Scientific Workflows Deployment: Container Provenance in High-Performance Computing

Liliane Kunstmann[1], Débora Pina[1], Daniel de Oliveira[2], Marta Mattoso[1]

[1]PESC/COPPE – Universidade Federal do Rio de Janeiro (COPPE/UFRJ)

[2]Instituto de Computação – Universidade Federal Fluminense (IC/UFF)

{lneves, dbpina, marta}@cos.ufrj.br, danielcmo@ic.uff.br

Abstract. *Deploying scientific workflows in high-performance computing (HPC) environments is increasingly challenging due to diverse computational settings. Containers help deploy and reproduce workflows, but both require more than just accessing container images. Container provenance provides essential information about image usage, origins, and recipes, crucial for deployment on various architectures or engines. Current support is limited to container actions and processes without workflow traceability. We propose extending workflow provenance to include container data using `ProvDeploy`, which supports workflow deployment with various container compositions in HPC, using W3C-PROV for container representation. We evaluated this with a real scientific machine learning workflow.*

1. Introduction

Scientific workflows are chained activities with a data flow in between [de Oliveira et al. 2019]. The workflow activities comprise multiple programs, software libraries, and software dependencies [Orzechowski et al. 2018]. These workflows are usually deployed in various environments due to their resource requirements and scientific nature [Orzechowski et al. 2018]. Their design and initial development are performed on personal computers and later are migrated to High Performance Computing (HPC) environments where they can be properly executed. So, their execution demands a deployment process to be performed multiple times in heterogeneous environments. Current machine learning experiments share this same profile of scientific workflow deployment.

Containerization can assist in deploying applications in multiple and heterogeneous environments [Merkel et al. 2014]. Containers are a lightweight form of virtualization that packages an application and its software dependencies into a single, self-contained unit [Merkel et al. 2014]. This executable unit is called a container image. Using containers to deploy scientific workflows requires a careful containerization design [Kunstmann et al. 2024]. This design includes deciding which container images to use for the workflow activities and how to group these images in containers, known as container composition. There are three alternatives to compose the workflow container. In the fine-grained composition, each workflow activity is containerized in independent containers. The other extreme is the coarse-grained composition that uses a single container to group all workflow activities. In the hybrid composition, workflow activities are grouped in a few containers to avoid one single container or managing the many containers of the fine-grained. This containerization design is implicit and yet needs to be known for reproducibility. Provenance data is often associated with workflow execution to add trust and reproducibility [Costa et al. 2013]. The workflow provenance is not aware of the containerization design. It is important to make a workflow provenance container-aware to represent how the container images are chosen and grouped to workflow activities, with the necessary ports to access those images, *etc*.

Current approaches to container provenance focus on capturing I/O operations and tracking container behavior through low level function calls. They do not relate container provenance to workflow provenance traceability. Analyzing these provenance data is further complicated because related work does not comply with standard representations like W3C PROV [Moreau and Groth 2013] and the Open Container Initiative (OCI) annotations[1].

We present a workflow provenance approach that is container-aware to represent globally the traceability of the workflow execution. We argue that both workflow and container provenance are essential for the analysis and reproducibility of the workflow, as they are complementary. Our approach aims at providing workflow analysis like "*which container images were used for a specific activity of the workflow*", and traceability on the base images of the workflow execution path.

We use `ProvDeploy`[2] [Kunstmann et al. 2022] to evaluate our container-aware workflow provenance approach. `ProvDeploy` is a framework that deploys containerized workflows in HPC environments, supporting different container compositions. We contribute with a provenance data model based on the W3C PROV and the OCI annotations to represent container descriptions, requirements, recipes, and files [Campagna et al. 2020, Paranhos et al. 2023]. We explore our container-aware workflow provenance with analytical queries submitted to the execution of DenseED [Freitas et al. 2021], a real scientific machine learning workflow. We show how the provenance of multiple containers can be used to help analyze and reproduce the workflow.

The remainder of this paper is structured as follows. Section 2 presents container provenance related work. Section 3 presents `ProvDeploy` extended with the proposed provenance data model. Section 4 shows the use of provenance through `ProvDeploy` to evaluate different container compositions with DenseED in an HPC environment, and Section 5 concludes this paper.

2. Workflow Containerization and Current Provenance Support

This section presents challenges in containerizing workflows and container provenance. Current approaches for container provenance are limited to representing container actions and processes without workflow traceability. We did not find related work for workflow provenance traceability that is container-aware. Capturing and relating container with workflow provenance for analysis is an open problem, particularly in HPC.

