tailored to widely used computational environments such as Snakemake, Jupyter, and Hadoop. The concrete auditing mechanism depends on the respective PAT and the abstractions provided by the underlying platform.

Existing PATs in this category include ZOOM [23], Vistrails [24], Kepler [25], PLUS [26], REDUX [27], Chimera [28], myGrid/Taverna [29], Dataprov [30], Wings-Pegasus [31], RAMP [32], ProvBook [33], and Karma [34]. Dataprov targets Snakemake-based workflows, RAMP is designed for Hadoop environments, and ProvBook focuses on Jupyter Notebooks. The remaining systems are either integrated into workflow managers or function as standalone systems that natively support provenance capture and were specifically designed for provenance auditing.

Compute platform-specific PATs can leverage structural information explicitly defined by the user, such as declared dependencies and workflow structure. Computations are often split into multiple steps, which can also be represented as a directed acyclic graph, with explicitly specified inputs, outputs, and dependency relations. This structured representation provides insight into the computational process that is not readily accessible

to platform-agnostic systems. Furthermore, provenance collection can be simplified, as the workflow specification itself can be parsed to extract relevant provenance information.

Dataprov illustrates this approach. Snakemake is widely used within the scientific community to define reproducible workflows. In Snakemake files, users explicitly specify computational steps and the corresponding input and output files. Dataprov performs a Snakemake dry run to capture dependency and execution information without executing the workflow itself. This approach introduces no additional runtime overhead and remains transparent, as the provenance information is already defined independently of the provenance auditing mechanism. However, this also limits the approach, as it can only capture provenance information that is available through static pre-execution analysis and may therefore miss runtime-dependent behavior.

3.2.2 API-Based PATs

API-based PATs provide an application programming interface that must be explicitly invoked within the user's program to record provenance-relevant events. Through these API calls, developers notify the PAT of operations that should be tracked.

This approach enables highly customized and fine-grained auditing tailored to specific application requirements. However, it also requires manual integration into the codebase. Furthermore, there is a significant risk of incomplete provenance coverage, as users may fail to instrument all relevant operations. While individual systems differ in language support and implementation details, they generally follow the same principle of explicit instrumentation through API calls.

Existing PATs in this category include PReServ [35], ProvLake [36], AiiDA [37], Core Provenance Library [38], and RDataTracker [39].

3.2.3 Library Wrappers

Library wrapper–based PATs integrate provenance auditing capabilities into existing libraries used by the computation. This is typically achieved by overriding or wrapping selected functions to include logging functionality. The scope and granularity of the captured provenance therefore depend on the instrumented libraries and the functions being intercepted.

Existing PATs in this category include SPROV [40] and PROV-IO [41].

PROV-IO serves as a representative example and was specifically designed for HPC environments. Rather than being built to work with one or a few libraries, it can be integrated into C and Python libraries, enabling adoption across a wide range of scientific applications. Reported evaluations indicate that the runtime overhead is low in most cases. The system has already been integrated by its developers into selected HPC-oriented libraries to demonstrate its capabilities. By contrast, SPROV wraps `stdio.h` to capture I/O performed specifically by C code.

3.2.4 Kernel Instrumentation–Based PATs

When a process wants to modify a file, it will send a request to the kernel, which performs the low-level operation on the hard drive. Kernel instrumentation–based PATs audit these operating system–level events to detect these operations. This approach is transparent to the application and agnostic to the programming language or runtime environment.

Such systems rely on mechanisms for inspecting system calls, including tools such as strace or SystemTap. Observed operations can be associated with specific processes via their PIDs. To reconstruct complete provenance, it is necessary to track process hierarchies, including child processes created through fork operations.

Existing PATs in this category include LPS [42], SPADEv2 [16], PASSv2 [43], ES3 [44], and CamFlow [45].

3.3 Survey of Existing PATs

Previous work included a survey of 30 PATs, which were evaluated with respect to their suitability for deployment at the GWDG based on the criteria of overhead, transparency, scope, and applicability, where applicability is defined as a combination of transparency and scope. The systems were also classified according to the taxonomy introduced in the previous subsection, which included several additional niche categories omitted here for conciseness, namely User Space, Augmented Language, Language Interpreter, and Provenance Language [7]. The results of this assessment are presented in Table 1.

System	Overhead	Transparent	Scope	Applicability	Category
AiiDA [37]	-	No	◐	○	API
BURRITO [46]	-	Yes	○	○	User Space
CamFlow [45]	12–23%	Yes	●	●	Kernel Instrumentation
Chimera [28]	/	Yes	○	○	Compute Platform
CPL [38]	-	No	○	○	API
Cui 2000 [47]	-	Yes	○	○	Augmented Language
CXXR [48]	-	Yes	○	○	Language Interpreter
Dataprov [30]	None	Yes	◐	◐	Compute Platform
ES3 [44]	-	Yes	●	●	Kernel Instrumentation
Kepler [25]	/	Yes	○	○	Compute Platform
Karma [34]	-	Yes	○	○	Compute Platform
Lipstick [49]	16-35%	Yes	○	○	Augmented Language
LPS [42]	1%	Yes	●	●	Kernel Instrumentation
myGrid/Taverna [29]	-	Yes	○	○	Compute Platform
PASSv2 [43]	23%	Yes	○	○	Kernel Instrumentation
Perm-GProM [50]	-	Yes	○	○	Augmented Language
PLUS [26]	/	Yes	○	○	Compute Platform
PReServ [35]	10%	No	○	○	API
PROV-IO [41]	0.02–11%	Yes	◐	◐	Library Wrapper
ProvBook [33]	None	Yes	◐	◐	Compute Platform
ProvLake [36]	1%	No	◐	○	API
RAMP [32]	26–75%	Yes	◐	◐	Compute Platform
RDataTracker [39]	-	No	○	○	API
REDUX [27]	/	Yes	○	○	Compute Platform
SPADEv2 [16]	12%	Yes	●	●	Kernel Instrumentation
SPROV [40]	12–16%	Yes	◐	◐	Library Wrapper
Swift [51]	/	Yes	○	○	Provenance Language
VisTrails [24]	/	Yes	○	○	Compute Platform
Wings-Pegasus [31]	/	Yes	○	○	Compute Platform
ZOOM [23]	/	Yes	○	○	Compute Platform

Table 1: Performance overhead, transparency, scope, applicability, and category of various provenance systems.

Legend: *Overhead:* - = Not measured, / = Not measurable
Scope, Applicability: ● = Broad, ◐ = Partial, ○ = Limited.

In this previous work [7], we concluded that all PATs included in the survey except for LPS, Dataprov, and ProvBook were unsuitable for broad deployment at GWDG, as they were either insufficiently performant, lacked transparency and therefore imposed additional burdens on users or system administrators, or did not provide the scope required to support most computations performed in HPC environments end to end. Under these constraints, only these three systems were considered applicable across GWDG at scale.

3.4 Gap analysis

While the previous analysis focused on PATs that could generally be applied at GWDG based on performance, transparency, and compatibility with existing workflows, the focus of this thesis is more specific, namely the identification of an ideal provenance auditing tool rather than one that is merely deployable. Accordingly, the requirements have been refined further. In particular, we additionally require the ability to integrate with SLURM, be applicable across almost all scientific computations, record close to all relevant operations, and be adaptable such that the resulting provenance artifacts can be backed up in a data lake. On this basis, while the three PATs LPS, Dataprov, and ProvBook were previously evaluated as potentially usable, we now perform a gap analysis to assess their suitability with respect to these additional requirements.

3.4.1 Overview of selected PATs

In order to better understand what makes the previously selected PATs suitable or unsuitable given our focus, this section will provide a basic overview of each PAT and why we considered that it could be deployable.

ProvBook ProvBook [33] records the code executed in each Jupyter Notebook cell together with its corresponding output and allows users to conveniently inspect previous inputs and outputs directly within the notebook environment. It supports navigation through the full execution history via a slider, which makes provenance data readily accessible and immediately useful in practice. This low-friction access to provenance information is an important factor for adoption. In addition, notebooks, together with their provenance data, can be exported to RDF and later reconstructed from that representation. Although the overhead has not been explicitly evaluated, it is most likely very low, since the system simply stores the input code and output produced by each cell execution.

Given the widespread use of Jupyter Notebook in scientific computing environments, ProvBook could possibly be applied quite well in the GWDG. Its integration directly into the Jupyter user interface also makes it particularly easy and intuitive to use. The ability to inspect previous versions of individual cells can be highly valuable, as it helps users understand how the changes they made in the code relate to the resulting outputs directly within the notebook. On the other hand, there is no high-performance overhead that would make this PAT expensive to use.

Dataprov Dataprov [30] provides provenance auditing for Snakemake with effectively no relevant runtime overhead and complete transparency, since it relies on a Snakemake dry run rather than instrumenting the workflow during execution. It records the declared input and output files, the programs involved together with their versions, the kernel version, and user-related information entered manually. Much of this information could

in principle also be preserved by backing up the Snakemake file for each run, but Dataprov extracts it and structures it in XML, which makes subsequent storage in a database more straightforward. Dataprov also supports additional auditing methods, but these are not considered here, as they are less broadly applicable in scientific computing environments than its use with Snakemake.

Like Jupyter Notebook, Snakemake is widely used in scientific computing environments, including the GWDG. It is able to capture a large amount of useful low-level information, including accessed files and invoked programs, while remaining fully transparent to the user and requiring little active involvement in the auditing process. In addition, its performance overhead is negligible.

LPS LPS [42] is a provenance auditing tool for HPC environments that captures provenance by instrumenting kernel-level system calls using SystemTap [21]. Rather than relying on application-specific integration, it observes low-level process and file system activity directly at the operating system level. This makes it transparent to users and applicable across a wide range of workloads. In doing so, it records three categories of events that together establish a detailed history of the files a process reads, writes, or otherwise modifies.

- **Execution:** LPS records events related to process creation, execution, and termination.
- **File access:** Open, close, read, and write operations are recorded to track file accesses and modifications.
- **Metadata:** Rename, link, unlink, and related file system operations are recorded to capture changes to file names and file system structure.

To determine which events belong to a given computation, LPS uses the original process as a root node and constructs a process tree whose child nodes represent all spawned processes. In HPC environments, users typically launch computations from login nodes via SSH and a shell, while execution on compute nodes proceeds through shells or runtime libraries such as MPI launchers. LPS identifies these origin processes, builds the corresponding execution tree, and records all file-related activity triggered by descendant processes, meaning that even parallel and distributed executions can be traced back to the original root process.

The collected events are not stored as isolated records but are first filtered and aggregated before being fused into a provenance graph. Information collected across login and compute nodes is combined by using scheduler-provided environment variables to associate local processes with their corresponding higher-level jobs. This mechanism is particularly relevant for integration with job schedulers, as it allows provenance from processes distributed across multiple nodes to be linked back to the originating job submission. The paper describes this mechanism using scheduler-exposed job identifiers in environment variables, for example `PBS_JOBID` in PBS [52]. Although SLURM is not explicitly discussed in the original work, the approach appears generally adaptable to it, since it depends on scheduler-provided job identifiers rather than on any scheduler-specific mechanism. Finally, to balance provenance completeness against runtime overhead, LPS supports multiple provenance granularities for file access tracking that can be switched at runtime. At coarser granularities, only the first and last access within a file interaction are recorded rather than every individual read and write operation.

LPS appears to be the most promising of the three systems. Because it operates at the system-call level, it can be applied across any computation. The paper reports overheads

that usually remain below 1%, and the approach is fully transparent to users, to the point that no active involvement or awareness on their part is required. It was explicitly designed for HPC environments, which is reflected in its support for distributed execution across multiple nodes and its integration model based on job schedulers used in HPC.

3.4.2 Assessment Against Additional Requirements

Beyond the baseline properties required for broad deployment at GWDG, namely low overhead, transparency, and applicability to a reasonable range of computations, the goal of this thesis to provide an ideal PAT for the GWDG requires additional capabilities, specifically the ability to handle HPC-specific execution abstractions such as multi-node execution and job-scheduler-aware provenance capture, applicability to a large majority of scientific workloads rather than only a subset, and support for integration with a data lake for persistent artifact storage.

Evaluated against these extended requirements, Dataprov and ProvBook are ruled out. Both are confined to a single platform, Snakemake and Jupyter Notebook respectively, and neither integrates with job schedulers. ProvBook does not record file dependencies at all, as it is designed only to capture the execution history of notebook cells, making data lake integration impossible by design. Dataprov does record input files and executables declared in the Snakemake workflow, but its coverage is limited to what is explicitly specified there. If the workflow invokes a child process that accesses additional files outside the Snakemake specification, those accesses go unrecorded, meaning the dependency information is incomplete and insufficient to fully support data lake integration.

LPS, on the other hand, was designed with goals closely aligned to those of this thesis, operates at the system-call level and is therefore applicable to any computation, explicitly supports distributed execution across multiple nodes, integrates with job schedulers to associate provenance records with the corresponding jobs, and produces structured dependency records that could be forwarded to a data lake ingestion pipeline without fundamental architectural changes. Unfortunately, LPS is not publicly available, which makes adoption impossible and directly motivates the design and implementation of a new PAT in this thesis.

3.5 Conclusion

This section defined the requirements a PAT must satisfy for practical HPC deployment, introduced a taxonomy of auditing methods, and surveyed existing PATs drawing on prior work evaluating 30 systems. A gap analysis was then applied to the three potential candidate systems from that prior work, of which only LPS met the requirements of this thesis. Unfortunately, its source code is not publicly available.

4 Methodology

This section presents the design and implementation of the PAT, explains the rationale behind the key design decisions, and describes the functionality of its individual components. Section 4.1 compares the available auditing methods and motivates the selection of `libc` *interposition. Section 4.2 introduces the overall design of the PAT. Sections 4.3 to 4.9 describe the individual components, namely the injector, the Prov Tool, the backend, the querier, the retriever, the visualizer, and the configuration. Section 4.10 explains how these components interact during provenance capture, querying, retrieval, and visualization. Finally, Section 4.11 summarizes the overall design.*

4.1 Comparison of Auditing Methods

The first step in designing a PAT is selecting the auditing method. With an understanding of the available approaches for collecting provenance data, each method is assessed against the requirements. Rather than comparing the performance of existing PATs, the focus when choosing a method is on what each method enables and constrains.

4.1.1 Compute-Platform-Specific PATs

PATs in this category are tailored to a particular compute platform, which limits their general applicability. Their primary advantage is the ability to leverage workflow-specific information provided by the user. This often ensures complete transparency and minimal overhead. Furthermore, platform-specific knowledge can help give context to operations within the workflow. However, such PATs are restricted to the original job script and operate at a high level and generally cannot track child processes spawned by the workflow. The approach would also require the users to use the particular platform explicitly, which is not reasonable.

4.1.2 API-Based PATs

API-based PATs provide fine-grained control over provenance capture in one or more programming languages. While the performance impact can be small, their use requires considerable expertise in programming and specific knowledge of the codebase. Injecting API calls into large scientific codes, often tens of thousands of lines, can be highly complex and potentially create errors. On the flip side, if applied correctly, they provide highly precise and relevant provenance information tailored to the specific needs, enabling a level of relevance that automated systems could not provide. However, these methods cannot reliably capture operations performed by external programs or libraries, limiting the scope of the audit.

4.1.3 Library-Wrapping PATs

Library-wrapper PATs intercept operations at the library level. Overhead depends on the specific implementation, but should generally remain manageable because the tool monitors only relevant operations. Nevertheless, some performance cost is unavoidable given the large number of operations that will still be observed. Transparency is preserved as calls pass through the normal library interfaces. The captured data can be very comprehensive and relevant if all operations of interest are performed by the library. The scope of

this approach will be as wide as the library used. While usually limited in scope, applying this to low-level libraries for system interactions can make this approach quite universal. Provided that programs are dynamically linked, nearly every relevant operation can be captured unless the program invokes system calls directly.

On Linux, which is the relevant OS for HPC, this approach is future-proof as user space never changes. Additionally, the captured data can be sufficient to reproduce computations provided every I/O operation goes through the hooked functions and can automatically extend to child processes as long as the library is used.

4.1.4 Kernel-Instrumentation-Based PATs

Kernel-level PATs are universally applicable, constrained only by kernel version compatibility. Because all operations ultimately pass through the kernel, this method can detect nearly every relevant operation. Kernel instrumentation typically incurs higher overhead, commonly 10–20% [7] possibly due to the need to filter system calls unrelated to the target workload and other additional complexities. However, a low-overhead implementation, LPS, exists and demonstrates that overhead can be reduced to approximately 1%. This approach is fully transparent, as all monitored operations must interact with the kernel, ensuring comprehensive provenance capture.

4.1.5 Summary of Method Comparison

Table 2 displays an overview of how well the listed methods can potentially perform in the criteria set out earlier:

Method	Scope	Time overhead	Transparent	Reproducibility
Compute platform	Limited	Low	Yes	Medium
API	Partial	Low	No	Medium
Library Wrapper	Broad	Medium	Yes	High
Kernel Instrumentation	Broad	High	Yes	High

Table 2: Scope, time overhead, transparency, and reproducibility of each method.

4.1.6 Auditing Methodology Selection Rationale

The PAT must be applicable across a broad spectrum of computational workloads while remaining transparent with respect to the host application. Among the available auditing strategies, only system library wrapping and kernel instrumentation operate at a sufficiently low abstraction level to meet these requirements. Both approaches, therefore, constitute viable foundations for a general-purpose auditing framework. The following paragraphs outline the considerations that led to the selection of library wrapping in this work.

Kernel instrumentation is largely agnostic to user-space programs and therefore highly versatile. Because all relevant operations ultimately pass through the kernel interface, this method enables comprehensive coverage independent of programming language or runtime environment. In addition, apart from tracing-based approaches such as strace, kernel-level monitoring does not directly interfere with application code. However, this generality entails increased implementation complexity and additional processing overhead. Although LPS demonstrates that low overhead can be achieved, many existing tools incur performance penalties that far exceed acceptable limits for HPC environments. Furthermore,

the kernel gets updated over time, and changes can introduce incompatibilities that require the PAT to be modified. This will create a permanent need for maintanance.

Wrapping low-level system libraries offers a comparatively contained and implementable alternative. Selected `libc` functions are overridden to incorporate logging while delegating execution to the original implementation. This approach does not capture programs that bypass `libc` and issue system calls directly, such as Go applications, nor does it apply to statically linked binaries. However, most scientific workloads are written in Python, C, or C++ and rely on dynamically linked C libraries, which limits the practical impact of this restriction. While function interposition can introduce instability if implemented carelessly, these risks can be mitigated through careful engineering, and the Linux user-space interface remains entirely stable, removing any dependence on long-term maintenance.

