Dataflow Analysis of Serverless Scientific Applications using Provenance Data*

Marcello W. M. Ribeiro[1,2], Ubiratam de Paula[3], Liliane Kunstmann[4]
Yuri Frota[1], Isabel Rosseti[1], Daniel de Oliveira[1]

[1]Universidade Federal Fluminense (UFF), Niterói, RJ, Brazil

[2]Instituto Brasileiro de Geografia e Estatística (IBGE), RJ, Brazil

[3]Universidade Federal Rural do Rio de Janeiro (UFRRJ), RJ, Brazil

[4]Mendelics Análise Genômica, SP, Brazil

marcellomessina@id.uff.br, upaula@ufrrj.br,
liliane.kunstmann@mendelics.com.br, {yuri,rosseti,danielcmo}@ic.uff.br

Abstract. *This paper presents an approach to ease dataflow analysis in Computational Science and Engineering (CSE) applications that invoke serverless functions. By capturing provenance data while executing CSE applications within a serverless environment, the approach organizes this information into an integrated database that users can query at runtime. Since serverless platforms typically lack native support for provenance tracking, the proposed solution helps users understand and analyze the behavior and outcomes of their CSE applications. We detail the main features of the approach as implemented in the* `DENETHOR` *tool and evaluate its effectiveness with a real-world CSE application. The results demonstrate that* `DENETHOR` *enhances analytical capabilities via its provenance database.*

1. Introduction

Over the past few decades, the number of Computational Science and Engineering (CSE) applications has grown rapidly [Rude et al. 2018, de Oliveira et al. 2019]. These applications are multidisciplinary, typically developed by experts from various scientific knowledge domains [Kamble et al. 2017]. CSE applications are found in several fields, including Bioinformatics [Ocaña and de Oliveira 2015] and branches of engineering [Guerra et al. 2012]. Despite their differences, these applications share common characteristics: (i) they rely on complex software stacks, (ii) involve multi-step executions based on mathematical models and simulations, and (iii) frequently invoke third-party libraries and frameworks—often structured as dataflows [de Oliveira et al. 2019]. Moreover, each execution can run for extended periods. As a result, these applications often require high-performance computing (HPC) environments to deliver results within a feasible time.

Throughout the life cycle of a scientific experiment [Mattoso et al. 2010], a single CSE application may be executed multiple times to confirm or refute a hypothesis, often requiring the exploration of different parameter configurations or input datasets. Each execution involves configuring numerous parameters—many of which are difficult to define *a priori*. As a result, users may be unsure whether a given parameter combination will yield the

*The authors would like to thank the financial support from the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), and the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ).

desired outcome. Since such executions can be time-consuming and require several computing resources, monitoring and debugging capabilities become a top priority. These features allow users to identify potential issues during execution and understand their root causes.

Users can already take advantage of existing approaches for monitoring their CSE applications in both local and HPC environments [Silva et al. 2018, Pimentel et al. 2017]. Some of these solutions also support the analysis of the data derivation path during dataflow execution, *i.e.*, the analysis of provenance data [Herschel et al. 2017]. By examining provenance data, users can assess the current execution status and track the evolution of specific parameters or metrics of interest, such as p-value, accuracy, precision, and other domain-specific parameters. This analysis is typically performed over a defined time window rather than relying solely on the most recent values of these metrics.

Although these approaches represent a step forward, they are not designed to monitor certain classes of CSE applications that explore novel environments and computing paradigms. For instance, since 2009, many applications have migrated to the cloud, adopting the Software-as-a-Service (SaaS) model [Bux et al. 2015]. This transition has been fostered by cloud providers offering virtual machines (VMs) pre-configured with software, storage services, and other infrastructure components [Wen et al. 2021], simplifying CSE application development in the cloud. However, configuring and maintaining the entire (and complex) software stack still demands substantial human effort from users, primarily due to ongoing software updates and the challenges of managing dependencies.

Hence, the adoption of serverless functions—such as AWS Lambda[1]—has emerged as a way to simplify the development and deployment of distributed systems, particularly CSE applications that process large volumes of data [Hellerstein et al. 2019]. This paradigm frees users from infrastructure management, allowing them to focus on application logic while the cloud provider handles the container-based execution (*e.g.*, Docker). Figure 1 illustrates an abstract example of serverless function invocations within a CSE application. The application consists of four steps, denoted as A_1, A_2, A_3, and A_4, where three of these steps invoke distinct functions (A_1 invokes f_1, A_2 invokes f_2, and A_4 invokes f_3). Each function invocation involves loading data into a data store. Once the data is available, the function is triggered for execution. After execution, the output is stored in the data store, making it accessible to subsequent functions or users on their local machine. Due to these advantages, several CSE applications have started to adopt this paradigm [Hautz et al. 2023, Elshamy et al. 2023].

At first glance, serverless computing simplifies the development of large-scale CSE applications. However, as noted in [Khochare et al. 2022], this paradigm introduces limitations, including cold starts, message indirection, and the *lack of provenance support*. Since the execution environment is a "black box" from the CSE application's perspective, capturing detailed runtime information becomes challenging. To the best of our knowledge, only a few approaches [Datta et al. 2022, Satapathy et al. 2023, Huang et al. 2024, Ben-Shimol et al. 2025] have proposed solutions for capturing provenance in serverless applications, and none specifically target the unique needs of CSE applications and their associated dataflows. To address this gap, we propose in this paper a lightweight approach named DENETHOR[2], designed for runtime capture, management, monitoring, and analysis of distributed provenance data generated by CSE applications in serverless computing environ-

[1] https://aws.amazon.com/pt/pm/lambda/

[2] The name DENETHOR is inspired by Denethor, son of Ecthelion and the Steward of Gondor, a fictional character from J.R.R. Tolkien's The Lord of the Rings.