Provenance support for workflows is not new [Silva et al. 2020], however, many solutions that claim to support provenance, do not represent the typical relationships that define the derivation paths for traceability [Pina et al. 2024] or cannot capture provenance in HPC. PROV-IO$^+$[Han et al. 2024] is an exception designed to capture workflow provenance for HPC environments. It shares similarities with `ProvDeploy` like using an extensible W3C PROV-compliant data model and providing relationships through prospective provenance

[1]`https://opencontainers.org/`
[2]`https://bitbucket.org/lilianeKunstmann/provdeploy/`

(*p-prov*), where steps of the workflow are represented before execution like a recipe to be followed, in addition to retrospective provenance (*r-prov*), which is captured during execution [Freire et al. 2008]. Despite a rich workflow provenance, PROV-IO$^+$ is unaware of container provenance.

Containerizing applications is almost straightforward, whereas composing workflow activities into containers is challenging [Lampa et al. 2019, Novella et al. 2019]. The coarse-grained composition helps deployment, however, replacing and reusing workflow activities is easier with a fine-grained composition. Reproducing a workflow execution without container composition awareness can be challenging. In HPC workflows there are often restrictions on image building [Priedhorsky et al. 2021]. Containers are not isolated like virtual machines, they rely on the host OS to execute their isolated processes and can be affected by other containers, their configurations [Straesser et al. 2023], and the execution environment [Straesser et al. 2023, Wofford et al. 2022].

All those issues increase workflow containerization challenges. Like scientific workflows, ML often adopts containers through ML studios and other cloud hosted services that provide access to computing resources. In the ML context, users face challenges in container compositions and limited provenance support [Schlegel and Sattler 2023, Pina et al. 2024].

Capturing container provenance is addressed by a few approaches, with variations in what data is collected and how it is stored, depending on their goals. Most of these approaches [Shaffer et al. 2023, Chen et al. 2021, Abbas et al. 2022, Ahmad et al. 2020, Han et al. 2024] automatically collect metadata from containers of single applications or microservices. However, this metadata does not relate the application artifacts, has low-level traces, lacks workflow support, and, is available only for *post-mortem* analysis, *i.e.* only after execution. Our analysis finds that current related work limits workflow traceability analyses and reproducibility of container images.

We discuss and compare some of these approaches with `ProvDeploy` in Table 1. The column *Container & Workflow* specifies if the provenance captured can represent container provenance related to workflow. *Provenance Awareness level* details the level of container provenance representation if it represents single applications, microservices, or workflows. *Provenance graph* specifies if the approach allows the derivation of a provenance graph. *Data Model* describes whether provenance is represented following a standard such as W3C PROV or in an *ad-hoc* way. *Access Availability* indicates whether the provenance data are available for analysis at runtime or *post-mortem*. *Query Support* indicates the query support for analyzing provenance data.

Chen et al. (2021) discuss the challenges of sound and clear provenance tracking for microservices proposing `CLARION`, a *namespace* and container-aware provenance tracking solution. Similarly, `ALASTOR` enables tracking of suspicious events in serverless applications. `PACED` [Abbas et al. 2022] is designed to detect container escape attacks through the isolation of cross-*namespace* events. Satapaphy et al. (2023) discuss the lack of provenance data capture for microservice applications and propose DisProTrack[Satapathy et al. 2023] for capturing provenance from microservices in an integrated way, handling parallel calls inherent to microservices.

Modi et al. (2023) discuss the challenges of capturing container provenance for

[3]Universal Provenance Graph

Table 1. Comparison of provenance support for containerized workflows.

Container Provenance Solution	Container & Workflow	Provenance Awareness Level	Provenance Graph	Data Model	Access Availability	Query Support
`CLARION` [Chen et al. 2021]	No	Microservice	No	N/A	*Post-mortem*	N/A
`ALASTOR` [Datta et al. 2022]	No	Microservice	Yes	*Ad-hoc*	*Post-mortem*	Provenance graph
`PACED` [Abbas et al. 2022]	No	Microservice	No	N/A	Runtime	N/A
`DisProTrack` [Satapathy et al. 2023]	No	Microservice	Yes	UPG [3]	*Post-mortem*	RegEx
[Modi et al. 2023]	No	Single Application	Yes	*Ad-hoc*	*Post-mortem*	Hypergraph
[Wofford et al. 2022]	No	Single Application	Yes	*Ad-hoc*	Runtime	SQLite
`PROV-CRT` [Ahmad et al. 2020]	No	Single Application	Yes	W3C PROV	Runtime	N/A
[Olaya et al. 2022]	Yes	Workflow	No	*Ad-hoc*	*Post-mortem*	Jupyter Interface
`ProvDeploy` extended	Yes	Workflow	Yes	W3C PROV OCI	Runtime	MonetDB

standalone applications using container namespaces. They examine *post-mortem* container provenance from auditing tools like PROV-CRT and introduce a hypergraph-based model for tracking provenance in single containerized applications. Their model is limited to *post-mortem* analysis of single applications.