To the best of our knowledge, no provenance auditing tool focused on `libc` interposition has been published to date [7]. PROV-IO mentions this approach in its paper but provides no evaluation results for it, and the code available in the GitHub repository remains incomplete and does not compile. SPROV is the closest existing system, but it wraps only the set of I/O functions defined in `stdio.h` such as `fread` and `fwrite`, which are primarily used by C programs. Languages like Python do not route their I/O through `stdio.h` but instead call lower-level `libc` functions such as `read` and `write` directly, meaning SPROV would miss essentially all of their I/O activity. Even for C programs, `stdio.h` does not cover process creation via the exec family or file system operations such as `rename` and `unlink`, leaving significant gaps in provenance coverage.

This absence in the literature creates a clear motivation to investigate the feasibility and practical implications of employing `libc` interposition as the auditing mechanism. Accordingly, system library wrapping was selected as the auditing strategy in this thesis. At the same time, it is acknowledged that kernel instrumentation offers broader theoretical coverage and certain advantages, including minimal interference with host processes.

4.2 PAT Design

Beyond the injector itself, the PAT has other components that facilitate the auditing process, store the provenance data and artifacts, and provide the collected information to the end user. All components of the PAT were implemented in C/C++, both to remain uniform with the injector and because performance efficiency is a core goal of this thesis. The implementation presented in this thesis is intended as a proof of concept to evaluate the feasibility of the overall design and its core mechanisms. Its source code is available in a public repository [53]. Accordingly, some components remain incomplete or support only a subset of the functionality envisioned for a production-ready system. These limitations are discussed where relevant throughout this section.

This section lists the various components and outlines on a high level why each of these components exists and what role it plays, while subsequent sections go into detail on how each component is designed, the reasoning behind specific design decisions, and how it is implemented and used.

4.2.1 Components overview

The PAT consists of multiple components:

- **Injector**: the injector is overwriting `libc` functions and is attached to the target processes via `LD_PRELOAD` to include logging

- **Prov Tool**: the prov tool is responsible for attaching the injector to the target process, processing the resulting provenance data, backing up the involved files, and transmitting the resulting data to the backend for storage
- **Backend**: the backend accepts data from the prov tool which it stores in an SQL database. It can then provide it to the visualizer and querier components.
- **Querier**: The querier allows the user to query for previous jobs and execs. This includes job IDs, exec IDs, and the specific files involved. With that data, users can access the files that were used and produced in and by the job. The job ID can also be used for the visualizer.
- **Visualizer**: The visualizer creates SVG files providing an overview for a given job by visualizing the different execs, the processes for each exec, and the different ways each process accessed which files.
- **Retriever**: The retriever copies the artifacts from the beginning of a job or a specified tracked computation within a job back to their original locations on disk.

All components except the injector include functions that generate JSON strings for transmission to other components via HTTP using `curl`, as well as a `simdjson` [54] parser tailored to the expected JSON format. Each component loads the received data into appropriate structs for further processing. All inter-component communication follows this pattern, with JSON over HTTP as the uniform exchange format. The injector is the only exception, as it only produces JSON output and does not perform any parsing or network communication. Instead, it transfers its data to the prov tool by writing it to a directory in `/dev/shm`, from where the prov tool reads it after the process has exited.

4.2.2 Attaching the Injector

As described above, the auditing is performed by a `libc` library wrapper. This wrapper must somehow be attached to the host processes. There are multiple ways to accomplish this:

- **Direct `LD_PRELOAD` Injection**: The user sets `LD_PRELOAD` directly, either via the `sbatch` command:

  ```
  sbatch --export=ALL,LD_PRELOAD=/path/to/libinjector.so myjob.sh
  ```

  or by adding the following line near the top of the job script:

  ```
  export LD_PRELOAD=/path/to/libinjector.so
  ```

- **SLURM Prolog and Epilog**: System administrators configure the SLURM Prolog to attach the injector via `LD_PRELOAD` and the SLURM Epilog to process the collected provenance data after the job completes.
- **Prov Tool**: The user invokes a dedicated tool that attaches the injector, handles post-processing of the provenance data, and manages communication with the backend.

There are several drawbacks for our application when using direct `LD_PRELOAD` injection or SLURM Prolog and Epilog:

Direct `LD_PRELOAD` Injection
Asking the user to attach the injector manually via `LD_PRELOAD` has several drawbacks. First, it exposes low-level Linux internals to the user, which is not in line with the transparency requirement. Second, it creates an architectural problem: without a dedicated tool to handle post-processing, either a separate service would have to be running on the node at all times to process and transmit the provenance data, or this responsibility would fall on the injector library itself. The first is inefficient and creates unnecessary overhead and complexity. The latter is undesirable because it conflates concerns that are better kept separate, increases the complexity of the injector, and raises the risk of disrupting the host process. Keeping the injector as simple as possible is an explicit design goal. Third, tying the user interaction directly to `LD_PRELOAD` means that switching to a different auditing mechanism in the future, such as eBPF, would require changing how the user interacts with the system entirely. Abstracting this behind a dedicated tool means such changes can be made without the user noticing. Finally, direct injection provides no way for the user to supply additional parameters such as a custom JSON object or the folder whose provenance artifacts should be backed up, both of which are necessary for the broader functionality of the PAT.

SLURM Prolog and Epilog
Attaching the injector and processing the provenance data via the SLURM Prolog and Epilog would make the auditing completely invisible to the user. However, this approach is only appropriate for a PAT that is safe and compatible with every possible workload, as it would be applied to all jobs without exception. The `libc` wrapper is not meant to be used this way, as it is not guaranteed to be stable across all applications and is explicitly intended to be opt-in rather than applied system-wide. Furthermore, this method also provides no way for the user to supply additional parameters. It also requires root privileges to configure.

Prov Tool
A dedicated tool is the best approach. It is opt-in and abstracts the auditing process behind a clean and intuitive interface that is agnostic to the underlying auditing mechanism. It launches the computation with the injector attached and handles all post-processing. Future changes to the auditing mechanism do not affect how the user interacts with the system, and no root privileges in order to apply changes to the SLURM Prolog and Epilog configuration are required. It also allows the user to provide additional information through parameters. Lastly, the user can choose which specific lines within the bash script submitted to SLURM the auditing should be applied to, each of which is referred to throughout this thesis as an exec step.

4.2.3 Client-Server Architecture

The PAT is organized as a client-server system in which the prov tool and the DB interface act as clients that send requests to a central backend. The backend is solely responsible for storing provenance data in the SQL database and for serving this metadata back to the clients. It should be noted that the prov tool does maintain its own local SQLite [55] database for data lake operations, specifically for checksum lookups to avoid redundant file transfers, but this is a separate concern from the main provenance database.

Centralizing all provenance database interactions in the backend ensures that concurrency is never an issue for the main database, as only one component ever writes to or reads from it. We do have this issue in the prov tool, where the checksum database is accessed directly and concurrently by multiple prov tool instances across nodes, which could become an issue at scale. This is discussed in more detail later, in the dedicated prov tool section.

Keeping all provenance database interactions in a single component also provides a clean abstraction boundary between the components that collect and use provenance data and the component that stores it. This puts all provenance database logic in one place and makes the rest of the system agnostic to how or where data is stored. The interactions between clients and server are clearly defined through JSON over HTTP, which makes individual components easy to replace or extend without affecting the rest of the system and provides a clear picture of how data flows through the PAT.

4.2.4 Provenance Data Access and Retrieval

After the provenance data has been collected, it must be made accessible and useful to the user. In this regard, the PAT pursues two aims. The first is to provide a clear overview of a job, covering which processes were involved, how each process interacted with which files, which processes invoked which other processes, and how data was passed between exec steps. The second is to give the user access to the provenance artifacts that were present during the computation and to allow the state before any job or individual exec step to be reconstructed so that computations can be repeated. To accomplish this, three components are provided.

The querier serves as the primary interface for retrieving provenance metadata. It allows users to list their previous jobs and retrieve the job ID and cluster name that uniquely identify each one, both of which are required as input for the visualizer and retriever. Beyond this, it provides a fine-grained interface for inspecting the contents of a job in detail, listing the individual exec steps and the processes within each step, along with the files each process accessed and the operations it performed on them. This level of detail makes the querier a useful tool in its own right for understanding how a computation unfolded, and it also helps users identify the job for which they want to retrieve data.

The visualizer helps to get a quick overview of complex computations. By generating a graphical representation of the job, it allows users to quickly understand the structure of the computation, trace how data flowed between processes and exec steps, and identify dependencies. This kind of overview is quite valuable when revisiting a computation after some time, when trying to understand a job that was written by someone else, or when trying to understand why a computation fails.

The retriever provides reproducibility. Having recorded which files were present before and after each exec step, the PAT can restore those files to their original locations, allowing the user to rerun either the entire job from scratch or a specific exec step in isolation. This is made possible by the data lake, which stores the actual file contents rather than just their names, ensuring that the required artifacts remain available even if the originals have since been modified or deleted. The ability to restore intermediate states within a job is specifically useful in iterative scientific workflows, where a user may want to rerun only the later stages of a computation with modified parameters without repeating expensive earlier steps.

4.3 Injector

The injector overrides selected `libc` functions that are involved in modifying files or starting other processes. Each overridden function performs logging and then calls the original `libc` implementation. In addition, a constructor and destructor are defined to log process start and termination events. It records all observed operations regardless of which file they were performed on or where that file is located in the file system. This contrasts with the prov tool, which only backs up artifacts located within the folder specified by the user.

Because `LD_PRELOAD` propagates automatically to child processes, logging extends to every process spawned within the job without any additional configuration. Internally, all data collected by the hooks is appended to a queue in memory and later written in JSON format to `/dev/shm`, a directory backed by RAM rather than disk, where it can be accessed by the prov tool once the process has exited.

The injector is designed to be modular and independent of both the prov tool and SLURM, requiring only a single environment variable specifying the output location. The prov tool is similarly decoupled. Rather than assuming any particular auditing implementation, it expects only a well-defined JSON format at the file path it supplies to the auditing tool. This separation means the injector can be replaced by an alternative auditing mechanism in the future without requiring significant changes to the prov tool. The output path is currently communicated via an environment variable, which is a natural fit for the `LD_PRELOAD`-based injector since environment variables propagate automatically through the process tree. Should the auditing mechanism change in the future, for example to kernel instrumentation, the path could instead be passed as a command-line argument to the auditing tool, with no impact on the rest of the system.

4.3.1 Constructor

The injector records process metadata for each monitored process, specifically the timestamp of the start of the process, the PID, PPID, launch command, and environment variables. Rather than collecting this in the constructor, the metadata is captured lazily on the first recorded operation and will be added first in the output file as a separate JSON object. Performing this work during process initialization was found to cause crashes in some programs, and deferring the collection avoids this problem. Correctness is preserved regardless, since the process start record is inserted into the queue with a timestamp preceding that of the triggering operation, ensuring it always appears first in the output.

4.3.2 Hooks

The hooks record the relevant operations of interest: reads, writes, executes, unlinks, and renames. Additional operations related to I/O exist but are intentionally not tracked to prevent unnecessary complexity and verbosity. Depending on the use case, some of these omitted hooks could be included in alternative versions. The following `libc` functions are currently categorized as follows:

- **Reads**: `read`, `pread`, `pread64`, `readv`, `preadv`, `preadv2`, `recvfrom`, `recvmsg`, `recvmmsg`, `getdents`, `getdents64`, `fread`, `fgets`, `fgetc`, `getc`, `getchar`, `fscanf`, `scanf`, `sscanf`, `vsscanf`

- **Writes**: `write`, `fwrite`, `writev`, `pwrite`, `pwrite64`, `fputs`, `fprintf`, `vfprintf`, `dprintf`, `vdprintf`, `fputc`, `fputs_unlocked`, `fwrite_unlocked`, `pwritev`, `pwritev2`
- **Executes**: `execve`, `execveat`, `fexecve`, `execv`, `execvp`, `execvpe`, `execl`, `execlp`, `execle`
- **Unlinks**: `unlink`, `unlinkat`, `remove`, `rmdir`, `shm_unlink`, `mq_unlink`, `sem_unlink`
- **Renames**: `rename`, `renameat`, `renameat2`
- **Exits**: `_exit`, `_Exit`, `quick_exit`
- **Ignored**: `clone`, `open`, `open64`, `creat`, `openat`, `openat2`, `close`, `close_range`, `fclose`, `pipe`, `pipe2`, `dup`, `dup2`, `dup3`, `mmap`, `mmap64`, `munmap`, `msync`, `mprotect`, `madvise`, `mincore`, `ftruncate`, `truncate`, `posix_fadvise`, `posix_fallocate`, `link`, `linkat`, `symlink`, `symlinkat`, `access`, `chmod`, `chown`, `utime`

The hooks all look similar to this:

```
ssize_t read(int fd, void* buf, size_t count) {
    static auto real_read = (ssize_t (*)(int, void*, size_t)) nullptr;
    RESOLVE_REAL(real_read, "__libc_read", "read", (ssize_t)-1);
    ssize_t ret = real_read(fd, buf, count);
    SAVE_ERRNO;
    log_read_fd(fd);
    RESTORE_ERRNO;
    return ret;
}
```

Listing 1: Hooked `read` function in C++ for logging file descriptors

This example overwrites the `libc read` function. The original implementation is resolved once using `dlsym(RTLD_NEXT, "read")`, which searches for the next occurrence of the symbol in the dynamic linker chain, effectively retrieving the real `libc` function. The result is stored in a `static` local variable so that the resolution only occurs on the first call and is reused on all subsequent ones. The original `read` is then called, and its return value is saved. The error number is saved immediately afterwards to ensure that any failure within `log_read_fd` does not overwrite it. `log_read_fd` resolves the file path from the file descriptor and appends the provenance record to the in-memory queue, from where it will later be written to `/dev/shm` by the destructor. The error number is then restored and the return value returned, leaving the observable behavior of `read` unchanged. This pattern is applied to all other hooks with only minor variations.

Special consideration is needed when handling execution-related functions. The functions `fork` and `vfork` are deliberately not hooked, as tests have shown that these are particularly sensitive and can potentially crash the program. Execution relationships are reconstructed later instead using PID, PPID, and timestamp information, which is used to build an execution tree. PIDs alone are not sufficient because Linux could potentially reuse them, and more importantly, the PAT treats processes resulting from `execv` as separate, even while the PID remains unchanged to provide a better overview. Since this mechanism already exists and works for forks, `posix_spawn`, `posix_spawnp`, and `system` are not hooked either. Doing so would unnecessarily increase complexity.

In contrast, all `exec`-family functions are hooked. Upon a successful exec call, the process image is replaced, and the destructor of the previous instance is not executed, bypassing the recording of the provenance data in memory since this is done by the destructor, which is not executed then. To mitigate this, the functionality normally executed in the destructor is invoked manually before calling the original `libc` exec function. The exec event is logged explicitly. Because the timestamp of this log entry precedes the start timestamp of the new process image, the reconstruction logic used for fork-based process trees cannot be applied when an exec occurs. Failed exec attempts are also logged, allowing the processor to identify which exec calls not to record. After the failed attempt has been recorded, the process continues without being replaced.

Each hook appends a JSON object to the output vector. The recorded fields depend on the operation type, where each hook is grouped into either READ, WRITE, EXECUTE, UNLINK, or RENAME. Reads, writes, and unlinks store the affected path. Rename operations store both the original and new paths. Every JSON object includes the operation type and a timestamp. Timestamps are required to preserve ordering across processes and nodes, as renames and execs could otherwise create ambiguity regarding which files and processes they are attributed to. Since all records for a given process are written to the same file and the process start record appears first, subsequent operations are implicitly associated with the process metadata it contains without needing to repeat the PID, PPID, launch command, or environment variables in every record. An example JSON structure is shown below:

```
{
  "event_header": {
    "operation": "PROCESS_START",
    "ts": 1700000000000000000
  },
  "event_data": {
    "pid": 12345,
    "ppid": 1234,
    "launch_command": "/usr/bin/cat input.txt",
    "env_variables": [
      {
        "HOME": "/home/user"
      },
      {
        "PATH": "/usr/bin:/bin"
      }
    ]
  }
},
{
  "event_header": {
    "operation": "READ",
    "ts": 1700000000000100000
  },
  "event_data": {
    "path": "/home/user/input.txt"
  }
},
{
  "event_header": {
    "operation": "WRITE",
    "ts": 1700000000000200000
  },
  "event_data": {
    "path": "/home/user/output.tmp"
  }
},
{
  "event_header": {
    "operation": "RENAME",
    "ts": 1700000000000300000
  },
  "event_data": {
    "original_path": "/home/user/output.tmp",
    "new_path": "/home/user/output.txt"
  }
},
{
  "event_header": {
    "operation": "PROCESS_END",
    "ts": 1700000000000400000
  },
  "event_data": {}
}
```


Listing 2: Example provenance JSON structure

The exit functions are explicitly hooked because, without this mechanism, the destructor code would be bypassed, resulting in the complete loss of process provenance data. However, these exit functions are designed to terminate execution immediately, and executing additional code would constitute a violation of their intended specification. While this approach may introduce potential undefined behavior and violates their explicit specification, the alternative, flushing all recorded provenance data at each step, would impose a substantial performance overhead.

4.3.3 Destructor

The destructor logs process termination and writes all JSON entries accumulated in the in-memory queue to a JSONL [56] file, where each line represents one independent event. Compared to standard JSON, this format reduces syntactic overhead and allows `simdjson` to parse each line as a self-contained object without loading the entire file at once. The output directory is provided by the prov tool via the `PROV_PATH_WRITE` environment variable, which propagates automatically through the process tree. Each output file is named using a random identifier combined with the PID of the process. The random identifier is re-generated after each exec call so that the new process image, which retains the same PID, writes to a distinct file rather than colliding with output from the previous image. The data is written to `/dev/shm`, a directory whose contents reside in RAM rather than on disk, which reduces write latency and avoids unnecessary storage I/O.