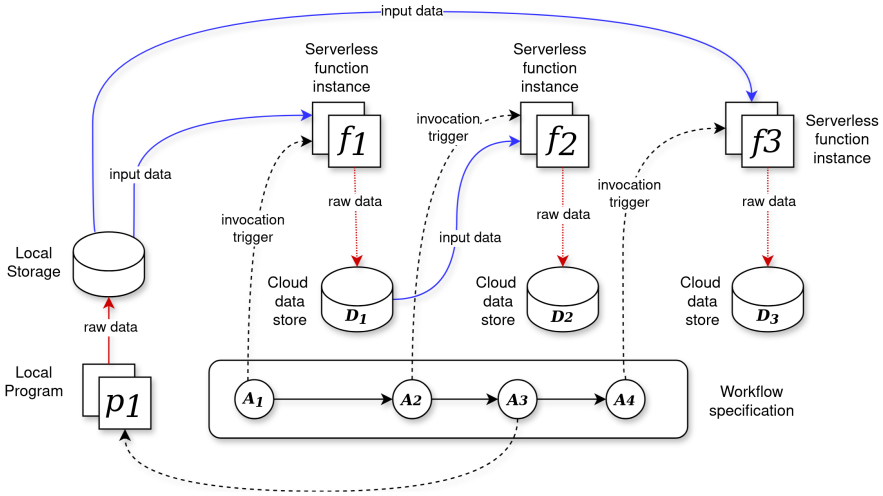

Figure 1. An example of implicit dataflow in a serverless CSE application.

ments. DENETHOR captures and stores function parameters along with the associated input and output data files for each invocation, thereby reconstructing the implicit dataflow within the CSE application. Importantly, DENETHOR allows users to specify which parameters are relevant for analysis, thereby avoiding the overhead of collecting unnecessary data. This selective capture enhances both performance and usability. By maintaining a detailed provenance trail of the application's dataflow, DENETHOR enables users to trace data dependencies, reproduce experiments, and diagnose issues during execution.

All captured provenance data is represented using the W3C PROV standard [Moreau and Groth 2013], ensuring interoperability and seamless data exchange with other PROV-compliant systems. DENETHOR can operate in hybrid environments, capturing provenance from serverless, local and VM-based executions. Furthermore, its compliance with the PROV standard enables integration with other provenance-capturing tools like NoWorkflow [Pimentel et al. 2017] and DfAnalyzer [Silva et al. 2018] to create a comprehensive view of local and distributed scientific executions. Provenance data is stored in a centralized database, which serves as a map of the application's execution. This enables users to analyze the entire dataflow of the CSE application over time. To evaluate the effectiveness of DENETHOR, we have chosen a real-world CSE application designed for mining frequent subtrees in phylogenetic databases. The results demonstrate that DENETHOR enhances the ability to analyze outputs and eases monitoring and debugging.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents important concepts such as the serverless paradigm and provenance data. Section 4 presents the DENETHOR architecture and its characteristics. Section 5 shows the DENETHOR evaluation with a real-world CSE application. Finally, Section 6 concludes the study and discusses future work.

2. Related Work

Monitoring and analyzing the execution of CSE applications in distributed environments is not a new challenge. Several approaches have been proposed, such as Magpie [Barham et al. 2004] and DTrace [Cantrill et al. 2004], to address it. However, these systems focus on monitoring the execution environment rather than the data derivation path, *i.e.*, unaware of the CSE application's implicit dataflow. Another limitation is that many of these

monitoring systems lack support for serverless architectures. On the other hand, some recent approaches have focused on capturing provenance in serverless environments and applications [Kiar et al. 2019, Datta et al. 2022, Satapathy et al. 2023].

Clowdr [Kiar et al. 2019] is a serverless platform for executing neuroscience experiments on both HPC and cloud environments. It captures provenance data to support analysis and avoids redundant executions following the W3C PROV. However, Clowdr is specific for neuroscience and requires users to adopt its execution platform, which may be a barrier for researchers used to specific frameworks. ALASTOR [Datta et al. 2022] is a tool that captures provenance data to trace malicious behavior in serverless applications. It monitors function invocations, data accesses, and storage operations to build a global provenance graph to audit executions and detect attacks. Despite its contributions, ALASTOR does not adopt the W3C PROV, limiting interoperability. It also lacks runtime provenance analysis, critical for long-running CSE applications.

Similarly to ALASTOR, DisProTrack [Satapathy et al. 2023] focuses on debugging through provenance data. It combines system and microservices logs to build a provenance graph. However, it does not follow the W3C PROV model, using custom relationships (*e.g.*, *listen*, *bind*, *connect*) that reduce applicability to other domains. FaaSRCA [Huang et al. 2024] is a root cause analysis method for serverless applications. It captures provenance and integrates platform- and application-level observability data via a Global Call Graph, addressing the dynamic nature of serverless environments. [Ben-Shimol et al. 2025] presented a threat detection model that leverages cloud-native monitoring tools to identify anomalies in serverless functions. While it does not capture the complete provenance graph (just some subset), it uses such data to detect potential compromises.

Xtract [Skluzacek et al. 2019] is a serverless-based system designed for automated metadata extraction from large volumes of scientific data. The extracted metadata provides structure and context, improving the organization and traceability of scientific data lakes. Xtract operates as a FaaS and supports metadata extraction from centralized servers and the edge. As manual metadata extraction becomes impractical at scale, Xtract orchestrates specialized extractor functions within dynamic workflows, enabling more effective data management and enhancing the value of scientific repositories. Finally, [Wen et al. 2025] propose SCOPE, a serverless-oriented performance testing approach, to provide precise and reliable tests. SCOPE determines the number of repetitions required for an experiment through accuracy and consistency checks that use non-parametric confidence intervals that the user collects. The data can be collected through logs, scripts, or approaches like `DENETHOR`.