Wofford et al. (2022) propose the definition of requirements for capturing the provenance of HPC applications and the issues related to hardware metadata capture. They propose the design and implementation of a container-based provenance capture system that is limited to a single application.

`PROV-CRT` [Ahmad et al. 2020] is a provenance module integrated into container runtimes such as LXC and Docker. It tracks and audits provenance during container creation and execution. `PROV-CRT` captures provenance at the granularity of system calls, which, although difficult to analyze, enables verification and validation of computations during container replay by comparing audited provenance data.

Olaya et al. (2022) propose a tool that automatically generates a record trail for each data container of the workflow. They represent this record trail inside each data container separately. Their approach provides a Jupyter interface to process these data containers to join the traces into a workflow graph. There is no independent provenance graph to be traced by third-party tools. Since their provenance is attached to Singularity containers, they rely on Singularity/Apptainer compatibility and availability. Their record trail is not based on W3C PROV relationships, which forces the user to learn a new representation, and it is only available for *post-mortem* analysis that also relies on data container availability.

Our approach, implemented in `ProvDeploy` is inline with, Wofford et al. (2022) and Canon (2020) highlighting the importance of documenting container characteristics for analysis, explanation, and reproducibility, given that containers can employ different drivers to execute the same task. `ProvDeploy` [Kunstmann et al. 2022, Kunstmann et al. 2024] is a framework that eases the deployment of scientific workflows in HPC environments with integrated provenance capture. ProvDeploy was first developed as a framework to support provenance data capture in scientific applications, as it allowed using provenance services through containers. In its initial version, ProvDeploy included a catalog that stored metadata on container images available for execution. This catalog was later expanded to provide more detailed information about the container images. When we extended ProvDeploy to execute workflows, we identified the need for container provenance that also registers the workflow. To the best of our knowledge, Olaya et al. (2022) and ProvDeploy are the only approaches that offer container provenance at the workflow level.

3. Container-aware Provenance with ProvDeploy

In this section, we present a comprehensive workflow provenance data model implemented in ProvDeploy to become container-aware. This model extends the previous provenance data of ProvDeploy with container provenance. Using the data model presented in Figure 1, we aim to represent container provenance and provide meaningful analysis with workflow provenance without tying the user to a specific provenance service. This data model is based on OCI and Common HPC Container Conformance Initiative[4], and relevant data presented in [Priedhorsky et al. 2021, Canon 2020, Straesser et al. 2023, Gruening et al. 2018, Wofford et al. 2022].