4.4 Prov Tool

The prov tool controls the auditing process. It attaches the injector to the target processes via `LD_PRELOAD`, processes the collected provenance data, and transmits the result to the backend. The prov tool can be integrated into SLURM job scripts as follows:

```
#!/usr/bin/env bash

prov start --json '{"example": "json"}'
prov exec "./first_folder/first_exec" --path "./first_folder"
prov exec "./second_folder/second_exec" --path "./second_folder"
prov end
```

Listing 3: Example usage of the PAT CLI with bash commands

At the beginning of a job, the user invokes `prov start` and can provide a custom JSON object, which can later be retrieved. Individual job steps are then tracked using `prov exec`, which requires the command corresponding to the job step and a path to the root directory of that step. This path enables optional filtering of files in the interface and determines which files are backed up to the data lake. The job is then ended with `prov end`. Other commands belonging to the computation, such as module loading or additional steps, can still be used normally without any influence from the PAT.

4.4.1 Start

When `prov start` is invoked, directories are created for the injector to store JSONL files. The backend is then notified that a new job has started. Information about the job name and user is transmitted to the backend together with a header that uniquely identifies the job. The header contains the type, the job ID, and the cluster name. Since a data center may contain multiple clusters, the job ID alone is insufficient for identification. The job ID, job name, and cluster name are made available to the provenance tool through environment variables populated by SLURM at job launch time. During startup, `prov` reads these variables from its process environment and uses them to populate the corresponding fields in the request sent to the backend. This avoids the need for manual user input and ensures that the job metadata recorded by the provenance system matches the scheduler-assigned identifiers of the running SLURM job. The JSON sent has the following shape:

```
{
  "header": {
    "type": "start",
    "job_id": 1,
    "cluster_name": "cname1",
    "timestamp": 1772684001018742000
  },
  "payload": {
    "job_name": "test_name",
    "username": "example_user",
    "json": "{}",
  }
}
```

Listing 4: Example JSON message sent by the prov tool at job start

4.4.2 Exec

The `prov exec` command is responsible for backing up artifacts, running the computation with the injector attached, processing the resulting provenance data, and transmitting it to the backend. The following pseudocode gives an overview of this process:

```
func exec(cmd, custom_json, work_dir) {
    unbackupped_paths = get_unbackupped_paths(work_dir);
    path_checksums = update_data_lake(unbackupped_paths);

    json_out_path = run_with_injector(cmd); // blocks until exec completes

    prov_data = process_provenance_data(json_out_path, path_checksums, work_dir);

    path_checksums = update_data_lake(prov_data.written_files); // excludes deleted files
    for (path, checksum) in path_checksums {
        prov_data[path].checksum_after = checksum;
    }
    send(prov_data, to=BACKEND_ENDPOINT);
}
```

Listing 5: High-level structure of the `prov exec` command

Each step is described in detail below.

`get_unbackupped_paths` and `update_data_lake` (pre-exec)
For each file in the provided folder, a checksum is computed and looked up in a dedicated SQLite database that records the checksums of all previously backed up files. SQLite automatically indexes primary keys, making these lookups efficient. If the checksum is already present, the file has not changed since a prior backup, and no transfer is needed. If it is not present, the checksum is inserted into the database, and the file is copied into the data lake under a folder named after the first two characters of its checksum, with the filename set to the full checksum. This organization avoids redundant storage of identical files across the same and different jobs. The original filename is preserved in the provenance database and can be recovered from there when needed. However, the current implementation does not back up executables to the data lake. Although the launch command of each process is recorded, the executable itself is not yet tracked as a provenance artifact.

Another limitation of this design is that in a multi-node job, multiple prov tool instances running concurrently on different nodes will all attempt to read from and write to the same SQLite database simultaneously. SQLite is not designed for high-concurrency

write access, and this could lead to lock contention or corrupted state at scale. One resolution would be to route all checksum lookups and insertions through the existing backend, which would then serialize these database interactions. Alternatively, SQLite could be replaced with a database engine designed to handle concurrent writes, such as PostgreSQL [57].

Routing this through the existing backend would resolve the concurrency issue cleanly and has the additional benefit of consolidating all database and data lake interactions within a single component. The overhead introduced would be minimal, as only the folder path needs to be transmitted to the backend, which then performs all the work currently done by the prov tool locally and returns a short response once it has finished. For these reasons, this improvement is targeted for a future version.

`run_with_injector`
The command provided to the prov tool is executed with the injector attached via `LD_PRELOAD`. This call blocks until the exec step has completed.

`process_provenance_data`
After the execution has completed, the prov tool parses the JSONL files written by the injector to the output directory it provided and loads the recorded events into structs. Then, the processing begins, which is done by file. The structure is laid out in the following listing:

```
func process_provenance_data(events_by_file, path_checksums) {
    process_map = init_processing(events_by_file);

    events = combine_and_sort_by_timestamp(events_by_file);

    execute_set_map = resolve_forks(process_map);

    rename_map, seen_files, enqueued_execs = {}, {}, {}
    for event in events:
        switch event.operation:
            ProcessStart -> register computation_id if not yet present
            Rename       -> update rename_map, move existing operations to new path
            Read/Write/Unlink -> resolve path, record operation, and start checksum
            Transfer     -> treat as Read + Write
            Exec         -> enqueued_execs[computation_id].push(event)
            ExecFail     -> enqueued_execs[computation_id].pop()

    execute_set_map = resolve_execs(enqueued_execs, process_map, execute_set_map)

    return ProcessedExecData { process_map, execute_set_map, rename_map }
}
```

Listing 6: High-level structure of `process_provenance_data`

The processor begins by iterating over all event files generated by the injector, one file per monitored process instance. From the ProcessStart event in each file, it extracts the PID, PPID, launch command, and environment variables. A computation ID is derived by combining the launch command with a hash of the environment variables. This identifier is distinct from the Linux PID in that it represents the logical computation rather than the concrete process instance, which matters both for handling exec calls where the PID is retained across an image replacement and for potential future support of distributed execution where the same logical computation appears on multiple nodes with different local PIDs. The end time of each process instance is taken from the timestamp of its

final ProcessEnd or Exec event. All events in the file are then annotated with the derived computation ID, and Exec events are additionally annotated with the PID of the process in which they occurred, since this information is needed later but is not stored in the raw injector output.

All events from all files are then merged into a single list and sorted by timestamp. A global ordering is necessary because renames must be processed in the order they occurred and because the Exec and ExecFail pairing depends on processing events sequentially.

Fork-derived parent-child relationships are reconstructed from PID, PPID, and process lifetime information. For each process instance, the processor searches for a monitored process whose PID matches the PPID and whose lifetime fully contains that of the child. When such a parent is found, the child computation ID is inserted into the execute set of the parent computation ID. The result is a map where each computation ID points to the set of computation IDs it directly spawned, capturing the full execution hierarchy across the job.

The sorted event stream is then processed in sequence. ProcessStart events register a new entry for the computation ID if one does not yet exist, storing the launch command, environment variable hash, and an empty file operation map. The full environment string in JSON format is also stored in a separate map keyed by the hash. Rename events update a rename map that links each current path back to its earliest known path, collapsing chains so that intermediate names are never stored, and operations already recorded under the old path are moved to the new filename.

Read, Write, and Unlink events resolve the affected path through the rename map and record the operation in the file operation map of the corresponding computation. On the first encounter of each original file identity, the pre-execution checksum is retrieved from the checksum map computed before the exec and stored alongside the operation. Whether a file has been encountered before is tracked using a set of seen filenames, and the rename map is consulted to ensure that a file that previously appeared under a different name is correctly recognized as already seen rather than treated as a new file. Transfer events are treated as both a Read and a Write, with the read side logged first and the write side logged immediately after. Exec events are pushed onto a stack stored in a map keyed by computation ID, and ExecFail events pop the most recent entry from the stack of the corresponding computation ID, so that only successful exec attempts remain on their respective stacks once all events have been processed.

The remaining Exec events are then used to reconstruct exec-derived relationships. When a process calls exec, its image is replaced by an entirely different program or a different invocation of the same program with different arguments, all while retaining the same PID. From a provenance perspective, this should be treated as an entirely different process even though Linux associates the same PID with it, which is what the computation ID mechanism achieves. Because the PID is retained, the new process image will appear in the process map under the same PID as the original, which is why Exec events were annotated with their PID during initialization even though operations are organized and interpreted by computation ID. The processor searches for the earliest process instance that shares the PID of the exec event and started after the timestamp of the exec attempt. Since there is exactly one new process image per successful exec call, only this earliest match is taken. This associates the exec correctly since a surviving Exec event is, by definition, one that succeeded, and the new process image will have been started immediately after the previous one was marked as ended by the injector. The reason it was marked as ended is because a successful exec triggers the same end recording

as the destructor, as described in the injector section. The identified computation ID is inserted into the execute set map, extending it with exec-derived relationships alongside the fork-derived ones.

It is worth noting that if the tool is extended to support multi-node auditing, the PID alone will no longer suffice to identify exec relationships, as processes on different nodes may coincidentally share the same PID while running at the same time. The machine ID of each node, available on Linux via `/etc/machine-id`, would need to be incorporated alongside the PID, first in the ProcessStart metadata recorded by the injector and then in the processing logic. This is simple to implement and is left for a future version supporting multi-node auditing.

The result is a `ProcessedExecData` struct containing four fields. The `process_map` is a map from computation ID to a `Process` struct, which stores the launch command, the environment variable hash, and an `operation_map` that records for each file path whether it was read, written, or deleted, along with the pre-execution checksum of that file if one was available before the exec step began. Importantly, all operations are keyed by the most recent path the file held at the time processing completed, that is, the path after every rename observed during the exec step has been applied so that the operation map always reflects the last known name of the file rather than any earlier name it may have carried. The `execute_set_map` maps each computation ID to the set of computation IDs it directly spawned, capturing the full execution hierarchy reconstructed from both fork and exec relationships. The `rename_map` and `env_variables_hash_to_variables` mappings carry forward the rename chains and environment string lookups built up during processing.

`update_data_lake` and checksum association (post-exec)

Once processing is complete, the prov tool iterates over all recorded operations and identifies files that were written during the exec step and not subsequently deleted. For each such file, the current checksum is computed from the file at its post-rename path and stored as the end checksum in the corresponding operation struct. The file is copied into the data lake using the same mechanism as the pre-exec backup, skipping files whose checksum is already present. This ensures that the data lake contains not only the state of the folder before the exec step but also the state of every file that was produced or modified during it, making it possible to restore the output of any individual exec step without having to rerun it.

`send`

The processed provenance data is serialized to JSON and transmitted to the backend via HTTP. The JSON payload contains the process map, the execute set map, the environment variable hash-to-string mappings, the rename map, the path provided by the user, and the command that was executed. The backend stores this data in the SQL database, associating it with the current job identified by the job ID and cluster name provided in the header. The header provided in the start step is provided here as well. The JSON has the following shape:

```
{
  "header": {
    "type": "exec",
    "job_id": 1,
    "cluster_name": "cname1",
    "timestamp": 1772684001036213000
  },
  "payload": {
    "processes": [
      {
        "process_command": "./script_child1 param",
        "process_id": "./script_child1 param_1",
        "env_variable_hash": 1234567890,
        "operations": {
          "/tmp/file2.txt": [
            "write"
          ]
        }
      },
      {
        "process_command": "./script_main",
        "process_id": "./script_main_1",
        "env_variable_hash": 1234567890,
        "operations": {
          "/tmp/file2.txt": [
            "read",
            "deleted"
          ]
        }
      }
    ],
    "execute_map": [
      {
        "parent_process_id": "./script_main_1",
        "child_process_id_array": [
          "./script_child1 param_1"
        ]
      }
    ],
    "env_variable_hash_pair_array": [
      {
        "env_variables_hash": 1234567890,
        "env_variables_array": [
          {
            "HOME": "/home/example_user"
          },
          {
            "PATH": "/usr/bin"
          }
        ]
      }
    ],
    "path": "/home/example_user",
    "command": "./script_main"
  }
}
```

Listing 7: Example JSON message generated by the prov tool for an exec event

4.4.3 End

When `prov end` is called, the backend is notified that the computation has concluded. The JSON has the following shape:

```
{
  "header": {
    "type": "end",
    "job_id": 1,
    "cluster_name": "cname1",
    "timestamp": 1772684001045776000
  },
  "payload": {
  }
}
```

Listing 8: Example JSON message sent by the prov tool at job completion

4.4.4 Limitations and Future Extensions

The current implementation does not support usage across multiple nodes yet. In its present form, each node launches an independent instance of the prov tool. A potential future extension could leverage the MPI [10] rank, a unique integer identifier assigned to each participating process in an MPI job, to coordinate processing, where only the prov tool instance on rank 0 would perform the actual processing, while instances on all other ranks would simply initialize the process with the injector preloaded. To ensure that processing occurs only after every node has completed its portion of a given exec step, MPI barriers, which force all participating ranks to wait until every rank has reached the same point in the program, could be inserted at the end of each step, forcing the rank 0 instance to wait until all other nodes have finished before initiating processing. All nodes could then save their data to the shared filesystem rather than the node-specific `/dev/shm` path so that the rank 0 instance can access it. Additionally, as noted in the processing section, the machine ID of each node would need to be incorporated alongside the PID in both the injector output and the processing logic to correctly identify exec relationships.

The prov tool is intended for use within SLURM job scripts, but its design allows compatibility with other job schedulers with minimal adjustments. SLURM provides relevant environment variables such as the job ID and cluster name, which the prov tool uses to identify the job the computation belongs to. Other schedulers typically expose comparable information through environment variables as well. Adapting the prov tool, therefore, generally requires only modifying which environment variables are read.

4.5 Backend

The backend stores the provenance data received from the prov tool and serves requests issued by the visualizer, querier, and retriever. Each client type communicates with a dedicated endpoint. Incoming write requests are processed sequentially from a queue to ensure that all database interactions are serialized, avoiding concurrency issues. It uses an SQLite database. The structure of the database is shown in Figure 1.

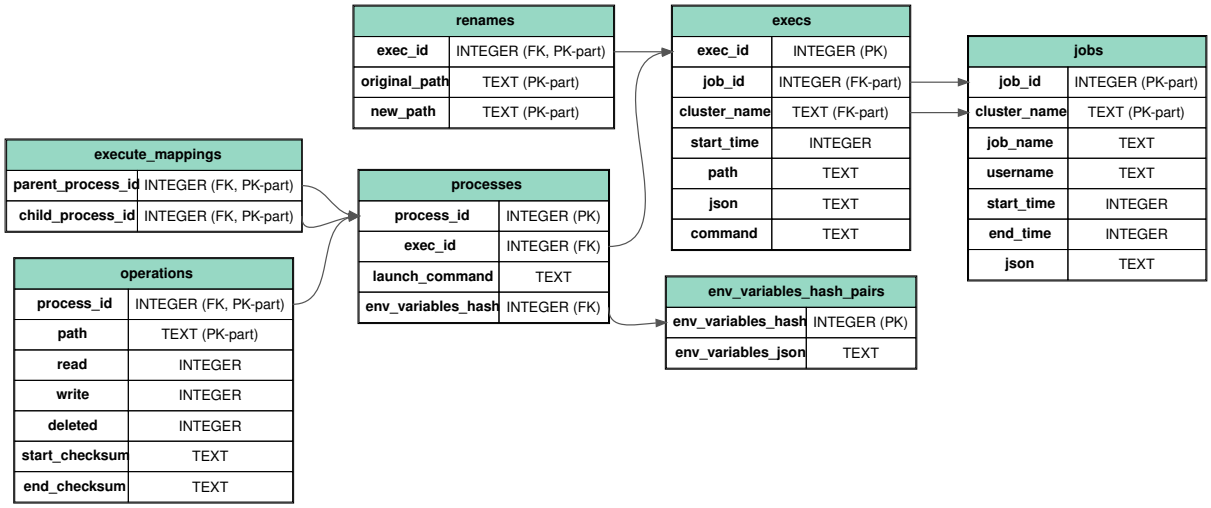

Figure 1: The SQL database

jobs

Each SLURM job is represented by one entry in the `jobs` table. The combination of job ID and cluster name uniquely identifies a job. Additional attributes stored in this table include the job name, username, start time, end time, and a custom JSON object optionally provided by the user.

execs

Each job consists of at least one step, corresponding to a line in the SLURM script. If a step is tracked via `prov exec`, an entry is created in the `execs` table. Every exec has a synthetic, auto-incrementing primary key. The job ID and cluster name are stored as foreign keys referencing the associated `jobs` entry. The start time of the exec is recorded. An explicit end time is not required, as it can be derived from the start time of the subsequent exec or, in the case of the final step, from the job's end time. The command used to invoke the step, as well as the path and optional JSON object provided by the user, are also stored.

renames

Each exec may be associated with multiple entries in the `renames` table. Every rename entry references its corresponding exec via a foreign key. The original path and the new path are stored. Together with the exec ID, these fields form the composite primary key.

environment variables

Environment variable sets are often large and frequently identical across processes, execs, or even entire jobs. To avoid redundancy, each unique combination is stored only once. The full set of environment variables is encoded as JSON and stored alongside a hash value, which serves as the primary key.

processes

Each exec is associated with at least one process and typically several. The `processes` table models Linux processes with one modification: when `execve` is used, Linux retains

the same PID, whereas the PAT treats the resulting program image as a separate process, as it represents a distinct execution context. Each process entry has a synthetic, auto-incrementing primary key and stores the corresponding exec ID as a foreign key. The launch command, consisting of the executable and its `argv` arguments, is recorded. In addition, the environment variable hash is stored as a foreign key referencing the `environment variables` table.

operations
All file-related operations are stored in the `operations` table. The composite primary key consists of the process ID (which is also a foreign key to `processes`) and the file path on which the operation was performed. Three integer fields represent read, write, and delete operations. A value of 0 indicates that the operation did not occur, while 1 indicates that it did. Since multiple operation types may apply to the same file within a single process, this representation provides a compact encoding. SQLite does not provide a native boolean type.

Two optional text fields store the checksum of the file before and after the exec step, referred to as the start and end checksum respectively. Both fields are nullable. They are null if the file was outside the folder specified for their respective exec step. Otherwise, the start checksum is null if the file did not exist before the exec step began. The end checksum is null if the file was deleted during the step, regardless of whether it existed beforehand or if the file existed before the step but was not modified during it. Consequently, both checksums are null if the file was created and deleted within the same step.

execute mappings
Parent–child relationships between processes are stored in the `execute mappings` table. The parent process ID and child process ID together form the composite primary key. Both fields also serve as foreign keys referencing the `processes` table.