3. Background

3.1. A Brief Tour Through Serverless Computing

Serverless is a computing paradigm where users focus only on application logic while the cloud provider manages the backend infrastructure [Hellerstein et al. 2019]. This approach suits scientific applications, as users often lack infrastructure expertise but can implement the application's logic, such as a mathematical model or a specialized algorithm. Serverless computing is typically categorized into two models: (i) Backend as a Service (BaaS) and (ii) Function as a Service (FaaS), each with its own pros and cons.

Under the BaaS model, users access external services via APIs, such as database services. In contrast, FaaS requires users to implement server-side logic. Applications run in

containers managed by the cloud provider. As containers are stateless, handling data persistence can be challenging. A common workaround is deploying DBMSs as functions. Unlike the IaaS model, where users manage the full VM lifecycle, FaaS abstracts infrastructure management. Users only need to build a container image with the application and its dependencies. Once deployed, the cloud provider handles function invocation and execution. Moreover, beyond its deployment simplicity, FaaS offers faster startup times. While a function typically starts within milliseconds to seconds, launching a VM can take 30 seconds.

Despite its advantages, serverless computing presents limitations. A key challenge is debugging, as the lack of provenance support makes it difficult to trace errors and understand the execution flow. Privacy is another concern. Since serverless functions run on shared infrastructure managed by the cloud provider, sensitive data may face risks due to potential cross-tenant interference, despite isolation mechanisms. Cost-effectiveness is also not always clear. Using a VM may be more economical for high-frequency or memory-intensive tasks. Choosing between serverless and traditional models requires carefully analyzing the application's resource and usage patterns.

3.2. Provenance in a Nutshell

Provenance [Herschel et al. 2017] refers to metadata that describes how a specific piece of data was produced, capturing its data derivation path. Its main purpose is to record this path in a structured, queryable format. Provenance is widely used to evaluate data quality and support reproducibility in scientific applications, as it provides detailed context, including input parameters, execution times, and errors found. This paper uses provenance data to better understand how CSE applications invoke serverless functions, enabling monitoring and intervention when execution results deviate from expectations.

The W3C proposes a recommendation for representing provenance data called W3C PROV [Moreau and Groth 2013]. This recommendation includes a family of documents that introduce a data model (PROV-DM), an ontology (PROV-O), a human-readable notation (PROV-N), and additional guidelines. The goal is to enable interoperability of provenance data across PROV-compliant systems. Provenance is described using three core components: (i) *Entities*, (ii) *Activities*, and (iii) *Agents*. An *Entity* denotes a physical or abstract object, such as a file or a parameter. An *Activity* represents an operation that acts upon entities, such as a serverless function invocation, and can include execution time and error details. An *Agent* is the user responsible for triggering the activity.

In addition to *Entities*, *Agents*, and *Activities*, the W3C PROV model defines a set of relationships that capture how these elements interact. While all relationships are specified in detail by [Moreau and Groth 2013], this section highlights the four most relevant to `DENETHOR`: (i) *Used*, (ii) *WasGeneratedBy*, (iii) *WasAssociatedWith*, and (iv) *WasDerivedFrom*. The *Used* relation indicates that an *Activity* consumed an *Entity*. In contrast, *WasGeneratedBy* denotes that an *Entity* was produced by an *Activity*. *WasAssociatedWith* links an *Agent* to the *Activity* initiated. Finally, *WasDerivedFrom* captures the transformation of one *Entity* into another. [Freire et al. 2008] categorize provenance into two types: (i) prospective provenance (p-prov) and (ii) retrospective provenance (r-prov). Prospective provenance refers to the defined sequence of steps in a dataflow, *i.e.*, within this paper's context, representing the CSE application's dataflow. Retrospective provenance captures metadata from the actual execution, such as function run times, errors, and parameters. `DENETHOR` focuses on capturing p-prov and r-prov. Finally, [Neves et al. 2017] defines implicit provenance as

provenance data that is not explicitly specified in the workflow definition.

4. Proposed Approach: DENETHOR

As previously discussed, serverless computing provides several benefits but lacks native support for provenance, making it difficult to analyze the implicit dataflow common in CSE applications that rely on serverless functions. To address this limitation, this section presents DENETHOR, a lightweight approach for analyzing the dataflow of serverless CSE applications through provenance data. DENETHOR architecture comprises seven main components, presented in Figure 2: (i) Broker, (ii) Monitor, (iii) Function Catalog, (iv) Provenance Importer, (v) Provenance Database, (vi) API, and (vii) Provenance Exporter.

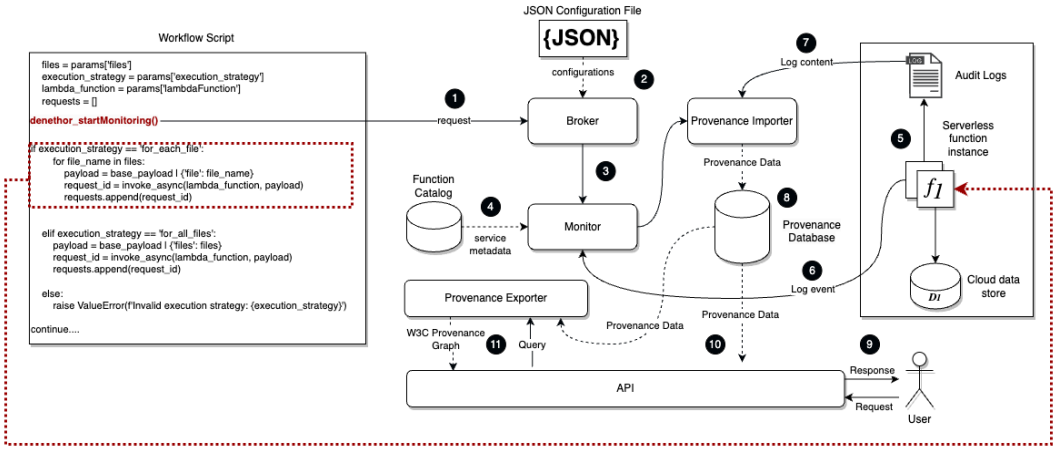

Figure 2. The Architecture of DENETHOR.