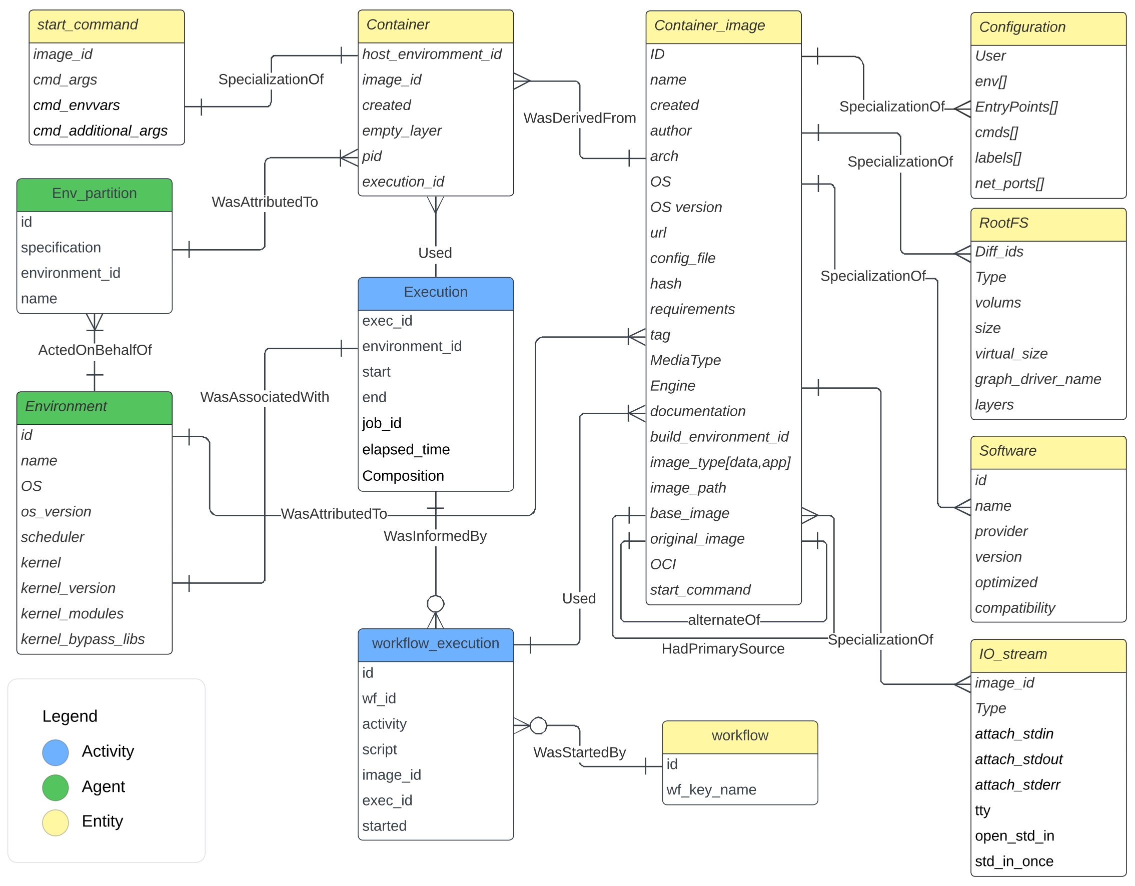

Figure 1. PROV-DM diagram of container aware provenance data model.

[4]https://github.com/container-in-hpc/container-hpc-conformance

The model presented in Figure 1 follows the W3C PROV-DM recommendation, where provenance is represented in terms of entities (the data objects), activities (data transformations), and agents (users and systems) with their W3C PROV-DM relationships. In the proposed model, the yellow classes represent Entities, the green classes represent Agents, and the blue classes represent Activities. The colors of our class diagram are represented as stereotypes in compliance with the W3C PROV classes. This data model aims to provide information about container image origins and features, improving its findability and reliability for the rebuild process. `ProvDeploy` implements this data model using MonetDB, a column-oriented RDBMS, that as a DBMS solution, allows for querying provenance graphs and easy integration with other libraries for analysis and visualization.

The entity that allows the connection between container and workflow provenance is the entity *workflow* that has as an attribute workflow name (*wf_key_name*) that identifies it on the provenance database of the provenance capture service (*e.g. dataflow_tag* in DfAnalyzer [Silva et al. 2020] or *hashID* in noWorkflow [Murta et al. 2015]). The activity *execution* associates the container composition, which can be coarse-grained (one container for the whole workflow), fine-grained (one container for each workflow activity), or hybrid (multiple, but not all, activities executed by the same container). This entity also stores *job_id*, from schedulers in HPC environments, which can be used later for tuning and debugging.

The entity *container_image* represents the container image details, enabling it to be rebuilt. A container image is specialized by the entity *configuration*, which includes lists of *entrypoints*, *labels*, environment variables (*env*), network information (*networks*), required volumes (*rootFS*), and commands (*cmd*). This information is also stored in the container image configuration file or recipe (*config_file*), and in the container image. Still, there is no guarantee that recompiling the container image using the configuration file will produce an identical image [Canon 2020], or that the configuration file is up-to-date with the image.

Additionally, container images are linked to the build environment (*build_environment_id*) in which they were created. This build environment defines the container's compatible kernel architectures. A container image can be the source for other container images, either as a primary source (*base_image*) for a new container image or by generating alternative versions with different container engines (*original_image*) through container image conversion. For instance, a container image built on Docker might be converted to Singularity. In such cases, it is no longer associated with the *config_file* but references the original container image.

The entity *container* describes the isolated process that runs using a container image and specifies the environment in which it is executed. A container relies on a container image to be executed. One single image can be used to execute multiple containers. This entity is used during *execution* activity and documents what occurred during the workflow execution. Both *execution* and *workflow_execution* activities capture the expected behavior of the containerized workflow, referred to as prospective provenance. The *container* entity records the activities that were carried out, referred to as retrospective provenance. If an activity fails to start, it is not recorded. The entity *container* is specialized by entity *start_command* that stores the commands, arguments, and environment variables used at container start because the user can start a container image with arguments that are different from what is stored in the *container_image*.