4.6 Querier

The querier is invoked with `./db_interface query` and provides access to provenance metadata stored in the backend. Based on the input provided, it constructs and sends a JSON request to the backend, receives the response, parses it, and prints the result in a human-readable format. The full JSON request and response structures for all subcommands are provided in Appendix A.1. Three subcommands are available.

jobs
The `jobs` subcommand requires no arguments and prints the jobs belonging to the current user. To inspect jobs belonging to a different user, the `-u`/`-user` flag followed by a username can be provided. Access control has not been implemented yet. Filtering by date using `-b`/`-before`, `-a`/`-after`, or `-r`/`-range` would be a useful addition but has not been fully implemented. The output of the following command is shown below:

```
./db_interface query jobs
```

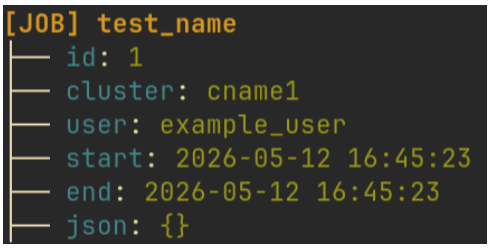

Figure 2: Output of a jobs query

execs

The `execs` subcommand requires a job ID and cluster name and lists all exec steps belonging to that job. Passing `-f`/`-files` additionally lists the processes associated with each exec alongside the files they accessed and the operations performed on them. The output of the following command is shown below. Only the second step is shown for brevity:

```
./db_interface query execs 1 cname1 -f
```

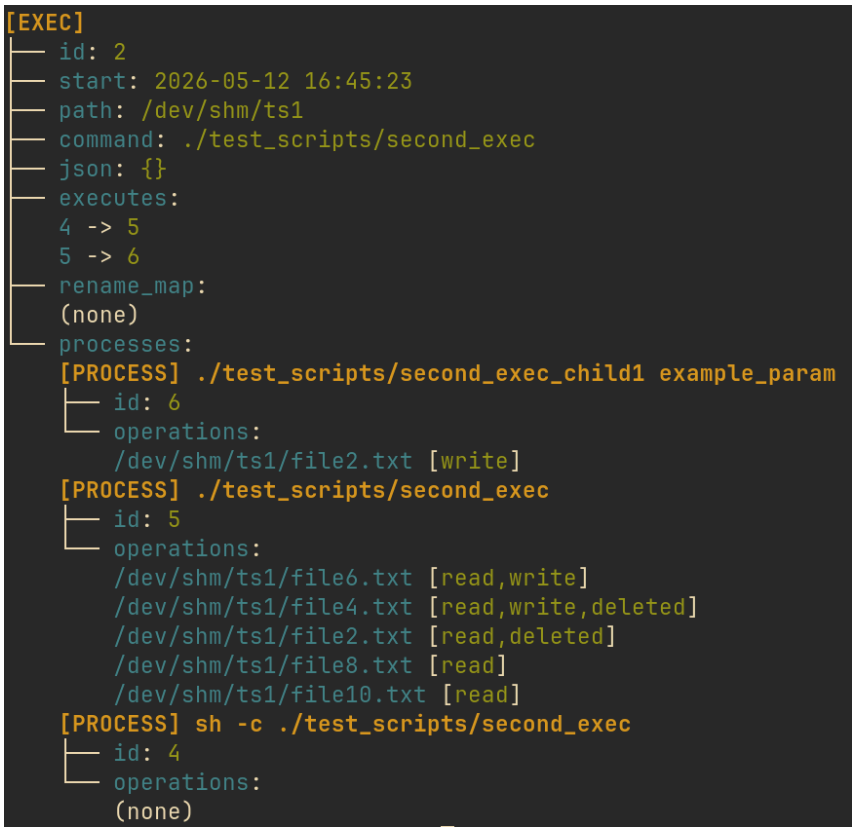

Figure 3: Output of an execs query with `-f` (second step only)

processes

The `processes` subcommand requires an exec ID and lists the processes belonging to that exec. The `-f`/`-files` flag is also available here and lists the files each process accessed along with the operations performed on them. The output of the following command is shown below:

```
./db_interface query processes 2 -f
```

```
[PROCESS] ./test_scripts/second_exec_child1 example_param
├─ id: 6
└─ operations:
   /dev/shm/ts1/file2.txt [write]
[PROCESS] ./test_scripts/second_exec
├─ id: 5
└─ operations:
   /dev/shm/ts1/file6.txt [read,write]
   /dev/shm/ts1/file4.txt [read,write,deleted]
   /dev/shm/ts1/file2.txt [read,deleted]
   /dev/shm/ts1/file8.txt [read]
   /dev/shm/ts1/file10.txt [read]
[PROCESS] sh -c ./test_scripts/second_exec
├─ id: 4
└─ operations:
   (none)
```

Figure 4: Output of a process query with `-f`

All three subcommands also support `-j`/`-json` to print the raw output as JSON rather than the formatted display and `-h`/`-help` for usage information. Together these commands allow provenance data to be inspected directly in the terminal. The `jobs` subcommand provides the job ID and cluster name needed to retrieve artifacts or generate a visualization for a specific job, which makes it especially useful.

4.7 Retriever

The retriever is started with `./db_interface retrieval` and allows users to restore file artifacts from previous jobs back to their original locations on disk. It queries the backend via JSON over HTTP to obtain the checksums and file paths required for restoration, then retrieves the corresponding artifacts directly from the data lake and copies them to their original locations on disk. In a future release, artifact retrieval could be moved into the backend itself to help centralize all database and data lake access within a single component. The full JSON request and response structures are provided in Appendix A.2. Two subcommands are available.

4.7.1 job

The retrieval command can be executed as follows to retrieve artifacts of a job:

```
./db_interface retrieval job 1 cname1
```

The `job` subcommand requires a job ID and cluster name and restores the complete file state that existed before the tracked computation began, aside from the executables, allowing the job to be rerun from scratch. Thus, `job` retrieval always reconstructs the initial state before any tracked exec step has been executed.

To determine this state, the retriever walks forward through all exec steps in chronological order and considers each file path only at its first occurrence within the job. If that first occurrence contains a non-empty start checksum, the file is treated as having

already existed before the job began, and the corresponding artifact is restored. If the first occurrence has no start checksum, the file is treated as having been created during the job and is therefore excluded from `job` retrieval. This ensures that all external input files and other pre-existing dependencies are restored, even if they are first accessed only in a later exec step, while files produced by the job itself are not restored.

It should be noted that rename resolution is not currently applied during retrieval. File paths are used as recorded in the database, meaning that a file renamed during a prior exec step will be restored to its post-rename path rather than its original name. Resolving this correctly would require applying the rename map of each exec step in reverse and represents a known limitation of the current implementation.

4.7.2 exec

The retrieval command can be executed as follows to retrieve artifacts of a exec step:

```
./db_interface retrieval exec 1
```

The `exec` subcommand requires an exec ID and restores the cumulative file state after that particular exec step has completed. Thus, providing exec ID x restores the state immediately after exec x has finished running. In contrast, the `job` subcommand restores the state before the first tracked exec of the job.

To reconstruct the requested state, the retriever first resolves the provided exec ID to the job it belongs to and retrieves the full ordered sequence of exec steps for that job. It then initializes the reconstructed state with the same external dependency set used by the `job` subcommand. Concretely, it scans the complete job in chronological order and, for each file path, considers only the first occurrence of that path within the job. If this first occurrence has a non-empty start checksum, the file is treated as having already existed before the computation began and is inserted into the initial baseline state. This applies even when the file is first observed only in a later exec step. As a result, `exec` retrieval includes not only files already accessed up to exec x, but also external dependencies that will only be needed by later exec steps, ensuring that the remaining part of the workflow can still run from that point onward.

Starting from this baseline, the retriever walks forward through all exec steps in chronological order up to and including the requested exec ID and incrementally updates the reconstructed state for every file path encountered. If an operation record contains a non-empty end checksum, that checksum becomes the current version of the file in the reconstructed state, as it represents the file after that exec step completed. This covers both files newly created by the computation and files modified by it. If a file is marked as deleted, it is removed from the reconstructed state. If a file is only read and therefore has no end checksum, its already known state remains unchanged.

In this way, `exec` retrieval reconstructs the complete cumulative state of the job after the selected exec step. This includes three classes of files: files that existed before the computation and are still present, files produced by exec steps up to and including exec x that have not subsequently been deleted, and external dependency files that are only first observed in later exec steps but are inferred to have existed before the job because their first occurrence carries a start checksum. Importantly, files produced by earlier execs are restored regardless of whether they are accessed again later, since the goal is to reproduce the full state after the selected exec rather than only the subset of files immediately consumed by the next step.

4.8 Visualizer

The visualizer is currently invoked as a standalone executable. In a future release it will be integrated into the `db_interface` program as a subcommand alongside the querier and retriever. It takes the job ID and cluster name as input parameters, which uniquely identify the job as described earlier. It retrieves the corresponding provenance data from the backend and generates an SVG file in the current working directory named after the job ID, cluster name, and the current timestamp. The SVG format was chosen deliberately, as it supports keyword search, which is particularly valuable given that a single graph may contain a large number of file entries. The full JSON request and response exchanged with the backend are provided in Appendix A.3.

The visualization separates the job into its individual exec steps. Within each step, accessed files are grouped by process, and the associated operations are listed. Table types and operation categories are color-coded using the colorblind-friendly Okabe–Ito palette [58]. If multiple operation types were performed on the same file, the coloring of the corresponding cell is split vertically in order to fit multiple colors into the same cell.

For each exec step, executions are displayed in separate tables. The header row identifies the process initiating the execution, followed by rows listing the processes spawned by it.

Additional tables indicate cases where the same file was accessed by multiple processes. On the right-hand side, further tables show when files were used across different exec steps, allowing users to trace file usage throughout the job. Aside from their scope, which is either process-level or exec-level, these tables follow the same structure and color-coding.

Tables have been employed in place of arrows because each execution step may encompass thousands of files. Attempting to trace arrows within each execution step, as well as between successive steps, would be next to impossible and incredibly tedious. In contrast, tables consistently maintain clarity even as the number of files increases. The output of the following command is shown below:

```
./visualizer 1 cname1
```

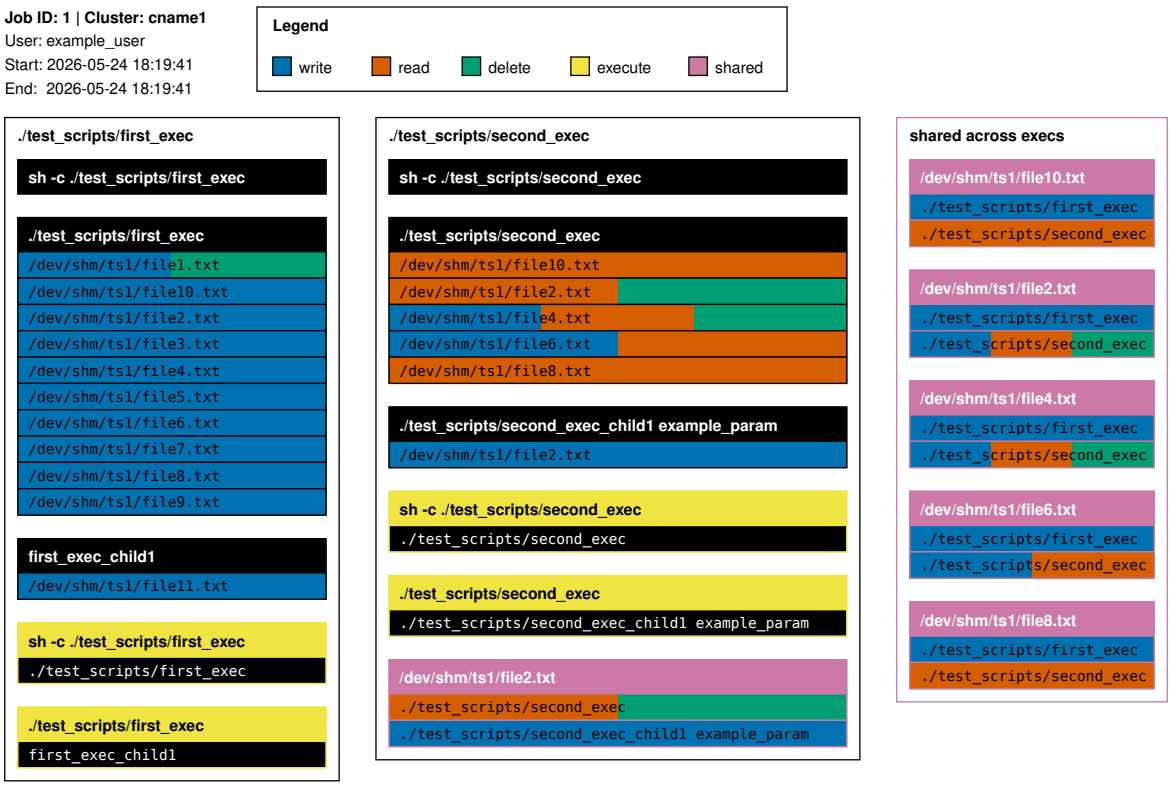

Figure 5: A basic toy example involving two exec steps

4.9 Configuration

A `config.json` file is provided to configure the PAT without requiring recompilation. It has the following shape:

```
{
    "injector_path": "/home/example_user/libcprov/build/injector/libinjector.so",
    "prov_artifacts_path": "/dev/shm/artifacts",
    "post_request_ip": "127.0.0.1",
    "post_request_port": 9000
}
```

Listing 9: PAT configuration file

The `injector_path` field specifies the location of the injector shared library, allowing the prov tool to locate it without hardcoding the path into the binary. The `prov_artifacts_path` field specifies the root directory of the data lake where provenance artifacts are stored, allowing system administrators to configure the storage location. The `post_request_ip` and `post_request_port` fields specify the address of the backend server. The configuration is parsed at startup and loaded into a struct that is shared across all components.

4.10 Component interactions

Figure 6 illustrates the interactions between the previously described components. The individual steps are enumerated in the order in which they are executed. Steps without

a leading letter correspond to the provenance auditing process itself. Steps prefixed with A describe the process of querying the metadata of a job. Steps prefixed with B relate to retrieving provenance artifacts associated with a job. Finally, steps prefixed with C represent the operations involved in generating the graph used to visualize the provenance data of the computation:

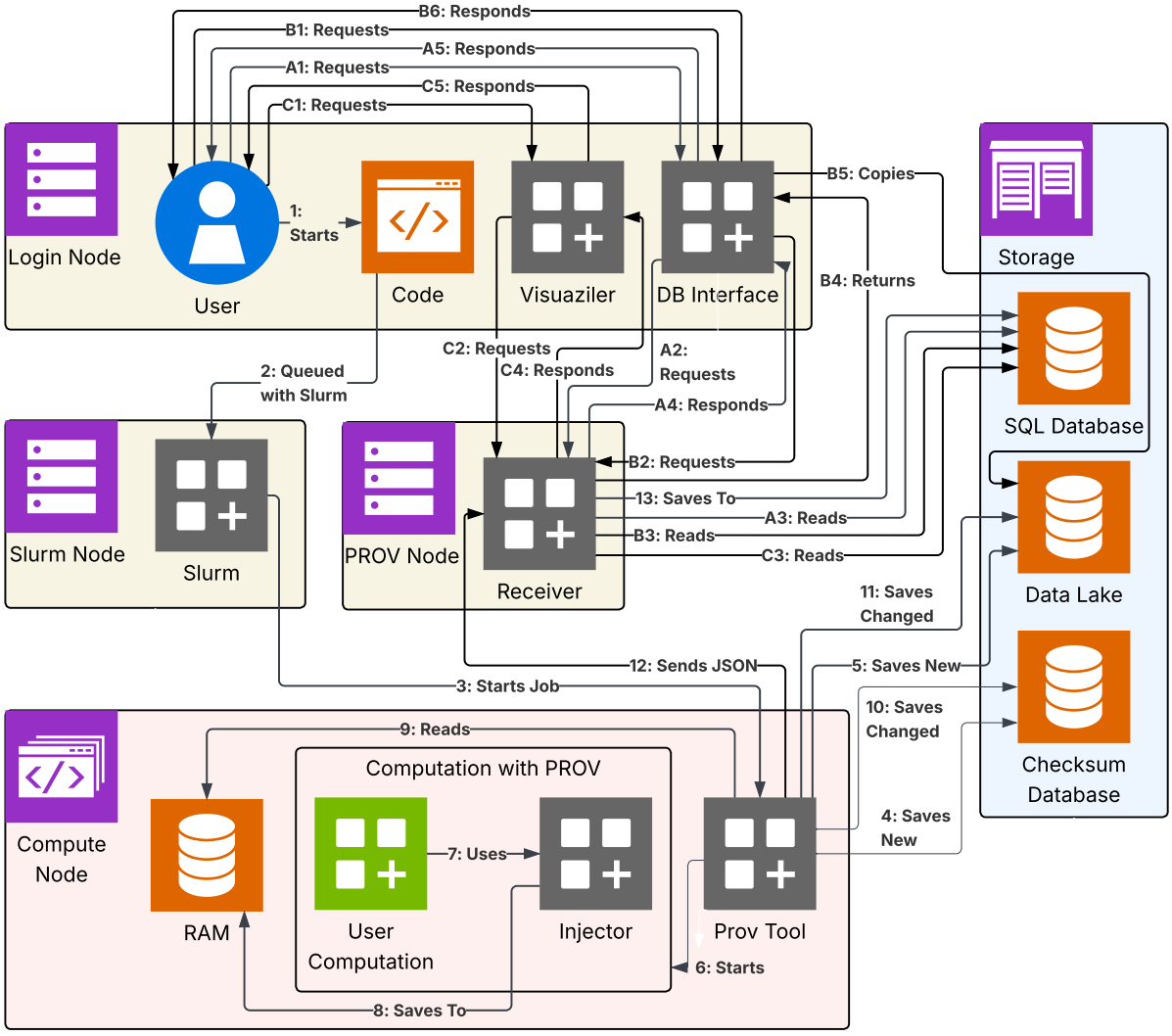

Figure 6: The Interactions of the Components

Steps 4–11 are repeated for each `exec` step. The `start` and `end` steps do not initiate computations and are therefore not involved in backing up any artifacts. Consequently, the prov tool skips steps 4–11 and sends the JSON data directly to the backend in step 12.

The table below provides a brief explanation of each step, where steps 4–11 refer specifically to the `exec` variant only. A more detailed breakdown is provided above in the sections describing each component, respectively.