The use of DENETHOR begins with the execution of the CSE application. This application can be implemented using scripts, workflow systems, or big data frameworks. The key requirement is that it must be instrumented to send a message to the *Broker* (step ❶ in Figure 2) before invoking a serverless function. The message consists in: (i) a JSON file with the workflow specification and metrics to be captured, (ii) the environment settings, and (iii) calls to DENETHOR's methods at critical points of the CSE code.

Upon receiving this message, the *Broker* analyzes the input JSON file provided by the user (step ❷). This file contains which type of provenance has to be captured—specifying which data to collect and from which serverless functions. A fragment of the JSON file is shown in Figure 3, where the user lists the serverless functions (*e.g.*, AWS Lambda) and the associated files to track. If the metadata is valid, the *Monitor* is triggered (step ❸).

The *Monitor* component receives the user-defined metadata and accesses the *Function Catalog* (step ❹) to obtain information needed to monitor the serverless functions. The catalog includes details such as the cloud user account used to log in, whether the function has a static IP address, and other relevant configuration data. Using this information, the *Monitor* begins listening for the execution of the specified function within the cloud provider's environment (step ❺). After the function completes execution, audit logs are automatically generated, and a trigger message is sent to the *Monitor* (step ❻). This message prompts the *Monitor* to invoke the *Provenance Importer*, which then accesses the audit logs to extract provenance data (step ❼). The *Provenance Importer* has to be customized for each cloud provider. Currently, it is customized for AWS CloudWatch, and extending it to other providers requires implementing parsing rules for the new log file.

```
{
     "name": "Invoke function execution",
     "module": "function_invoker",
     "handler": "invoke_lambda_execution",
     "params": {
         "functionName": "tree_constructor",
         "inputBucket": "denethor-input-files",
         "outputBucket": "denethor-tree-files",
         "executionStrategy": "for_each_file"
     },
     "active": true
 },
 "dataFiles": {
         "path": "_data/full_dataset",
         "files": ["ORTHOMCL1","ORTHOMCL256"]
     }
```

Figure 3. A fragment of the JSON file containing metadata for r-prov capturing.

The extracted data is stored in the *Provenance Database* (step ❽), which follows the relational schema shown in Figure 4. This database is compliant with the W3C PROV recommendation—its contents can be exported in PROV-N format using simple SQL queries. The schema includes tables for dataflows, activities, executions, and related files, and we have augmented it with metadata such as cloud provider information, execution statistics, and function-specific attributes. This enables seamless integration of serverless provenance with local execution provenance collected by provenance systems such as DfAnalyzer and NoWorkflow. The database can be deployed on a separate VM or physical machine to avoid impacting CSE application performance.

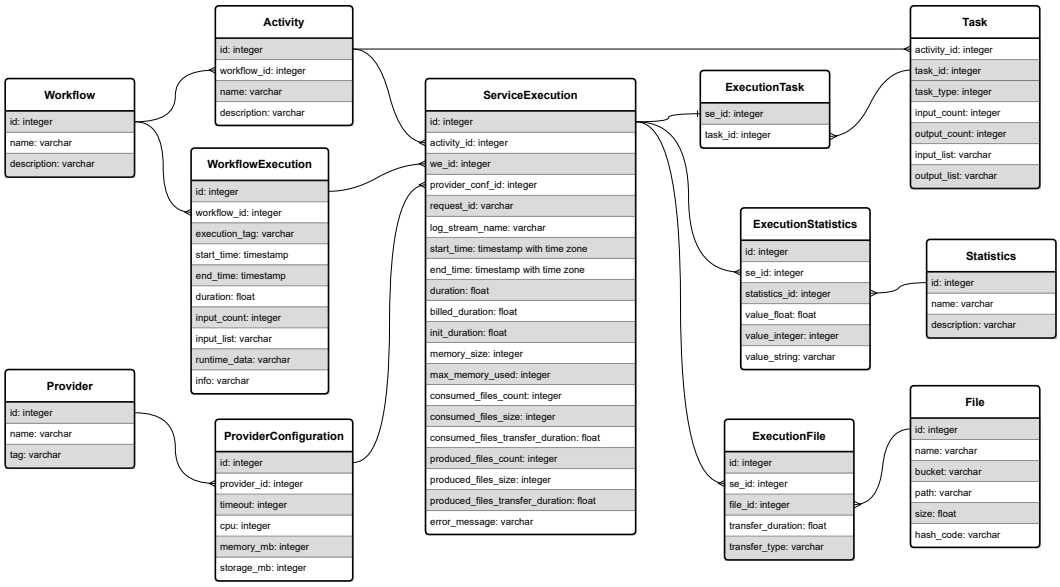

Figure 4. The schema of the *Provenance Database* in DENETHOR.

Once the *Provenance Database* is populated, the user can interact with the approach via the API (step ❾). The API offers a set of operations, each mapped to a query submitted to the *Provenance Database*. Although SQL supports a wide range of queries, DENETHOR focuses on three types: (i) List, (ii) Statistical, and (iii) Provenance Graph. The first two are handled by directly querying the database and returning the results to the user (step ❿). For example, the operation `getAverageExecutionTime(function,[constraints])` returns the average execution time of a specified function, filtered by user-defined constraints (*e.g.*, time window, minimum duration). When a provenance graph is requested,

the API invokes the *Provenance Exporter* (step 11). This component queries the database, converts the results into PROV-O format, and generates visualizations as presented in Figure 5. The `DENETHOR` code is open-source and available at `https://github.com/UFFeScience/denethor`.

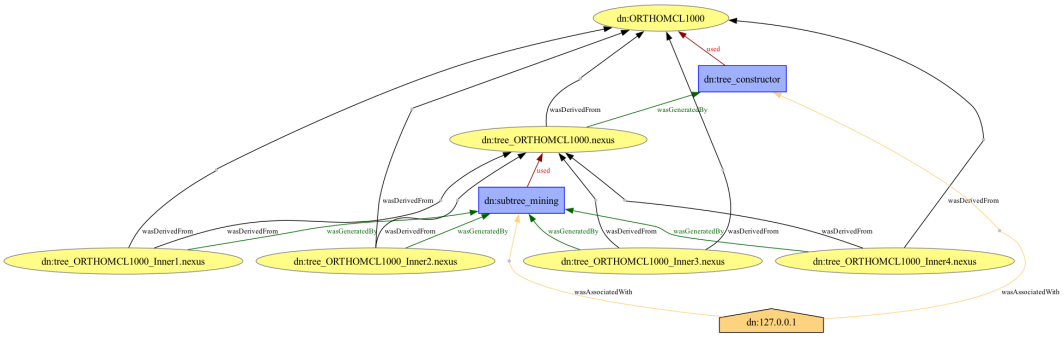

Figure 5. Visualization of the provenance graph generated by `DENETHOR`.

5. Experimental Evaluation

To evaluate the proposed approach, we conducted an experiment using a real-world CSE application and analyzed its dataflow based on the collected provenance. The following subsections describe the application, experimental setup, and the results of our analysis.

5.1. Use Case: Frequent Subtree Mining in Phylogenetic Databases

In phylogeny, a common task is the generation of large phylogenetic trees [Goloboff et al. 2009], which are essential for understanding evolutionary relationships among organisms. Various tools and functions generate such trees using different methods, *e.g.*, maximum parsimony, or maximum likelihood. Since each method has unique characteristics, the same input data (*i.e.*, DNA, RNA, or amino acid sequences) may produce different trees depending on the method used [Puigbò et al. 2019]. Selecting the most appropriate method is often non-trivial, so users typically apply multiple methods, generating several trees. Extracting meaningful insights from these trees then becomes necessary. One option is to identify frequent subtrees within the tree database. However, this task is NP-hard [Amir and Keselman 1997] and may require comparing hundreds of trees. Due to its computational complexity, frequent subtree mining qualifies as a CSE application, and we selected it as our use case. The application follows a well-defined dataflow consisting of eight activities, as shown in Figure 6.

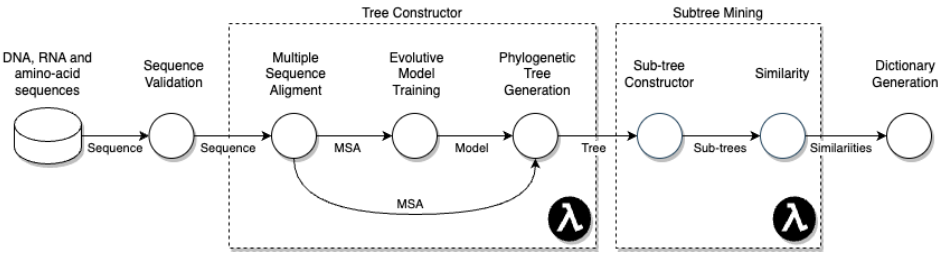

Figure 6. The dataflow for frequent subtree mining in phylogenetic databases.

The workflow begins with *Sequence Validation*, using the Biopython library[3] to check if the input files are valid. Next, *Multiple Sequence Alignment* is performed using the ClustalW tool[4], and the resulting alignment is stored for subsequent steps. *Evolutionary Model Selection* defines the Neighbor-Joining (NJ) model, a method for constructing phylogenetic trees based on evolutionary distance data. In the *Phylogenetic Tree Generation* step, Biopython is used to construct the tree. Once the trees are generated, the frequent subtree mining process begins. The first step, *Subtree Constructor*, extracts all possible subtrees from each phylogenetic tree.

Once all possible subtrees are extracted, the *Similarity Calculation* computes a similarity index for each pair of subtrees, with values ranging from 0 to 100, where higher values indicate greater similarity. This step outputs a database containing similarity scores between subtree pairs, and it is highly parallelizable, as the comparison of a pair of subtrees is independent of the others (*i.e.*, bag-of-tasks parallelism). Finally, the *Dictionary Generation* activity exports this database for further usage. This workflow invokes two serverless functions in the current version, as shown in Figure 6. The *Multiple Sequence Alignment*, *Evolutionary Model Selection*, and *Phylogenetic Tree Generation* steps were grouped into one AWS Lambda function named `tree_constructor`, while the *Subtree Constructor* and *Similarity Calculation* steps were encapsulated in another Lambda function called `subtree_mining`.