Step	Description
1	The user submits their code using `sbatch`.
2	The computation request is queued in SLURM.
3	SLURM schedules and launches the job on a compute node.
4	New checksums of the files in the specified path are stored in the checksum database.
5	New files in the specified path are stored in the datalake.
6	The provenance tool is initialized to capture execution metadata.
7	The user's computation interacts with the injector library.
8	Provenance data is written to RAM in `/dev/shm`.
9	The prov tool reads the provenance data generated during computation.
10	New checksums of modified files are stored in the checksum database.
11	New files or modified files are stored in the datalake.
12	The prov tool sends the processed data, encoded in JSON, to the backend via HTTP.
13	Final provenance records are stored in SQL databases.
A1	The user requests provenance metadata from previous jobs via the querier.
A2	The querier requests the provenance metadata from the backend.
A3	The backend retrieves the metadata from the SQL database.
A4	The backend returns the metadata to the querier.
A5	The querier displays the retrieved metadata on the user's terminal.
B1	The user requests provenance artifacts through the retriever.
B2	The retriever requests the provenance metadata from the backend.
B3	The backend retrieves the requested paths and checksums from the SQL database.
B4	The backend returns the metadata to the retriever.
B5	The retriever copies the specified provenance artifacts from the data lake to their respective paths based on the metadata.
B6	The retriever should display the status of the artifact retrieval on the user's terminal (currently empty output).
C1	The user requests a visualization graph for a specific job via the visualizer.
C2	The visualizer forwards the request to the backend.
C3	The backend retrieves the required metadata from the SQL database.
C4	The backend sends the metadata back to the visualizer.
C5	The visualizer generates the graph and saves it in the current folder.

Table 3: Interactions between the components involved in the provenance auditing

4.11 Conclusion

In this section, the components making up the PAT have been explained, including the way they communicate with each other and how the users can interact with the entire system. The section covered how the PAT gathers the data, what data is being collected, how it is processed, and how it can ultimately be used. It also demonstrated the modularity of this approach, highlighting how other workload managers and provenance auditing methods can be applied without having to apply any changes to the vast majority of the codebase.

5 Validation

This section presents the benchmarks used to evaluate whether the PAT functions correctly across a variety of applications, explains the rationale behind them, and discusses the results. Section 5.1 provides an overview of the benchmarks. Sections 5.2 to 5.5 evaluate correctness, coverage, and robustness across synthetic and real-world workloads. Finally, Section 5.6 summarizes the findings and assesses the applicability of the PAT in HPC systems.

5.1 Benchmark Overview

To validate the correctness, coverage, and practical applicability of the PAT, a set of targeted tests was conducted.

- **Synthetic Hook Tests**: Scripts in several languages invoking the specific `libc` functions that are being traced
- **Fork and Exec Test**: A test designed to verify that propagation throughout forks and execs is successful and that the PAT processes these operations correctly
- **Multithreading and Multiprocess Test**: A test to verify that multithreading and multiple processes work correctly
- **Real World Computations**: The PAT is being applied to multiple processes that one would commonly encounter in real world HPC

5.2 Synthetic Hook Tests

Purpose This benchmark suite is designed to systematically exercise each individual `libc` hook interposed by the PAT across a range of programming languages. The primary objective is to verify that each hook operates correctly and transparently, without introducing unintended side effects or behavioral changes in the target process, and to assess this across different language runtimes. Go is intentionally included as a contrasting case, since it does not instrument or rely on traditional `libc` function calls, which demonstrates the limitations of the approach.

Implementation This test group consisted of a set of benchmarks designed to execute each `libc` function for which the PAT hooks. Functionally equivalent programs were executed in multiple languages. Benchmarks were implemented in C, Python, R, Julia, and Go. The C benchmark was compiled using both `gcc` and `clang` to check for compiler-dependent differences. The full implementations in C and Python are provided in Appendix B.1 and Appendix B.2 respectively.

Results For C and Python, all benchmarked operations were recorded as expected. This indicates that the corresponding hooks function correctly and that the tested operations are routed through `libc` functions in these languages. For R, coverage was only partial. Only a few of the used `libc` functions were recorded, while the remaining operations were systematically missing, demonstrating that parts of R's runtime bypass `libc` functions. Almost no relevant activity was recorded using Go aside from process creation. This is due

to Go's frequent use of direct system calls instead of `libc` functions. When using Julia, the benchmark crashed when executed with the PAT. This behavior was attributable to interposition of `_exit`-family functions, which are used to ensure immediate shutdown with minimal additional operations. The PAT is actively violating these constraints by hooking this family.

5.3 Fork and Exec Test

Purpose This test is designed to evaluate how the PAT handles process creation and image replacement via `fork` and `exec`, operations that require special consideration as outlined in the methodology section. Since `fork` is not directly interposed, parent–child relationships are reconstructed within the processor using the child process's PPID to establish lineage. In contrast, `exec` calls are explicitly hooked but require additional handling in the processing step. The test further verifies that distinct `exec` invocations are internally modeled as separate process instances. It also ensures that the `LD_PRELOAD` mechanism used to inject the `libc` interposer is correctly propagated throughout the execution tree.

Implementation When the program starts, it writes to a file and calls `fork()` exactly once to create a child process. In the child, the program then performs multiple `execv()` calls that execute this program with different roles, replacing the process image multiple times and writing to files to provide feedback while the PID stays the same. In the final role, an additional `execv()` is executed with a deliberately non-existent path, which leads to an exec failure. Since `execv()` returns only on error, a following write only happens if the exec attempt failed.

Results All file accesses were successfully tracked by the PAT. As intended, the different execs were interpreted as separate processes. This demonstrates that the PAT is successfully propagated through the process tree and that the processor is interpreting the execs as well as the failed exec as intended.

5.4 Multithreading and Multiprocess Test

Purpose This benchmark was designed to evaluate whether the provenance auditing tool maintains correctness and stability under concurrent execution. Because multiple concurrency paradigms exist and each has distinct synchronization and scheduling characteristics, dedicated testing across representative patterns is required. Robust performance in this domain is particularly critical, as HPC environments rely extensively on parallelism and fine-grained concurrency.

Implementation The program combines three concurrency patterns that are commonly found in HPC applications: OpenMP [59], `pthreads`, and process-level parallelism via `fork()`. First, an OpenMP matrix-addition kernel is executed to invoke a common OpenMP runtime activity. The result is written to a file to provide an indication of this event.

Next, a threading test is executed using `pthread_create()`/`pthread_join()`, where each thread performs various I/O operations, exercising multiple `libc` functions under concurrent access. After the multithreading in the parent process, several child processes

are spawned using `fork()`. Each child repeats the same pattern, which consists of creating a file, renaming it, and deleting it. The parent waits for all children to complete and finally reopens the shared thread output files to ensure that post-execution reads are also observed.

Results The PAT tracks all the operations outlined successfully, highlighting that deadlocks introduced by the auditing mechanism under contention do not occur.

5.5 Real World Computations

Purpose Since real-world computations, particularly in HPC, are substantially more complex and potentially more sensitive than the synthetic benchmarks evaluated previously, a set of benchmarks representative of typical HPC workloads was employed. These workloads more accurately reflect the operational environment for which the PAT is intended. Evaluating the system under such realistic conditions is therefore a critical step in determining whether it can withstand the high degrees of concurrency, scale, and computational complexity found in modern HPC applications.

Implementation This test includes three benchmarks that reflect typical HPC workloads: a minimal machine learning workload using scikit-learn [60], a deep learning workload using PyTorch [61] and a representative GROMACS [62] run. The full implementations of the scikit-learn and PyTorch benchmarks are provided in Appendix B.3 and Appendix B.4 respectively. The minimized injector that reproduced the crash is provided in Appendix B.5 and the GROMACS job script is provided in Appendix B.6.

Results The scikit-learn and PyTorch benchmarks executed successfully under PAT, with the expected provenance information recorded and no observable deviations in application behavior. In contrast, the GROMACS benchmark exhibited reproducible crashes when executed with PAT enabled. Two independent sources of failure were identified. First, instrumentation of the `libc` functions `read` and `fgets` triggered crashes when the corresponding provenance operation structure was appended to the internal C++ queue, with each function leading to distinct crash behavior.

Notably, the failure could be reproduced in a reduced version in which the PAT performed no operations other than interposing these two functions and appending a struct containing a single integer to the queue. Second, separate crashes occurred during in-memory storage of the JSON-serialized provenance data. These results show that, while the interposition method works for several real workloads, this implementation is not robust enough to handle the high-frequency and highly concurrent I/O behavior of GROMACS. Given that the triggering code is so minimal, this is most likely an inherent constraint of `libc` interposition independent of how it is implemented, which is why no attempt was made to correct this issue. The minimized injector that reproduced the crash is shown in Appendix B.5.

5.6 Conclusion

In conclusion, the usability evaluation demonstrates that the `libc` interposition can be viable for simple C and Python programs, but it is not reliably applicable to real-world HPC workloads or to programs written in some other languages. The results demonstrate

that not every given program or programming language consistently executes relevant operations through interposable `libc` functions because direct system calls or alternative library implementations can bypass the interception and therefore avoid being tracked. In addition, the benchmarks show that interposition can introduce unexpected side effects that disrupt the host process. Even enqueuing a struct in C++, which is a seemingly harmless operation, can trigger crashes when faced with the high call rates and concurrency typical of HPC applications.

6 Performance Evaluation

This section evaluates the performance overhead introduced by the injector library. Section 6.1 describes the benchmarking setup. Section 6.2 presents fio [8] benchmark results across different library variants and I/O patterns. Section 6.3 applies perf [9] profiling to identify the root cause of the performance penalty caused by the PAT. Section 6.4 quantifies the additional system calls introduced by the injector. Finally, Section 6.5 summarizes and interprets the results.

6.1 Setup

The tests were run locally on Arch Linux (x86_64) with the 6.19.11-arch1-1 kernel, using an AMD Ryzen 7 7730U CPU and an `ext4` filesystem. The library was compiled with `gcc`. The system was otherwise left idle. No major programs besides Hyprland were running during the benchmarks.

In order to better understand the overhead introduced by the injector library, the provenance tool itself is not included in these benchmarks. This minimal setup allows for a more precise diagnosis of the different performance issues attributable to the injector.

Furthermore, for longer-running jobs, the overhead from the provenance tool processing the filenames and converting it for a response to the backend does not scale proportionally with runtime, since much of its complexity depends on the number of distinct files accessed rather than the total number of I/O operations, which is what drives the overhead of the library. And lastly, the prov tool also does many things related to backing up the files, like retrieving checksums and moving files. This would not be a fair comparison with other tools that do not do this.

6.2 fio Benchmarks

Fio is a flexible I/O benchmarking tool commonly used to measure storage and memory performance by generating configurable workloads. It provides a variety of I/O engines, including the `psync` engine, which can be used to simulate standard POSIX I/O. The configurations are highly flexible, and the performance metrics provided by fio are particularly useful. In particular, throughput (MiB/s) and IOPS can be directly inspected, as they are reported natively by fio.

6.2.1 Injector libraries

To highlight which components of the injector contribute most to the overhead, different versions of the PAT `libc` library are evaluated:

- **dummy**: This version only wraps the original `libc` functions without performing any logging.
- **no-mutex**: This version removes the mutex from the original implementation to demonstrate synchronization overhead.
- **no-fd-path**: This version omits the step in which the file path is resolved from a file descriptor, logging only the descriptor itself rather than the corresponding path. File descriptor path resolution was identified as the dominant source of overhead in

the root cause analysis presented below, and this variant is included to isolate and quantify its contribution to the overall throughput penalty.

- **unaltered**: The original implementation.

6.2.2 Tested in four scenarios

Different fio benchmarks are used to analyze how I/O behavior is affected by multithreading and request size. All files were read from and written to `/dev/shm` to further isolate the overhead from storage effects.

The benchmarks will be stored as `.fio` files, which define how they work, and they will be started through a bash script. This ensures consistency and reproducibility. The benchmark scripts are also available in the Appendix.

- `rnd_small.fio`: Random reads and writes with 4KB blocks
- `rnd_small_mt.fio`: Random reads and writes with 4KB blocks using multiple threads
- `seq_big.fio`: Sequential reads and writes with 4MB blocks
- `seq_big_mt.fio`: Sequential reads and writes with 4MB blocks using multiple threads

The `no-mutex` variant is not used in the multithreaded scenarios, as it would clearly lead to crashes and invalid results.

6.2.3 Results

All benchmarks are executed via a bash script, which also drops the page cache between runs. The following tables summarize the results of the fio benchmarks across the different library variants.

Table 4: fio benchmark results using `seq_big`

Variant	Op	IOPS	BW (MiB/s)	IOPS Δ
baseline	read	18,579	18,579.3	—
baseline	write	30,186	30,186.0	—
unaltered	read	16,616	16,615.8	−10.6%
unaltered	write	26,483	26,483.3	−12.3%
dummy	read	18,313	18,312.9	−1.4%
dummy	write	30,404	30,403.8	+0.7%
no-mutex	read	16,939	16,938.5	−8.8%
no-mutex	write	27,554	27,553.9	−8.7%
no-fd-to-path	read	18,330	18,329.9	−1.3%
no-fd-to-path	write	30,266	30,266.3	+0.3%

Table 5: fio benchmark results using `seq_big_mt`

Variant	Op	IOPS	BW (MiB/s)	IOPS Δ
baseline	read	20,752	20,751.7	—
baseline	write	12,905	12,905.2	—
unaltered	read	19,962	19,961.5	−3.8%
unaltered	write	13,295	13,295.5	+3.0%
dummy	read	21,058	21,057.9	+1.5%
dummy	write	12,369	12,369.3	−4.2%
no-fd-to-path	read	20,506	20,505.6	−1.2%
no-fd-to-path	write	12,569	12,568.6	−2.6%

Table 6: fio benchmark results using `rnd_small`

Variant	Op	IOPS	BW (MiB/s)	IOPS Δ
baseline	read	785,111	3,066.8	—
baseline	write	672,077	2,625.3	—
unaltered	read	242,991	949.2	−69.1%
unaltered	write	234,013	914.1	−65.2%
dummy	read	783,609	3,061.0	−0.2%
dummy	write	696,434	2,720.4	+3.6%
no-mutex	read	242,000	945.3	−69.2%
no-mutex	write	236,309	923.1	−64.8%
no-fd-to-path	read	690,822	2,698.5	−12.0%
no-fd-to-path	write	618,756	2,417.0	−7.9%

Table 7: fio benchmark results using `rnd_small_mt`

Variant	Op	IOPS	BW (MiB/s)	IOPS Δ
baseline	read	2,345,578	9,162.4	—
baseline	write	955,224	3,731.3	—
unaltered	read	807,417	3,154.0	−65.6%
unaltered	write	759,686	2,967.5	−20.5%
dummy	read	2,363,858	9,233.8	+0.8%
dummy	write	950,777	3,714.0	−0.5%
no-fd-to-path	read	2,013,525	7,865.3	−14.2%
no-fd-to-path	write	945,035	3,691.5	−1.1%

As is clearly visible, file descriptor to path resolution is the primary source of overhead. When this step is removed, the overhead becomes much more manageable. This is a very encouraging result and will be discussed further below.

6.3 perf Benchmarks

To support root cause analysis of the observed performance degradation and to investigate which components of the injector contribute most to overhead, perf was applied to the `rnd_small.fio` benchmark. This benchmark was chosen as it produces significantly higher injector overhead compared to `seq_big.fio`, making potential sources of slowdown easier to identify.

It is important to note that all perf measurements reflect CPU usage only and do not capture time spent blocked in the kernel during actual I/O or latency caused by storage operations. As a result, high CPU usage in perf does not necessarily correspond to a large impact on total I/O performance. In this context, fio benchmarks are a more meaningful measure of end-to-end throughput and IOPS, while perf provides insight into which functions consume CPU cycles and is useful for root cause analysis. Nevertheless, the relative CPU consumption of each component of the injector remains directionally informative, even if kernel-side I/O overhead remains outside the visibility of perf.

A standard perf report breaks down CPU usage into "self" and "children" time. However, the `libc` injector functions call many small helper functions, which leads to inaccuracies. Child time can be counted multiple times across different call paths, and self time is too fragmented to be meaningful. Instead, a flamegraph is used to better visualize how the injector impacts CPU utilization:

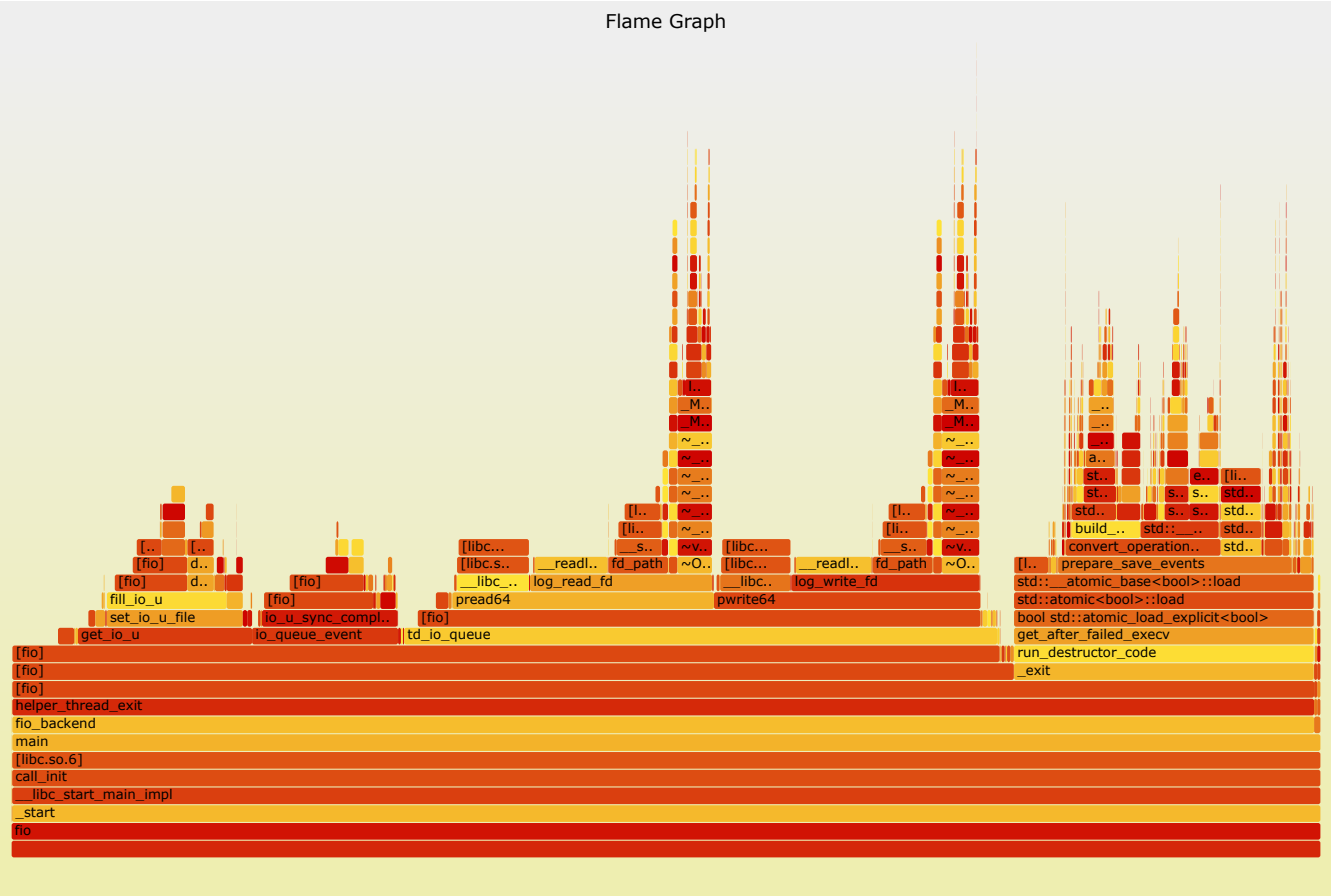

Figure 7: Breakdown of injector overhead using a flame graph

The functions highlighted in the flame graph are `log_read_fd`, `log_write_fd`, and `run_destructor_code`. Their CPU usage, including selected child functions, is as follows:

- `log_read_fd`: 13.91%, of which 5.56% is `_readlink` and 4.55% is `fd_path`
- `log_write_fd`: 14.35%, of which 5.76% is `_readlink` and 4.58% is `fd_path`
- `run_destructor_code`: 22.89%

As shown, resolving the file descriptor path for reads and writes is far more CPU intensive than the actual logging or data handling itself. We can also see that `run_destructor_code` introduces significant overhead because all provenance data is written to `/dev/shm` at program termination. It is evident that storing the data in `/dev/shm` is almost as costly as `log_read_fd` and `log_write_fd` combined, indicating that roughly half of the CPU overhead observed in the injector comes from saving provenance data, while the other half is dominated by file descriptor path resolution.