5.2. Experiment and Environment Setup

In the evaluation, we used ClustalW version 2.1 for sequence alignment, Biopython 1.81 for phylogenetic tree generation, and DendroPy 4.6.1 to read phylogenetic data in *.nexus* format. The input dataset consisted of 100 multi-FASTA files containing protein sequences from orthologous genes of protozoan species. All experiments were conducted on the Amazon AWS cloud platform (`https://aws.amazon.com/pt/`), using two Lambda functions: `tree_constructor` and `subtree_mining`, each configured with specific resource settings. The `tree_constructor` function was allocated 128 MB of memory, 512 MB of ephemeral storage, and a 15-second timeout. On the other hand, the `subtree_mining` function was configured with 256 MB of memory, 512 MB of ephemeral storage, and a 900-second (15-minute) timeout. This extended timeout is required due to the compute-intensive nature of subtree generation, comparison, and similarity calculation performed by this function. As future work, outside the scope of this paper, we plan to refactor the `subtree_mining` step into smaller, parallelizable functions.

5.3. Results

Multiple executions of the application were monitored by `DENETHOR`, with small variations in parameters and input datasets—such as dataset size—to analyze the dataflow, understand execution behavior, and detect potential issues, *e.g.*, function timeouts. Table 1 presents a set of provenance queries derived from those most frequently submitted by domain experts and specified in the Provenance Challenge [Moreau et al. 2008]. Inspired by the evaluation performed in [Pina et al. 2025], these queries are grouped into five categories: (i) C1 – retrieve entity attributes from a single provenance graph; (ii) C2 – retrieve activity-related information from a single provenance graph; (iii) C3 – access data from multiple provenance graphs; (iv) C4 – perform statistical analysis, typically using aggregation functions; and (v) C5 –

[3] `https://biopython.org/`

[4] `https://www.genome.jp/tools-bin/clustalw`

relate entity attributes to data derivation paths. While cloud providers like AWS claim to support provenance for serverless functions via logs, extracting such information is tedious, error-prone, and ill-suited for aggregation queries like Q4. Performing these tasks through raw logs requires considerable effort from the user.

Table 1. Provenance queries supported by DENETHOR classified by class.

Query ID	Description	Class
Q1	What are the produced files by `subtree_mining` for input file `ORTHOMCL1000`?	C1
Q2	What is the execution time of `tree_constructor` for a specific input file `ORTHOMCL1000`?	C2
Q3	Retrieve the combinations of function parameters for the last 3 executions of the CSE application.	C3
Q4	What is the average execution time of `tree_constructor` over the last 3 months?	C4
Q5	What is the data derivation path of file `ORTHOMCL1000`?	C5

To evaluate the capability of DENETHOR in analyzing the dataflow of a CSE application, we submitted queries Q1 through Q5, using SQL, to a provenance database populated with data from 100 executions of the application, as mentioned earlier. The outcomes of these queries are detailed in Tables 2, 3, 4, 5, and 6. Focusing first on the results of Query Q1 (Table 2), this query enables the exploration of the contents of files derived from a specific input file throughout the application's execution. This kind of analysis is particularly important in scientific experiments where researchers often need to examine the content of individual files, which typically contain domain-specific data. Tracing and identifying the exact locations where such derived files are stored when auditing a distributed application is crucial. This capability allows users to gain deeper insights into the behavior and output of the application, helping them to evaluate whether the observed results align with expected outcomes. If discrepancies are found, this information can guide necessary interventions or modifications to the application. It is worth mentioning that Query Q1 demonstrates good performance, with an average execution time of just 121.0 milliseconds.

Table 2. Results for Query Q1.

File Name	Size	Bucket	Creation Date
tree_ORTHOMCL1000_Inner1.nexus	187	denethor-bucket-output-subtree	2024-02-15 00:07:24.383+00
tree_ORTHOMCL1000_Inner2.nexus	243	denethor-bucket-output-subtree	2024-02-15 00:07:26.198+00
tree_ORTHOMCL1000_Inner3.nexus	191	denethor-bucket-output-subtree	2024-02-15 00:07:29.775+00
tree_ORTHOMCL1000_Inner4.nexus	405	denethor-bucket-output-subtree	2024-02-15 00:07:33.986+00

The results obtained from Query Q2 (Table 3) provide insights into the performance of individual function executions. Specifically, this query allows users to check whether a particular invocation of a serverless function is consuming more time than expected. Such analysis is key to identifying abnormal execution patterns that may represent potential inefficiencies or underlying issues within the CSE application. This type of investigation is especially relevant since each function invocation incurs a cost based on execution time and resource usage. By pinpointing outlier executions that deviate from expected performance, users can proactively optimize workflows and avoid unnecessary financial costs. Regarding performance, Query Q2 presented a good response time, with an average execution time of 247.0 milliseconds.

The results derived from Query Q3 (Table 4) empower users to examine whether specific combinations of function parameter values have an impact on the performance of the CSE application. This type of analysis is important, as serverless functions can behave

Table 3. Results for Query Q2.

Input File	Execution Time (ms)	Number of Errors
ORTHOMCL1000	4032.6	0

differently depending on the input parameters provided during execution. Certain configurations may lead to optimal performance, with functions completing their tasks rapidly, while others might cause significant delays or even result in execution failure due to timeout constraints. Users can better understand how parameter choices affect the function's behavior by analyzing these relationships. This insight enables them to fine-tune their experiments, avoid suboptimal configurations, and enhance the efficiency and reliability of the application. Query Q3 presented an efficient average execution time of 142.0 milliseconds.

Table 4. Results for Query Q3.

Function Name	Provider Name	Memory (MB)	Timeout (s)	CPU	Storage (MB)
tree_constructor	AWS Lambda	128	15	1	512
tree_constructor	AWS Lambda	128	15	1	512
tree_constructor	AWS Lambda	128	15	1	512
subtree_mining	AWS Lambda	256	900	1	512
subtree_mining	AWS Lambda	256	900	1	512
subtree_mining	AWS Lambda	256	900	1	512

The results obtained from Query Q4 (Table 5) enable users to estimate the average time required for executing the CSE application, which is interesting for planning and resource allocation. In contrast to the previous queries, Q4 has an aggregation function within its SQL formulation to compute average execution times across multiple executions. This type of analysis plays a critical role in performance profiling, particularly when the user cannot yet estimate the application's runtime. Such an estimate is essential for projecting the total makespan and calculating the associated financial costs, since billing is based on execution time and resource consumption. Query Q4 demonstrated an average execution time of 350.0 milliseconds, reflecting `DENETHOR` 's capability to handle statistical provenance queries that involve aggregation across multiple executions.

Table 5. Results for Query Q4.

Activity Name	Average Execution Time (ms)
tree_constructor	1784.1

Finally, the results derived from Query Q5 (Table 6) presents a representative subset of the results, providing the user with a detailed view of all transformations applied to a specific file throughout the execution of the CSE application, including both input and intermediate files. This level of inspection allows for the analysis of the file's data derivation path. By examining each transformation step, the user can evaluate whether each step in the dataflow is executing as expected or if additional steps are needed to enhance the quality or reliability of the results. Beyond performance and optimization, this query also reveals the activities that led to a particular result. Such insight is fundamental for scientific transparency and reproducibility. In the context of reproducibility, the user may choose to receive the query output in a structured tabular format (as shown in Table 6) or as a PROV-N document (Figure 7), which corresponds to the visual provenance graph illustrated in Figure 5. On average, Query Q5 was executed in 199.9 milliseconds.

Table 6. A subset of the results for Query Q5.

Consumed File	Produced File	Function Name	Produced Time
tree_ORTHOMCL1000.nexus	tree_ORTHOMCL1000_Inner1.nexus	subtree_mining	2024-02-15 00:07:24.383+00
tree_ORTHOMCL1000.nexus	tree_ORTHOMCL1000_Inner2.nexus	subtree_mining	2024-02-15 00:07:26.198+00
tree_ORTHOMCL1000.nexus	tree_ORTHOMCL1000_Inner3.nexus	subtree_mining	2024-02-15 00:07:29.775+00
tree_ORTHOMCL1000.nexus	tree_ORTHOMCL1000_Inner4.nexus	subtree_mining	2024-02-15 00:07:33.986+00

```
document
 prefix dn <http://denethor.ic.uff.br/>
 entity(dn:ORTHOMCL1000)
 entity(dn:tree_ORTHOMCL1000.nexus)
 entity(dn:tree_ORTHOMCL1000_Inner1.nexus)
 entity(dn:tree_ORTHOMCL1000_Inner2.nexus)
 entity(dn:tree_ORTHOMCL1000_Inner3.nexus)
 entity(dn:tree_ORTHOMCL1000_Inner4.nexus)
 agent(dn:127.0.0.1)
 activity(dn:tree_constructor, -, -)
 activity(dn:subtree_mining, -, -)
 used(dn:tree_constructor, dn:ORTHOMCL1000, -)
 wasGeneratedBy(dn:tree_ORTHOMCL1000.nexus, dn:tree_constructor, -)
 used(dn:subtree_mining, dn:tree_ORTHOMCL1000.nexus, -)
 wasGeneratedBy(dn:tree_ORTHOMCL1000_Inner1.nexus, dn:subtree_mining, -)
 wasGeneratedBy(dn:tree_ORTHOMCL1000_Inner2.nexus, dn:subtree_mining, -)
 wasGeneratedBy(dn:tree_ORTHOMCL1000_Inner3.nexus, dn:subtree_mining, -)
 wasGeneratedBy(dn:tree_ORTHOMCL1000_Inner4.nexus, dn:subtree_mining, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000.nexus, dn:ORTHOMCL1000, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner1.nexus, dn:ORTHOMCL1000, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner2.nexus, dn:ORTHOMCL1000, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner3.nexus, dn:ORTHOMCL1000, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner4.nexus, dn:ORTHOMCL1000, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner1.nexus, dn:tree_ORTHOMCL1000.nexus, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner2.nexus, dn:tree_ORTHOMCL1000.nexus, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner3.nexus, dn:tree_ORTHOMCL1000.nexus, -, -, -)
 wasDerivedFrom(dn:tree_ORTHOMCL1000_Inner4.nexus, dn:tree_ORTHOMCL1000.nexus, -, -, -)
 wasAssociatedWith(dn:127.0.0.1, dn:tree_constructor, -)
 wasAssociatedWith(dn:127.0.0.1, dn:subtree_mining, -)
endDocument
```

Figure 7. The PROV-N document generated by the *Provenance Exporter*.

6. Conclusions

The approach presented in this paper, named DENETHOR, is designed to ease the dataflow analysis of CSE applications that invoke serverless functions, such as AWS Lambda. Capturing provenance in serverless environments remains an unresolved challenge, mainly due to the lack of built-in support for provenance tracking in such infrastructures. DENETHOR addresses this gap by providing data analytics capabilities within a lightweight, modular architecture, allowing the CSE application to operate independently while enabling or disabling provenance tracking as needed. There are several key benefits to decoupling the dataflow analysis system from both the CSE application and the underlying serverless platform: (i) Provenance capture and storage are performed without imposing any performance overhead on the application's execution; (ii) The relationships between consumed and produced data files are automatically established during the CSE application's execution, eliminating the need for error-prone *post-mortem* analysis; (iii) Compliance with the W3C PROV recommendation ensures interoperability with other provenance tools; and (iv) It enables fine-grained root cause analysis of workflow errors by allowing queries on the W3C PROV-compliant *Provenance Database*. In addition to its modularity, DENETHOR offers flexibility by allowing users to define the specific metadata they wish to capture. Real-world experimentation with a CSE application in the Bioinformatics domain has demonstrated the practical value of provenance-based dataflow analysis in serverless environments. As part of future work, we aim to extend DENETHOR's capabilities by integrating it with other PROV-compliant provenance systems, generating a unified provenance graph that spans local, HPC, and serverless executions.