6.4 Syscall overhead

To quantify the syscall overhead introduced by `libinjector`, we compared the syscall traces of an instrumented and uninstrumented execution of the same workload using strace. The workload performs three `pread64`/`pwrite64` pairs on a single file.

6.4.1 Baseline Syscall Profile

The uninstrumented binary requires a minimal set of syscalls. Table 8 summarizes the syscalls observed in the baseline execution.

Table 8: Syscalls in the uninstrumented baseline execution

Syscall	Count
`pread64`	3
`pwrite64`	3
`write`	6
`openat`	2
`close`	2
`brk`	2
`fstat`	1
Total (I/O)	**6**
Total (all observed workload syscalls)	**19**

6.4.2 Injector Syscall Profile

Table 9 summarizes the syscalls observed in the execution where the injector was preloaded:

Table 9: Additional syscalls introduced by `libinjector`

Phase	**Syscall**	**Count**
Initialisation	`openat`	+4
Initialisation	`read`	+4
Initialisation	`fstat`	+4
Initialisation	`mmap`	+16
Initialisation	`close`	+4
Initialisation	`mprotect`	+4
Initialisation	`getcwd`	+1
Initialisation	`futex`	+1
First intercepted operation only	`getpid`	+1
First intercepted operation only	`getppid`	+1
First intercepted operation only	`openat`	+1
First intercepted operation only	`read`	+2
First intercepted operation only	`close`	+1
Per intercepted read/write op	`readlink`	+6
Teardown	`getpid`	+1
Teardown	`open`	+1
Teardown	`write`	+1
Teardown	`close`	+1
Total additional		**+52**

The table shows that 46 additional syscalls are incurred when using the PAT by default, as well as one syscall per I/O operation, which is used to get the filepath matching a file identifier. The readlink syscall is the reason the file lookup is so expensive and why the injector was so much slower in the fio benchmarks when using fd lookup.

However, other than that, nothing in the results is alarming. The 46 syscalls per process will remain constant regardless of the amount of I/O the process does. Therefore the overhead of these 46 syscalls is converging towards zero for more complex processes.

6.5 Conclusion

In conclusion, the primary source of time overhead introduced by the injector library is the resolution of file descriptor paths. Without this, read and write operations are only 8–14% slower. Considering that I/O operations typically account for only a fraction of the total time required for a scientific computation, the effective overhead on the entire computation is much lower. For instance, if a process spends one-third of its time on I/O, the overhead incurred would amount to only around 3%–5%.

This is in stark contrast to the substantial overhead caused when file descriptor paths are resolved. It is important to note that, unless a PAT operates at a very high level, every PAT must perform this resolution to accurately track which files are being accessed. Unfortunately, this cannot be deferred until after the process has finished.

As a result, many other PATs will also experience this overhead. Some may report lower overhead because they perform this lookup in a separate thread. Delegating file descriptor resolution to a dedicated thread or process could drastically reduce the overhead.

For example, LPS also performs this lookup, yet reports an overhead of only 1%. Once this issue is addressed, the performance results are quite satisfactory.

7 Discussion

This section discusses the results obtained in this thesis. Section 7.1 examines the limitations identified by the validation and performance benchmarks and highlights their implications. Section 7.2 outlines alternative approaches that could provide improved performance with respect to the previously identified limitations and evaluates their feasibility against the criteria described earlier.

7.1 Limitations

7.1.1 Applicability Limitations

The most apparent limitation identified a priori is that not all programs or programming languages rely on `libc` functions to perform I/O operations. As demonstrated by the usability benchmarks, however, the constraints extend beyond this issue. In order to persist provenance data at process termination, the PAT must interpose members of the `_exit` family. These functions are designed to guarantee immediate termination with minimal side effects. Interposing them necessarily violates their intended semantics and may introduce undefined behavior, which can destabilize the host application. This effect was observed in the Julia benchmark, where the instrumentation consistently led to program crashes.

Furthermore, the Gromacs benchmark highlights the sensitivity of real-world HPC workloads to even minimal interference. Such applications typically employ extreme levels of multithreading and operate under tight performance constraints, leaving little tolerance for additional synchronization, memory operations, or timing perturbations. Under these conditions, even slight invasiveness at the `libc` level can result in instability.

A detailed analysis of the GROMACS crashes is beyond the scope of this thesis. One potential mitigation strategy would be to transmit provenance data to the prov tool via a socket immediately for each operation, thereby avoiding in-process queuing and saving files to `/dev/shm`. However, even if such an approach were to resolve the specific failure observed here, the crashes highlight the fundamental concern that `libc` interposition cannot be assumed to be universally reliable, especially in complex, performance-critical applications. A fix tailored to this particular case would not guarantee stability across other computational workloads, and there is no way to establish with certainty that similar failures would not arise under different execution conditions.

The observed crashes therefore indicate that `libc` interposition cannot be considered a reliably safe auditing mechanism for HPC computations no matter how it is applied, as it introduces the possibility of application failure, which is an outcome that is unacceptable in production HPC environments.

7.1.2 Performance Limitations

The unaltered `libc` injector slows reads down by 69% and writes by 65% in the `rnd_small` benchmark, which is unacceptable for I/O-intensive workloads and far outside the performance targets of the PAT. Interestingly, this overhead does not originate from the sources one might expect. Neither dynamic loading nor mutex synchronization is the primary culprit. The dominant source of overhead is file descriptor path resolution, which introduces one additional syscall per intercepted I/O operation. This is an encouraging finding,

as it points to a concrete and addressable bottleneck. Without this resolution step, the slowdown reduces to 12% for reads and 8% for writes even with `rnd_small`, which is the most adversarial I/O pattern tested. This is considerably more acceptable given that real workloads spend more time on computation than I/O, and non-I/O execution is entirely unaffected by the injector.

It is important to note that, unless a PAT operates at a very high level of abstraction, file descriptor path resolution is unavoidable for accurate file access tracking and cannot be deferred until after process termination. Consequently, many other PATs face the same challenge. Those that report much lower overhead most likely delegate this resolution to a dedicated thread or process, thereby removing the latency through parallelization. Kernel instrumentation tools inherently benefit from this, as they observe the target process from the outside and perform all processing in parallel by design. LPS, for example, performs the same lookup yet reports an overhead of only 1%. Adopting a comparable approach for file descriptor resolution would likely yield substantial improvements, and once this issue is addressed, the overall performance of the injector should be quite satisfactory.

7.2 Alternative Approaches

As discussed in the Methodology section, kernel instrumentation remains the most promising approach, as it operates independently of the host processes and is therefore truly agnostic to the executed workload. Traditional system call tracing tools such as strace are known to be highly invasive and can impose substantial performance penalties. In contrast, modern mechanisms provide safer and more efficient alternatives. In particular, eBPF offers the ability to observe system-level events with comparatively low overhead and minimal interference with application execution.

To the best of our knowledge, eBPF has not been explored in the academic literature as a method for provenance auditing in HPC environments. This represents a missed opportunity, as it possesses many properties highly desirable for provenance auditing, namely good performance and zero disruption of host processes. Consequently, eBPF constitutes a promising direction for future research in system-level provenance capture.

8 Conclusion

This thesis has investigated the design and implementation of a provenance auditing tool intended for deployment in HPC environments. The work was motivated by the near-complete absence of automated provenance auditing in modern HPC infrastructures and the growing complexity of scientific workflows, which makes manual documentation increasingly insufficient.

A gap analysis of existing PATs revealed that no currently available system satisfies all requirements identified for practical HPC deployment. The closest candidate, LPS, addresses HPC-specific execution abstractions and reports acceptable overhead, but is not publicly available. This necessitated the design and implementation of a new PAT from scratch.

The implemented PAT employs `libc` interposition via `LD_PRELOAD` as its auditing mechanism, integrated with SLURM through a dedicated provenance tool. The system captures file access operations across process trees, processes the collected data, and transmits it to a backend for persistent storage. It also provides a visualizer and querier for downstream analysis. Artifacts are staged to a data lake to support long-term reproducibility and can be retrieved to support repeated execution of prior computations.

The validation benchmarks demonstrated that the approach functions correctly for C and Python workloads and handles fork- and exec-based process trees as intended. However, the benchmarks also revealed fundamental limitations. The Gromacs benchmark highlighted that even minimal `libc`-level interference can crash highly concurrent, performance-critical HPC applications, indicating that `libc` interposition cannot be assumed to be universally safe in production HPC environments and there is likely no way to make any `libc` interposition injector safe for this application.

The performance evaluation identified file descriptor path resolution as the dominant source of overhead, accounting for the majority of the observed 69% throughput reduction under adversarial I/O patterns. Without this step, overhead reduces to approximately 8–14% in the worst case. Delegating this resolution to a background thread or process, as done by tools such as LPS, would likely reduce overhead to acceptable levels.

The results show that while `libc` interposition could potentially be a viable auditing strategy for a small subset of HPC workloads, it cannot be applied universally. Kernel instrumentation via eBPF represents a more promising long-term direction since it has an even wider scope and does not have the problem of disrupting the job. Exploring eBPF as the basis for an HPC-native provenance auditing tool constitutes the most valuable direction for future work.

References

[1] G.E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. DOI: 10.1109/JPROC.1998.658762.

[2] Kristin M. Tolle, D. Stewart W. Tansley, and Anthony J. G. Hey. "The Fourth Paradigm: Data-Intensive Scientific Discovery". In: *Proceedings of the IEEE* 99.8 (2011), pp. 1334–1337. DOI: 10.1109/JPROC.2011.2155130.

[3] Yolanda Gil et al. "Examining the Challenges of Scientific Workflows". In: *Computer* 40.12 (2007), pp. 24–32. DOI: 10.1109/MC.2007.421.

[4] Beatriz Pérez, Julio Rubio, and Carlos Sáenz-Adán. "A systematic review of provenance systems". In: *Knowledge and Information Systems* 57.3 (2018), pp. 495–543. DOI: 10.1007/s10115-018-1164-3.

[5] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[6] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLoS ONE* 12.5 (May 2017). ISSN: ISSN 1932-6203. DOI: 10.1371/journal.pone.0177459. URL: https://www.osti.gov/biblio/1627824.

[7] Maxim Barnstorf. "Provenance auditing in the GWDG". In: (2025).

[8] Jens Axboe. *fio – Flexible I/O Tester*. https://github.com/axboe/fio. 2024.

[9] The Linux Kernel Developers. *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org. 2024.

[10] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. "The MPI Message Passing Interface Standard". In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by Karsten M. Decker and René M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8.

[11] Natalia Miloslavskaya and Alexander Tolstoy. "Big Data, Fast Data and Data Lake Concepts". In: *Procedia Computer Science* 88 (2016), pp. 300–305. ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.07.439. URL: https://www.sciencedirect.com/science/article/pii/S1877050916316957.

[12] Dipankar Mazumdar, Jason Hughes, and JB Onofre. *The Data Lakehouse: Data Warehousing and More*. 2023. arXiv: 2310.08697 [cs.DB]. URL: https://arxiv.org/abs/2310.08697.

[13] Free Software Foundation. *The GNU C Library (glibc)*. https://sourceware.org/glibc/. 2024.

[14] rofl0r and contributors. *proxychains-ng*. 2026. URL: https://github.com/rofl0r/proxychains-ng.

[15] Piotr Roszatycki and contributors. *fakechroot*. 2020. URL: https://github.com/dex4er/fakechroot.

[16] Ashish Gehani and Dawood Tariq. "SPADE: support for provenance auditing in distributed environments". In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. ontreal, Quebec, Canada: Springer-Verlag, 2012, pp. 101–120. ISBN: 9783642351693.

[17] F. Mölder et al. "Sustainable data analysis with Snakemake". In: *F1000Research* (2021). DOI: 10.12688/f1000research.29032.3.

[18] Thomas Kluyver et al. "Jupyter Notebooks — a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[19] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[20] Paul Kranenburg et al. *strace*. 2026. URL: https://github.com/strace/strace.

[21] SystemTap Developers. *Conditional Probes*. https://www.sourceware.org/systemtap/. 2025.

[22] Marcos A. M. Vieira et al. "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications". In: *ACM Comput. Surv.* 53.1 (2020). ISSN: 0360-0300. DOI: 10.1145/3371038. URL: https://doi.org/10.1145/3371038.

[23] Sarah Cohen-Boulakia et al. "Addressing the provenance challenge using ZOOM". In: *Concurr. Comput. : Pract. Exper.* 20.5 (Apr. 2008), pp. 497–506. ISSN: 1532-0626.

[24] L. Bavoil et al. "VisTrails: enabling interactive multiple-view visualizations". In: *VIS 05. IEEE Visualization, 2005.* 2005, pp. 135–142. DOI: 10.1109/VISUAL.2005.1532788.

[25] Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. "Provenance in collection-oriented scientific workflows". In: *Concurrency and Computation: Practice and Experience* 20.5 (2008), pp. 519–529. DOI: https://doi.org/10.1002/cpe.1226. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1226. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1226.

[26] Adriane Chapman et al. "PLUS: Provenance for life, the universe and stuff". In: *Proceedings of the VLDB Endowment* (Jan. 2010).

[27] Roger S. Barga and Luciano A. Digiampietri. "Automatic capture and efficient storage of e-Science experiment provenance". In: *Concurr. Comput. : Pract. Exper.* 20.5 (Apr. 2008), pp. 419–429. ISSN: 1532-0626.

[28] Ian T. Foster et al. "Chimera: AVirtual Data System for Representing, Querying, and Automating Data Derivation". In: *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*. SSDBM '02. USA: IEEE Computer Society, 2002, pp. 37–46. ISBN: 0769516327. DOI: 10.1109/SSDM.2002.1029704. URL: https://doi.org/10.1109/SSDM.2002.1029704.

[29] Tom Oinn et al. "Taverna: a tool for the composition and enactment of bioinformatics workflows". In: *Bioinformatics* 20.17 (Nov. 2004), pp. 3045–3054. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bth361. URL: https://doi.org/10.1093/bioinformatics/bth361.

[30] Felix Bartusch, Maximilian Hanussek, and Jens Krüger. "Automatic Generation of Provenance Metadata during Execution of Scientific Workflows". In: *IWSG*. 2018. URL: https://api.semanticscholar.org/CorpusID:155086949.

[31] Ewa Deelman et al. "Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems". In: *Scientific Programming* 13.3 (2005), p. 128026. DOI: https://doi.org/10.1155/2005/128026. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2005/128026. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2005/128026.

[32] Hyunjung Park, Robert Ikeda, and Jennifer Widom. "RAMP: a system for capturing and tracing provenance in MapReduce workflows". In: *Proc. VLDB Endow.* 4.12 (Aug. 2011), pp. 1351–1354. ISSN: 2150-8097. DOI: 10.14778/3402755.3402768. URL: https://doi.org/10.14778/3402755.3402768.

[33] Sheeba Samuel and Birgitta König-Ries. "ProvBook: Provenance-based Semantic Enrichment of Interactive Notebooks for Reproducibility". In: *Proceedings of the 17th International Semantic Web Conference (ISWC 2018) - Demo and Poster Track*. Vol. 2180. CEUR Workshop Proceedings. Monterey, California, USA, 2018, pp. 57–60. URL: https://ceur-ws.org/Vol-2180/paper-57.pdf.

[34] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. "A Framework for Collecting Provenance in Data-Centric Scientific Workflows". In: *2006 IEEE International Conference on Web Services (ICWS'06)*. 2006, pp. 427–436. DOI: 10.1109/ICWS.2006.5.

[35] Paul Groth, Simon Miles, and Luc Moreau. "PReServ: Provenance Recording for Services". In: (Dec. 2010).

[36] Leonardo Guerreiro Azevedo et al. "Experiencing ProvLake to Manage the Data Lineage of AI Workflows". In: *Anais Estendidos do XVI Simpósio Brasileiro de Sistemas de Informação (Anais Estendidos do SBSI 2020)* (2020). URL: https://api.semanticscholar.org/CorpusID:234655231.

[37] Sebastiaan Huber et al. "AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance". In: *Scientific data* 7 (Sept. 2020), p. 300. DOI: 10.1038/s41597-020-00638-4.

[38] Peter Macko and Margo Seltzer. "A general-purpose provenance library". In: *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*. TaPP'12. Boston, MA: USENIX Association, 2012, p. 6.