References

Amir, A. and Keselman, D. (1997). Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM J. on Comp.*, 26(6):1656–1669.

Barham, P. et al. (2004). Using magpie for request extraction and workload modelling. In *OSDI'04*, San Francisco, CA. USENIX Association.

Ben-Shimol, L. et al. (2025). Detection of compromised functions in a serverless cloud environment. *Computers & Security*, 150:104261.

Bux, M. et al. (2015). SAASFEE: scalable scientific workflow execution engine. *Proc. VLDB Endow.*, 8(12):1892–1895.

Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. (2004). Dynamic instrumentation of production systems. In *USENIX ATC'04*, Boston, MA. USENIX.

Datta, P., Polinsky, I., Inam, M. A., Bates, A., and Enck, W. (2022). Alastor: Reconstructing the provenance of serverless intrusions. In *USENIX Security Symposium*.

de Oliveira, D., Liu, J., and Pacitti, E. (2019). *Data-Intensive Workflow Management: For Clouds and Data-Intensive and Scalable Computing Environments*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Elshamy, A., Alquraan, A., and Al-Kiswany, S. (2023). A study of orchestration approaches for scientific workflows in serverless computing. SESAME '23, page 34–40, New York, NY, USA. ACM.

Freire, J., Koop, D., Santos, E., and Silva, C. T. (2008). Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21.

Goloboff, P. A. et al. (2009). Phylogenetic analysis of 73 060 taxa corroborates major eukaryotic groups. *Cladistics*, 25(3):211–230.

Guerra, G. et al. (2012). Uncertainty quantification in computational predictive models for fluid dynamics using a workflow management engine. *Int. J. for Uncert. Quant.*, 2(1):53–71.

Hautz, M., Ristov, S., and Felderer, M. (2023). Characterizing afcl serverless scientific workflows in federated faas. WoSC '23, page 24–29, NY, USA. ACM.

Hellerstein, J. M. et al. (2019). Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org.

Herschel, M., Diestelkämper, R., and Ben Lahmar, H. (2017). A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26.

Huang, J. et al. (2024). Faasrca: Full lifecycle root cause analysis for serverless applications. In *ISSRE'24*, pages 415–426. IEEE.

Kamble, S., Jin, X., Niu, N., and Simon, M. (2017). A novel coupling pattern in computational science and engineering software. In *Proceedings of the 12th International Workshop on Software Engineering for Science*, SE4Science '17, page 9–12. IEEE Press.

Khochare, A., Simmhan, Y., Mehta, S., and Agarwal, A. (2022). Toward scientific workflows in a serverless world. In *2022 IEEE e-Science*, pages 399–400.

Kiar, G. et al. (2019). A serverless tool for platform agnostic computational experiment management. *Frontiers in Neuroinformatics*, 13.

Mattoso, M., Werner, C., Travassos, G. H., Braganholo, V., Ogasawara, E., Oliveira, D., Cruz, S., Martinho, W., and Murta, L. (2010). Towards supporting the life cycle of large scale scientific experiments. *International Journal of Business Process Integration and Management*, 5(1):79.

Moreau, L. et al. (2008). Special issue: The first provenance challenge. *Concurrency and Computation: Practice and Experience*, 20(5):409–418.

Moreau, L. and Groth, P. (2013). Provenance: an introduction to prov. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 3(4):1–129.

Neves, V. C., de Oliveira, D., Ocaña, K. A. C. S., Braganholo, V., and Murta, L. (2017). Managing provenance of implicit data flows in scientific experiments. *ACM Trans. Internet Techn.*, 17(4):36:1–36:22.

Ocaña, K. and de Oliveira, D. (2015). Parallel computing in genomic research: advances and applications. *Adv. Appl. Bioinform. Chem.*, 8:23–35.

Pimentel, J. F. et al. (2017). noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *VLDB*, 10(12).

Pina, D., Kunstmann, L., Chapman, A., de Oliveira, D., and Mattoso, M. (2025). DLProv: a suite of provenance services for deep learning workflow analyses. *PeerJ Comput. Sci.*, 11(e2985):e2985.

Puigbò, P. et al. (2019). Genome-wide comparative analysis of phylogenetic trees: The prokaryotic forest of life. In *Evolutionary Genomics: Statistical and Computational Methods*, pages 241–269. Springer New York, New York, NY.

Rude, U., Willcox, K., McInnes, L. C., and Sterck, H. D. (2018). Research and education in computational science and engineering. *Siam Review*, 60(3):707–754.

Satapathy, U., Thakur, R., Chattopadhay, S., and Chakraborty, S. (2023). Disprotrack: Distributed provenance tracking over serverless applications. In *INFOCOM 2023*, pages 1–10.

Silva, V., de Oliveira, D., Valduriez, P., and Mattoso, M. (2018). Dfanalyzer: Runtime dataflow analysis of scientific applications using provenance. *Proceedings of the VLDB Endowment*.

Skluzacek, T. J. et al. (2019). Serverless workflows for indexing large scientific data. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 43–48.

Wen, J., Chen, Z., Zhao, J., Sarro, F., Ping, H., Zhang, Y., Wang, S., and Liu, X. (2025). Scope: Performance testing for serverless computing. *ACM Transactions on Software Engineering and Methodology*.

Wen, J. et al. (2021). An empirical study on challenges of application development in serverless computing. In *Proc. of the ESEC/FSE 2023*, pages 416–428.