[39] Barbara S. Lerner and Emery R. Boose. "RDataTracker and DDG Explorer". In: *Revised Selected Papers of the 5th International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes - Volume 8628*. IPAW 2014. Cologne, Germany: Springer-Verlag, 2014, pp. 288–290. ISBN: 9783319164618. DOI: 10.1007/978-3-319-16462-5_36. URL: https://doi.org/10.1007/978-3-319-16462-5_36.

[40] Ragib Hasan, Radu Sion, and Marianne Winslett. "Preventing history forgery with secure provenance". In: *ACM Trans. Storage* 5.4 (Dec. 2009). ISSN: 1553-3077. DOI: 10.1145/1629080.1629082. URL: https://doi.org/10.1145/1629080.1629082.

[41] Runzhou Han et al. "PROV-IO: An I/O-Centric Provenance Framework for Scientific Data on HPC Systems". In: *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '22. Minneapolis, MN, USA: Association for Computing Machinery, 2022, pp. 213–226. ISBN: 9781450391993. DOI: 10.1145/3502181.3531477. URL: https://doi.org/10.1145/3502181.3531477.

[42] Dong Dai et al. "Lightweight Provenance Service for High-Performance Computing". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017, pp. 117–129. DOI: 10.1109/PACT.2017.14.

[43] David Holland et al. "PASSing the provenance challenge". In: *Concurrency and Computation: Practice and Experience* 20 (Apr. 2008), pp. 531–540. DOI: 10.1002/cpe.1227.

[44] James Frew and Peter Slaughter. "ES3: A Demonstration of Transparent Provenance for Scientific Computation". In: *Provenance and Annotation of Data and Processes: Second International Provenance and Annotation Workshop, IPAW 2008, Salt Lake City, UT, USA, June 17-18, 2008. Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 200–207. ISBN: 9783540899648. URL: https://doi.org/10.1007/978-3-540-89965-5_21.

[45] Thomas Pasquier et al. "Practical whole-system provenance capture". In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, pp. 405–418. ISBN: 9781450350280. DOI: 10.1145/3127479.3129249. URL: https://doi.org/10.1145/3127479.3129249.

[46] Philip J. Guo and Margo Seltzer. "BURRITO: wrapping your lab notebook in computational infrastructure". In: *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*. TaPP'12. Boston, MA: USENIX Association, 2012, p. 7.

[47] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. "Tracing the lineage of view data in a warehousing environment". In: *ACM Trans. Database Syst.* 25.2 (June 2000), pp. 179–227. ISSN: 0362-5915. DOI: 10.1145/357775.357777. URL: https://doi.org/10.1145/357775.357777.

[48] Andrew Runnalls and Chris Silles. "Provenance Tracking in R". In: *Provenance and Annotation of Data and Processes*. Ed. by Paul Groth and James Frew. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 237–239. ISBN: 978-3-642-34222-6.

[49] Yael Amsterdamer et al. "Putting lipstick on pig: enabling database-style workflow provenance". In: *Proc. VLDB Endow.* 5.4 (Dec. 2011), pp. 346–357. ISSN: 2150-8097. DOI: 10.14778/2095686.2095693. URL: https://doi.org/10.14778/2095686.2095693.

[50] Bahareh Sadat Arab et al. "A Generic Provenance Middleware for Database Queries, Updates, and Transactions". In: June 2014.

[51] Yong Zhao et al. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation". In: *2007 IEEE Congress on Services (Services 2007)*. 2007, pp. 199–206. DOI: 10.1109/SERVICES.2007.63.

[52] Brett Bode et al. "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters". In: *4th Annual Linux Showcase & Conference (ALS 2000)*. Atlanta, GA: USENIX Association, Oct. 2000. URL: https://www.usenix.org/conference/als-2000/portable-batch-scheduler-and-maui-scheduler-linux-clusters.

[53] Maxim Barnstorf. *libcprov*. https://github.com/irlite/libcprov. GitHub repository, commit b29928787c24597e6c84f901a6b3cade7ca4b66b, accessed 2026-05-26.

[54] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (2019), pp. 941–960. DOI: 10.1007/s00778-019-00578-5.

[55] SQLite Development Team. *SQLite*. Self-contained, serverless, zero-configuration SQL database engine; version 3.53.1. URL: https://www.sqlite.org/.

[56] Ian Ward. *JSON Lines: Documentation for the JSON Lines Text File Format*. https://jsonlines.org/.

[57] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. https://www.postgresql.org.

[58] Yasuyo G. Ichihara et al. "Color universal design: the selection of four easily distinguishable colors for all color vision types". In: *Electronic imaging*. 2008. URL: https://api.semanticscholar.org/CorpusID:18296783.

[59] Barbara Mary Chapman, Gabriele Jost, and Ruud van der Pas. "Using OpenMP - portable shared memory parallel programming". In: *Scientific and engineering computation*. 2007. URL: https://api.semanticscholar.org/CorpusID:4680444.

[60] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *CoRR* abs/1201.0490 (2012). arXiv: 1201.0490. URL: http://arxiv.org/abs/1201.0490.

[61] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[62] Mark James Abraham et al. "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers". In: *SoftwareX* 1-2 (2015), pp. 19–25. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2015.06.001. URL: https://www.sciencedirect.com/science/article/pii/S2352711015000059.

A Component Communication

A.1 Querier JSON Structures

Jobs query request sent to the backend:


```
{"query_type":"jobs","payload":{"user":"example_user","before":"","after":""}}
```


Jobs query response returned to the querier:


```
{
  "jobs": [
    {
      "job_id": 1,
      "cluster_name": "cname1",
      "job_name": "test_name",
      "username": "example_user",
      "start_time": 1779639581212689778,
      "end_time": 1779639581242301382,
      "path": "",
      "json": "{}"
    }
  ]
}
```


Execs query request sent to the backend:


```
{"query_type":"execs","payload":{"job_id":1,"cluster":"cname1","files":true}}
```


Execs query response returned to the querier:


```
{
  "execs": [
    {
      "start_time": 1779639581218452557,
      "exec_id": 1,
      "processes": [
        {
          "process_command": "first_exec_child1",
          "process_id": 3,
          "env_variable_hash": 11319602362574279025,
          "operations": {
            "/dev/shm/ts1/file11.txt": ["write"]
          }
        },
        {
          "process_command": "./test_scripts/first_exec",
          "process_id": 2,
          "env_variable_hash": 11319602362574279025,
          "operations": {
            "/dev/shm/ts1/file1.txt": ["write","deleted"],
            "/dev/shm/ts1/file2.txt": ["write"],
```

```
            "/dev/shm/ts1/file10.txt": ["write"]
          }
        },
        {
          "process_command": "sh -c ./test_scripts/first_exec",
          "process_id": 1,
          "env_variable_hash": 11319602362574279025,
          "operations": {}
        }
      ],
      "execute_map": [
        {"parent_process_id": 2, "child_process_id_array": [3]},
        {"parent_process_id": 1, "child_process_id_array": [2]}
      ],
      "rename_map": {},
      "env_variable_hash_pair_array": [],
      "json": "{}",
      "path": "/dev/shm/ts1",
      "command": "./test_scripts/first_exec"
    }
  ]
}
```


Processes query request sent to the backend:


```
{"query_type":"processes","payload":{"exec_id":2,"files":true}}
```


Processes query response returned to the querier:


```
{
  "processes": [
    {
      "process_command": "./test_scripts/second_exec_child1 example_param",
      "process_id": 6,
      "env_variable_hash": 11319602362574279025,
      "operations": {
        "/dev/shm/ts1/file2.txt": ["write"]
      }
    },
    {
      "process_command": "./test_scripts/second_exec",
      "process_id": 5,
      "env_variable_hash": 11319602362574279025,
      "operations": {
        "/dev/shm/ts1/file10.txt": ["read"],
        "/dev/shm/ts1/file8.txt": ["read"],
        "/dev/shm/ts1/file2.txt": ["read","deleted"],
        "/dev/shm/ts1/file4.txt": ["read","write","deleted"],
        "/dev/shm/ts1/file6.txt": ["read","write"]
      }
    },
    {
```

```
      "process_command": "sh -c ./test_scripts/second_exec",
      "process_id": 4,
      "env_variable_hash": 11319602362574279025,
      "operations": {}
    }
  ]
}
```


A.2 Retriever JSON Structures

Job retrieval request sent to the backend with:

```
./db_interface retrieval job 1 cname1
```


```
{
  "retriever_request_type": "job",
  "payload": {
    "job_id": 1,
    "cluster": "cname1"
  }
}
```


Job retrieval response returned to the retriever:


```
{
  "status": 0,
  "payload": {
    "exec_operations_map": {
      "1": {
        "/dev/shm/ts1/file1.txt": {
          "performed_operations": ["write","deleted"],
          "start_checksum": "",
          "end_checksum": ""
        },
        "/dev/shm/ts1/file2.txt": {
          "performed_operations": ["write"],
          "start_checksum": "",
          "end_checksum":
          ↪  "92ac357dc98836f4636786af8807935db7e5cf6864d8ab72101e009662bf998d"
        },
        "/dev/shm/ts1/file3.txt": {
          "performed_operations": ["write"],
          "start_checksum": "",
          "end_checksum":
          ↪  "55ac036a7b94a741d02a4e63a5e59c43e3c16aaf9f1af5926492e6735ab843ef"
        },
        "/dev/shm/ts1/file4.txt": {
          "performed_operations": ["write"],
          "start_checksum": "",
          "end_checksum":
          ↪  "af3f8f5e9e83ef61358ddc9567685cc17984a8c66fc2a7372b42b9c680911571"
```

```
      },
      "/dev/shm/ts1/file5.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "996d3ef4755b645aa6f4575f1cf68b880d9021094583e2c7ace85e7a2a483a29"
      },
      "/dev/shm/ts1/file6.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "da11f78133f387ee74581e9bb5981a0a0fd1e87437766eb534e6cc9a9d178fed"
      },
      "/dev/shm/ts1/file7.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "8c02ccd2360385138537db33fa5c1411b2deddffd8493af6f37810cf8f21ac95"
      },
      "/dev/shm/ts1/file8.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "adebc421d22032b6108fb858c1c5741b23755c9140cb8746caedf1aa02117535"
      },
      "/dev/shm/ts1/file9.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "c2753d272d674f2c3423da5381cca24a0cf889e53298207c6b216f3dc7ed5d55"
      },
      "/dev/shm/ts1/file10.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "9da053f81942358a2b76cc6e4f234d587b7026684554dc010b481a7b95581ddd"
      },
      "/dev/shm/ts1/file11.txt": {
        "performed_operations": ["write"],
        "start_checksum": "",
        "end_checksum":
        ↪ "d6776ffaa7b9585f1e8018f2692c42816834353ad9f54e7d77022b7724584193"
      }
    },
    "2": {
      "/dev/shm/ts1/file2.txt": {
        "performed_operations": ["read","write","deleted"],
        "start_checksum":
        ↪ "92ac357dc98836f4636786af8807935db7e5cf6864d8ab72101e009662bf998d",
        "end_checksum":
        ↪ "d6776ffaa7b9585f1e8018f2692c42816834353ad9f54e7d77022b7724584193"
```

```
        },
        "/dev/shm/ts1/file4.txt": {
          "performed_operations": ["read","write","deleted"],
          "start_checksum":
          ↪  "af3f8f5e9e83ef61358ddc9567685cc17984a8c66fc2a7372b42b9c680911571",
          "end_checksum": ""
        },
        "/dev/shm/ts1/file6.txt": {
          "performed_operations": ["read","write"],
          "start_checksum":
          ↪  "da11f78133f387ee74581e9bb5981a0a0fd1e87437766eb534e6cc9a9d178fed",
          "end_checksum":
          ↪  "9ab3c1184a9eec377e14f1782ac9362bbb586225e2d6269a81d5cf2a2390bc9a"
        },
        "/dev/shm/ts1/file8.txt": {
          "performed_operations": ["read"],
          "start_checksum":
          ↪  "adebc421d22032b6108fb858c1c5741b23755c9140cb8746caedf1aa02117535",
          "end_checksum": ""
        },
        "/dev/shm/ts1/file10.txt": {
          "performed_operations": ["read"],
          "start_checksum":
          ↪  "9da053f81942358a2b76cc6e4f234d587b7026684554dc010b481a7b95581ddd",
          "end_checksum": ""
        }
      }
    }
  }
}
```


exec retrieval request sent to the backend with:

```
./db_interface retrieval exec 1
```


```
{
  "retriever_request_type": "exec",
  "payload": {
    "exec_id": 1
  }
}
```


Execution retrieval response returned to the retriever:


```
{
  "status": 0,
  "payload": {
    "exec_operations_map": {
      "1": {
        "/dev/shm/ts1/file1.txt": {
          "performed_operations": ["write","deleted"],
          "start_checksum": "",
          "end_checksum": ""
```

```
    },
    "/dev/shm/ts1/file2.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "92ac357dc98836f4636786af8807935db7e5cf6864d8ab72101e009662bf998d"
    },
    "/dev/shm/ts1/file3.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "55ac036a7b94a741d02a4e63a5e59c43e3c16aaf9f1af5926492e6735ab843ef"
    },
    "/dev/shm/ts1/file4.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "af3f8f5e9e83ef61358ddc9567685cc17984a8c66fc2a7372b42b9c680911571"
    },
    "/dev/shm/ts1/file5.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "996d3ef4755b645aa6f4575f1cf68b880d9021094583e2c7ace85e7a2a483a29"
    },
    "/dev/shm/ts1/file6.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "da11f78133f387ee74581e9bb5981a0a0fd1e87437766eb534e6cc9a9d178fed"
    },
    "/dev/shm/ts1/file7.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "8c02ccd2360385138537db33fa5c1411b2deddffd8493af6f37810cf8f21ac95"
    },
    "/dev/shm/ts1/file8.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "adebc421d22032b6108fb858c1c5741b23755c9140cb8746caedf1aa02117535"
    },
    "/dev/shm/ts1/file9.txt": {
      "performed_operations": ["write"],
      "start_checksum": "",
      "end_checksum":
      ↪ "c2753d272d674f2c3423da5381cca24a0cf889e53298207c6b216f3dc7ed5d55"
    },
    "/dev/shm/ts1/file10.txt": {
      "performed_operations": ["write"],
```

```
          "start_checksum": "",
          "end_checksum":
          ↪  "9da053f81942358a2b76cc6e4f234d587b7026684554dc010b481a7b95581ddd"
        },
        "/dev/shm/ts1/file11.txt": {
          "performed_operations": ["write"],
          "start_checksum": "",
          "end_checksum":
          ↪  "d6776ffaa7b9585f1e8018f2692c42816834353ad9f54e7d77022b7724584193"
        }
      },
      "2": {
        "/dev/shm/ts1/file2.txt": {
          "performed_operations": ["read","write","deleted"],
          "start_checksum":
          ↪  "92ac357dc98836f4636786af8807935db7e5cf6864d8ab72101e009662bf998d",
          "end_checksum":
          ↪  "d6776ffaa7b9585f1e8018f2692c42816834353ad9f54e7d77022b7724584193"
        },
        "/dev/shm/ts1/file4.txt": {
          "performed_operations": ["read","write","deleted"],
          "start_checksum":
          ↪  "af3f8f5e9e83ef61358ddc9567685cc17984a8c66fc2a7372b42b9c680911571",
          "end_checksum": ""
        },
        "/dev/shm/ts1/file6.txt": {
          "performed_operations": ["read","write"],
          "start_checksum":
          ↪  "da11f78133f387ee74581e9bb5981a0a0fd1e87437766eb534e6cc9a9d178fed",
          "end_checksum":
          ↪  "9ab3c1184a9eec377e14f1782ac9362bbb586225e2d6269a81d5cf2a2390bc9a"
        },
        "/dev/shm/ts1/file8.txt": {
          "performed_operations": ["read"],
          "start_checksum":
          ↪  "adebc421d22032b6108fb858c1c5741b23755c9140cb8746caedf1aa02117535",
          "end_checksum": ""
        },
        "/dev/shm/ts1/file10.txt": {
          "performed_operations": ["read"],
          "start_checksum":
          ↪  "9da053f81942358a2b76cc6e4f234d587b7026684554dc010b481a7b95581ddd",
          "end_checksum": ""
        }
      }
    }
  }
}
```

A.3 Visualizer JSON Structures

JSON request issued by the visualizer:

```
{"job_id":1,"cluster_name":"cname1"}
```

Provenance data JSON received by the visualizer:


```
{
  "type": "prov_data",
  "payload": {
    "execs": [
      {
        "start_time": 1772684001025548300,
        "processes": [
          {
            "process_command": "child1",
            "process_id": 3,
            "env_variable_hash": 1234567890,
            "operations": {
              "/tmp/file1.txt": ["write"]
            }
          },
          {
            "process_command": "main_exec",
            "process_id": 2,
            "env_variable_hash": 1234567890,
            "operations": {
              "/tmp/file2.txt": ["write","deleted"]
            }
          }
        ],
        "execute_map": [
          {"parent_process_id": 2, "child_process_id_array": [3]}
        ],
        "env_variable_hash_pair_array": [
          {
            "env_variables_hash": 1234567890,
            "env_variables_array": [{"HOME": "/home/example_user"},{"PATH":
            ↪  "/usr/bin"}]
          }
        ],
        "path": "/home/example_user",
        "command": "main_exec"
      },
      {
        "start_time": 1772684001036213000,
        "processes": [
          {
            "process_command": "child2 param",
            "process_id": 6,
            "env_variable_hash": 1234567890,
```

```
            "operations": {"/tmp/file3.txt": ["write"]}
          },
          {
            "process_command": "second_exec",
            "process_id": 5,
            "env_variable_hash": 1234567890,
            "operations": {"/tmp/file4.txt": ["read","write"]}
          }
        ],
        "execute_map": [{"parent_process_id": 5, "child_process_id_array":
        ↪  [6]}],
        "env_variable_hash_pair_array": [
          {
            "env_variables_hash": 1234567890,
            "env_variables_array": [{"HOME": "/home/example_user"},{"PATH":
            ↪  "/usr/bin"}]
          }
        ],
        "path": "/home/example_user",
        "command": "second_exec"
      }
    ],
    "job_name": "test_name",
    "username": "example_user",
    "start_time": 1772684001018742000,
    "end_time": 1772684001045776000,
    "path": "/home/example_user"
  }
}
```

B Benchmark Code

B.1 Synthetic Hook Test in C

```c
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/wait.h>
#include <unistd.h>

static void die(const char* msg) {
```

```
    perror(msg);
    exit(1);
}

static void ensure_dir(const char* path) {
    if (mkdir(path, 0777) != 0 && errno != EEXIST) die("mkdir");
}

static void write_file(const char* path, const char* data) {
    int fd = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) die("open write_file");
    ssize_t n = write(fd, data, strlen(data));
    if (n < 0) die("write");
    if (close(fd) != 0) die("close");
}

static int open_ro(const char* path) {
    int fd = open(path, O_RDONLY);
    if (fd < 0) die("open_ro");
    return fd;
}

static int open_rw(const char* path) {
    int fd = open(path, O_RDWR);
    if (fd < 0) die("open_rw");
    return fd;
}

static void test_write_family(const char* dir) {
    char p1[512], p2[512], p3[512];
    snprintf(p1, sizeof(p1), "%s/write.txt", dir);
    snprintf(p2, sizeof(p2), "%s/fprintf.txt", dir);
    snprintf(p3, sizeof(p3), "%s/pwrite.txt", dir);
    int fd1 = open(p1, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd1 < 0) die("open write.txt");
    if (write(fd1, "hello\n", 6) < 0) die("write");
    struct iovec iov[2];
    iov[0].iov_base = (void*)"vec";
    iov[0].iov_len = 3;
    iov[1].iov_base = (void*)"tor\n";
    iov[1].iov_len = 4;
    if (writev(fd1, iov, 2) < 0) die("writev");
    if (pwrite(fd1, "PWRITE\n", 7, 0) < 0) die("pwrite");
    if (close(fd1) != 0) die("close fd1");
    FILE* f = fopen(p2, "w");
    if (!f) die("fopen fprintf.txt");
    if (fputs("fputs\n", f) < 0) die("fputs");
    if (fputc('X', f) == EOF) die("fputc");
    if (fputc('\n', f) == EOF) die("fputc");
    if (fprintf(f, "fprintf %d\n", 123) < 0) die("fprintf");
    if (fwrite("fwrite\n", 1, 6, f) != 6) die("fwrite");
```

```
    if (fflush(f) != 0) die("fflush");
    if (fclose(f) != 0) die("fclose");
    int fd3 = open(p3, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd3 < 0) die("open pwrite.txt");
    if (pwrite64(fd3, "PWRITE64\n", 9, 0) < 0) die("pwrite64");
    if (close(fd3) != 0) die("close fd3");
    int fdout = open(p1, O_WRONLY | O_APPEND);
    if (fdout < 0) die("open append");
    if (dprintf(fdout, "dprintf %s\n", "ok") < 0) die("dprintf");
    if (close(fdout) != 0) die("close fdout");
}

static void test_read_family(const char* dir) {
    char src[512], dst[512];
    snprintf(src, sizeof(src), "%s/read_src.txt", dir);
    snprintf(dst, sizeof(dst), "%s/read_dst.txt", dir);
    write_file(src, "0123456789abcdef\n");
    int fdr = open_ro(src);
    char buf[8];
    if (read(fdr, buf, sizeof(buf)) < 0) die("read");
    if (pread(fdr, buf, sizeof(buf), 2) < 0) die("pread");
    if (pread64(fdr, buf, sizeof(buf), 3) < 0) die("pread64");
    struct iovec iov[2];
    char b1[4], b2[4];
    iov[0].iov_base = b1;
    iov[0].iov_len = sizeof(b1);
    iov[1].iov_base = b2;
    iov[1].iov_len = sizeof(b2);
    if (readv(fdr, iov, 2) < 0) die("readv");
    if (close(fdr) != 0) die("close fdr");
    FILE* f = fopen(src, "r");
    if (!f) die("fopen read_src.txt");
    char line[64];
    if (!fgets(line, sizeof(line), f)) die("fgets");
    if (fgetc(f) == EOF) {
    }
    if (getc(f) == EOF) {
    }
    if (fseek(f, 0, SEEK_SET) != 0) die("fseek");
    char rbuf[16];
    fread(rbuf, 1, sizeof(rbuf), f);
    fclose(f);
    int fdw = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fdw < 0) die("open read_dst.txt");
    if (write(fdw, "ok\n", 3) < 0) die("write dst");
    close(fdw);
}

static void test_transfer_family(const char* dir) {
    char src[512], dst1[512], dst2[512];
    snprintf(src, sizeof(src), "%s/transfer_src.bin", dir);
```

```
    snprintf(dst1, sizeof(dst1), "%s/transfer_dst_sendfile.bin", dir);
    snprintf(dst2, sizeof(dst2), "%s/transfer_dst_copy.bin", dir);
    write_file(src, "abcdefghijklmnopqrstuvwxyz0123456789\n");
    int in = open_ro(src);
    int out = open(dst1, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (out < 0) die("open dst1");
    struct stat st;
    if (fstat(in, &st) != 0) die("fstat");
    off_t off = 0;
    if (sendfile(out, in, &off, (size_t)st.st_size) < 0) die("sendfile");
    close(out);
    close(in);
    int in2 = open_ro(src);
    int out2 = open(dst2, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (out2 < 0) die("open dst2");
    off64_t off_in = 0, off_out = 0;
    close(out2);
    close(in2);
}

static void test_rename_unlink(const char* dir) {
    char p1[512], p2[512], p3[512];
    snprintf(p1, sizeof(p1), "%s/rename_me.txt", dir);
    snprintf(p2, sizeof(p2), "%s/renamed.txt", dir);
    snprintf(p3, sizeof(p3), "%s/unlink_me.txt", dir);
    write_file(p1, "rename\n");
    if (rename(p1, p2) != 0) die("rename");
    write_file(p3, "unlink\n");
    if (unlink(p3) != 0) die("unlink");
    if (unlink(p2) != 0) die("unlink renamed");
}

static void test_exec_hooks(const char* dir, const char* self_path) {
    char child_path[512];
    snprintf(child_path, sizeof(child_path), "%s/child_exec_test", dir);
    int fd = open(child_path, O_WRONLY | O_CREAT | O_TRUNC, 0755);
    if (fd < 0) die("open child");
    const char* script
        = "#!/bin/sh\n"
          "echo child_exec_ok\n";
    if (write(fd, script, strlen(script)) < 0) die("write child");
    if (close(fd) != 0) die("close child");
    pid_t p = fork();
    if (p < 0) die("fork");
    if (p == 0) {
        char* const argv[] = {(char*)child_path, NULL};
        execve(child_path, argv, environ);
        _exit(127);
    }
    int st = 0;
    waitpid(p, &st, 0);
```

```
    pid_t p2 = fork();
    if (p2 < 0) die("fork2");
    if (p2 == 0) {
        execl(child_path, child_path, (char*)NULL);
        _exit(127);
    }
    waitpid(p2, &st, 0);
    pid_t p3 = fork();
    if (p3 < 0) die("fork3");
    if (p3 == 0) {
        char* const argv[] = {(char*)child_path, NULL};
        execv(child_path, argv);
        _exit(127);
    }
    waitpid(p3, &st, 0);
    pid_t p4 = fork();
    if (p4 < 0) die("fork4");
    if (p4 == 0) {
        execlp("sh", "sh", "-c", "echo execvp_ok", (char*)NULL);
        _exit(127);
    }
    waitpid(p4, &st, 0);
    pid_t p5 = fork();
    if (p5 < 0) die("fork5");
    if (p5 == 0) {
        char* const argv[]
            = {(char*)"sh", (char*)"-c", (char*)"echo execvpe_ok", NULL};
        execvpe("sh", argv, environ);
        _exit(127);
    }
    waitpid(p5, &st, 0);
    (void)self_path;
}

int main(int argc, char** argv) {
    const char* dir = "/dev/shm/prov_test";
    ensure_dir("/dev/shm");
    ensure_dir(dir);
    test_write_family(dir);
    test_read_family(dir);
    test_transfer_family(dir);
    test_rename_unlink(dir);
    const char* self_path = (argc > 0) ? argv[0] : "./prov_hook_test";
    test_exec_hooks(dir, self_path);
    return 0;
}
```

B.2 Synthetic Hook Test in Python

Synthetic hook test in Python:

```
#!/usr/bin/env python3
```

```python
import os
import time
import shutil
import subprocess

DIR = "/dev/shm/prov_test"

def ensure_dir(p):
    os.makedirs(p, exist_ok=True)

def write_file(path, data: bytes):
    with open(path, "wb") as f:
        f.write(data)

def test_write_family(dirp):
    p1 = os.path.join(dirp, "write.txt")
    p2 = os.path.join(dirp, "fprintf.txt")
    p3 = os.path.join(dirp, "pwrite.txt")
    fd = os.open(p1, os.O_WRONLY | os.O_CREAT | os.O_TRUNC, 0o644)
    os.write(fd, b"hello\n")
    os.write(fd, b"vec")
    os.write(fd, b"tor\n")
    if hasattr(os, "pwrite"):
        os.pwrite(fd, b"PWRITE\n", 0)
    os.close(fd)
    with open(p2, "w", buffering=1) as f:
        f.write("fputs\n")
        f.write("X\n")
        f.write(f"fprintf {123}\n")
        f.write("fwrite\n")
        f.flush()
    fd3 = os.open(p3, os.O_WRONLY | os.O_CREAT | os.O_TRUNC, 0o644)
    if hasattr(os, "pwrite"):
        os.pwrite(fd3, b"PWRITE64\n", 0)
    else:
        os.write(fd3, b"PWRITE64\n")
    os.close(fd3)
    fdout = os.open(p1, os.O_WRONLY | os.O_APPEND)
    os.write(fdout, f"dprintf {'ok'}\n".encode("utf-8"))
    os.close(fdout)

def test_read_family(dirp):
    src = os.path.join(dirp, "read_src.txt")
    dst = os.path.join(dirp, "read_dst.txt")
    write_file(src, b"0123456789abcdef\n")
    fd = os.open(src, os.O_RDONLY)
    os.read(fd, 8)
    if hasattr(os, "pread"):
        os.pread(fd, 8, 2)
        os.pread(fd, 8, 3)
    os.read(fd, 8)
```

```
    os.close(fd)
    with open(src, "r") as f:
        _ = f.readline()
        _ = f.read(1)
        f.seek(0)
        _ = f.read(16)
    with open(dst, "wb") as f:
        f.write(b"ok\n")

def test_transfer_family(dirp):
    src = os.path.join(dirp, "transfer_src.bin")
    dst1 = os.path.join(dirp, "transfer_dst_sendfile.bin")
    dst2 = os.path.join(dirp, "transfer_dst_copy.bin")
    write_file(src, b"abcdefghijklmnopqrstuvwxyz0123456789\n")
    if hasattr(os, "sendfile"):
        in_fd = os.open(src, os.O_RDONLY)
        out_fd = os.open(dst1, os.O_WRONLY | os.O_CREAT | os.O_TRUNC, 0o644)
        st = os.fstat(in_fd)
        offset = 0
        remaining = st.st_size
        while remaining > 0:
            sent = os.sendfile(out_fd, in_fd, offset, remaining)
            if sent == 0:
                break
            offset += sent
            remaining -= sent
        os.close(out_fd)
        os.close(in_fd)
    else:
        shutil.copyfile(src, dst1)
    shutil.copyfile(src, dst2)

def test_rename_unlink(dirp):
    p1 = os.path.join(dirp, "rename_me.txt")
    p2 = os.path.join(dirp, "renamed.txt")
    p3 = os.path.join(dirp, "unlink_me.txt")
    write_file(p1, b"rename\n")
    os.rename(p1, p2)
    write_file(p3, b"unlink\n")
    os.unlink(p3)
    os.unlink(p2)

def test_exec_hooks(dirp):
    child = os.path.join(dirp, "child_exec_test")
    script = "#!/bin/sh\necho child_exec_ok\n"
    with open(child, "w") as f:
        f.write(script)
    os.chmod(child, 0o755)
    subprocess.run([child], check=False)
    subprocess.run([child], check=False)
    subprocess.run(["sh", "-c", "echo execvp_ok"], check=False)
```

```
    env = dict(os.environ)
    subprocess.run(["sh", "-c", "echo execvpe_ok"], env=env, check=False)

def main():
    time.sleep(3)
    ensure_dir("/dev/shm")
    ensure_dir(DIR)
    test_write_family(DIR)
    test_read_family(DIR)
    test_transfer_family(DIR)
    test_rename_unlink(DIR)
    test_exec_hooks(DIR)

if __name__ == "__main__":
    main()
```

B.3 Machine Learning Benchmark using scikit-learn

```
import os
import shutil
import joblib
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score

DATA_DIR = "/dev/shm/sklearn_data"
MODEL_PATH = "/dev/shm/mnist_sgd_model.joblib"
METRICS_PATH = "/dev/shm/mnist_metrics.txt"
os.environ["SCIKIT_LEARN_DATA"] = DATA_DIR

try:
    X, y = fetch_openml(
        "mnist_784",
        version=1,
        as_frame=False,
        return_X_y=True,
        parser="liac-arff",
    )
    y = y.astype(int)
    X, _, y, _ = train_test_split(X, y, train_size=10_000, random_state=0, stratify=y)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0, stratify=y
    )
    clf = make_pipeline(
        StandardScaler(with_mean=False),
        SGDClassifier(loss="log_loss", max_iter=10, tol=1e-3, random_state=0),
    )
    clf.fit(X_train, y_train)
```

```
    pred = clf.predict(X_test)
    acc = accuracy_score(y_test, pred)
    print("accuracy:", acc)
    joblib.dump(clf, MODEL_PATH)
    with open(METRICS_PATH, "w") as f:
        f.write(f"accuracy={acc}\n")

finally:
    for p in (MODEL_PATH, METRICS_PATH):
        try:
            os.remove(p)
        except FileNotFoundError:
            pass
    shutil.rmtree(DATA_DIR, ignore_errors=True)
```

B.4 Deep Learning Benchmark using PyTorch

```
import os
import shutil
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

DATA_DIR = "/dev/shm/torch_data"
MODEL_PATH = "/dev/shm/cifar10_tiny_cnn.pt"
METRICS_PATH = "/dev/shm/cifar10_metrics.txt"

torch.set_num_threads(1)

class TinyCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(8, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16 * 8 * 8, 64), nn.ReLU(),
            nn.Linear(64, 10),
        )

    def forward(self, x):
        x = self.features(x)
        return self.classifier(x)

try:
    os.makedirs(DATA_DIR, exist_ok=True)
    transform = transforms.Compose([
```

```
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])
    trainset = torchvision.datasets.CIFAR10(
        root=DATA_DIR, train=True, download=True, transform=transform
    )
    testset = torchvision.datasets.CIFAR10(
        root=DATA_DIR, train=False, download=True, transform=transform
    )
    train_subset = torch.utils.data.Subset(trainset, range(1000))
    test_subset = torch.utils.data.Subset(testset, range(200))
    trainloader = torch.utils.data.DataLoader(
        train_subset, batch_size=32, shuffle=True, num_workers=0
    )
    testloader = torch.utils.data.DataLoader(
        test_subset, batch_size=64, shuffle=False, num_workers=0
    )
    device = torch.device("cpu")
    model = TinyCNN().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.02, momentum=0.9)
    model.train()
    for i, (images, labels) in enumerate(trainloader):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        if i >= 20:
            break
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in testloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            preds = outputs.argmax(dim=1)
            total += labels.size(0)
            correct += (preds == labels).sum().item()
    acc = correct / total
    print("accuracy:", acc)
    torch.save(model.state_dict(), MODEL_PATH)
    with open(METRICS_PATH, "w") as f:
        f.write(f"accuracy={acc}\n")

finally:
    for p in (MODEL_PATH, METRICS_PATH):
        try:
            os.remove(p)
```

```
        except FileNotFoundError:
            pass
    shutil.rmtree(DATA_DIR, ignore_errors=True)
```

B.5 Minimized Injector Causing Gromacs Crash

```
#include <dlfcn.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <cstddef>
#include <cstdint>
#include <queue>

struct Operation {
    uint64_t ts = 0;
};
static std::queue<Operation> operations = {};

static void log_read_fd(int path_in_fd) {
    operations.push(Operation{});
}

extern "C" {
#define SAVE_ERRNO int saved_errno = errno
#define RESTORE_ERRNO errno = saved_errno
#define RESOLVE_REAL(real_fn, sym1, sym2, failret)                          \
    do {                                                                    \
        if (!(real_fn)) {                                                   \
            (real_fn) = decltype(real_fn)(dlsym(RTLD_NEXT, (sym1)));        \
            if (!(real_fn))                                                 \
                (real_fn) = decltype(real_fn)(dlsym(RTLD_NEXT, (sym2)));    \
            if (!(real_fn)) return (failret);                               \
        }                                                                   \
    } while (0)

ssize_t read(int fd, void* buf, size_t count) {
    static auto real_read = (ssize_t (*)(int, void*, size_t)) nullptr;
    RESOLVE_REAL(real_read, "__libc_read", "read", (ssize_t)-1);
    ssize_t ret = real_read(fd, buf, count);
    SAVE_ERRNO;
    log_read_fd(fd);
    RESTORE_ERRNO;
    return ret;
}

char* fgets(char* s, int size, FILE* stream) {
    static auto real_fgets = (char* (*)(char*, int, FILE*)) nullptr;
    RESOLVE_REAL(real_fgets, "__libc_fgets", "fgets", NULL);
    char* ret = real_fgets(s, size, stream);
    SAVE_ERRNO;
```

```
    int fd = stream ? fileno(stream) : -1;
    log_read_fd(fd);
    RESTORE_ERRNO;
    return ret;
}
} // extern "C"
```

B.6 Gromacs Job Script

SLURM job script used for the Gromacs benchmark:

```
#!/bin/bash
#SBATCH --job-name="NHR-Emmy-P3-gromacs-maxim"
#SBATCH --output "maxim-%x_%j.o"
#SBATCH --error "maxim-gromacs.e"
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 32
#SBATCH --partition medium96s
#SBATCH --time 10:00

MODULES="gcc/11.5.0 openmpi/4.1.7 gromacs/2023.3"

module load ${MODULES}

export OMP_NUM_THREADS=32

source $(which GMXRC)

../../libcprov/build/injector/prov start --path "."
../../libcprov/build/injector/prov exec "gmx_mpi mdrun -s
↪  /sw/chem/gromacs/mpinat-benchmarks/benchPEP.tpr -nsteps 100 -ntomp
↪  ${OMP_NUM_THREADS} -dlb yes -v" --path "."
../../libcprov/build/injector/prov end
```