The *environment* agent describes the environment associated with an *execution*, including its kernel architecture and scheduler. HPC and cloud environments can have heterogeneous resources, so the *env_partition* was designed to detail the characteristics of the specific environment used. This partition contains the hardware specifications of the environment and is where the containers are executed.

The *execution* activity, along with *workflow_execution* activity, describes the containerized execution. This includes the composition of containers, workflow activities, and the containers they used, along with the expected execution commands and parameters needed to start each activity. These activities enhance user execution management, especially in the case of provenance services, like DfAnalyzer [Silva et al. 2020], that do not natively distinguish multiple concurrent workflow executions.

This model aims to support a comprehensive analysis of the entire workflow execution in a containerized environment. Table 2 showcases a series of queries that the container-aware provenance data model can address when connected to the workflow provenance model. These queries are inspired by the First Provenance Challenge[5] and are categorized into information concerning container reuse, workflow reproducibility, and insights for workflow traceability analysis.

Table 2. Container and Workflow Provenance Queries inspired by the First Provenance Challenge

#	Query	Type
Q1	Retrieve job_id, containers, and host environments involved in a workflow execution.	Reuse Reproducibility
Q2	Retrieve the container images associated with the workflow with the *job_id* and activities they performed.	Reuse Reproducibility
Q3	Retrieve all workflows that were executed with a given container image.	Reuse
Q4	Retrieve all workflow activities that were run with a given container image	Reuse
Q5	Retrieve all images that are created by a specific user.	Reuse
Q6	Retrieve all workflows where a given variable satisfied a certain condition and was deployed with a specific composition.	Analysis Reproducibility
Q7	A user has run the workflow with different compositions and in distinct environments, increasing the number of containers. Retrieve the differences in the compositions in the distinct environments	Analysis
Q8	Which environment (host and container) executed the workflow with its best results?	Analysis Reproducibility

Query Q1 retrieves data that is usually available only during execution, most importantly, the register of the containers that were effectively used for workflow execution, since registering only the images provides limited information from execution. This query can be answered by joining entities *execution, workflow_execution, container_image* and *env_partition* and setting the target workflow.

Query Q2 presents information about container images used for workflow execution, and that can help users in future workflow executions with a similar software stack. Using the associated *job_id* we can also find more useful information about the batch job

[5]https://openprovenance.org/provenance-challenge/FirstProvenanceChallenge.html

executed, such as possible tasks that were executed before or after the workflow and are not modeled by workflow provenance. Query Q2 joins entities *workflow, workflow_execution, container_image* and *workflow*, setting the target workflow using attribute *wf_key_name*.

Query Q3 shows the possibilities of reusing container images across multiple workflows with common software requirements, and Q4 operates similarly to Q3 but provides more detailed information on the activities executed by the container images. Query Q4 joins *container_image, workflow*, and *workflow_execution* entities.

Query Q5 explores the *container_image* entity attributes. This query helps to identify shared base images and execution architectures (*arch*). If a user plans to execute any of these container images in a different environment, the first step would be to check if the base image has a compatible version with the target kernel architecture.

Queries Q6, Q7, and Q8, are more complex queries that rely on both container and workflow provenance. These queries allow the user to understand the provenance model of the workflow, which is extended by the container-aware provenance data model. These queries join container entities *execution, workflow_execution* with entities of the workflow provenance model, starting with the entity that identifies the workflow with the same identifier that the entity *workflow* has. For DfAnalyzer's provenance model, this identifier is stored on an entity called *dataflow*. Once joined with the workflow provenance, the user can explore data through an interval starting from the *execution_start* and ending on the added *elapsed_time*. Those queries help to understand the impact of the container composition on overall workflow performance in different environments.

Queries Q1, Q2, and Q7 cannot be answered using only workflow provenance. During the development of a scientific workflow, users often explore multiple algorithms and software libraries, resulting in numerous containers and container images combined with different sources and environments. In current approaches, containers are not linked with their associated container images, environments, libraries, or workflows. This information may be scattered across logs, files, and scripts. Without a predefined structure to represent this information, analyzing it becomes increasingly difficult and may not be possible, even though it is essential for reproducibility.

Queries Q3, Q4, Q5, and Q7 can improve container image reuse and reduce the execution and storage of unnecessary container images to execute a workflow since a container image can be used to execute multiple workflow activities and generate new container images. In addition, with the data model of `ProvDeploy` we can track a container image derivation path to rebuild it.

4. Evaluation of the Container Provenance Model with the DenseED Workflow

In this section, we evaluate the container provenance model to support queries during and after the execution of the DenseED workflow.

4.1. DenseED workflow

DenseED is a scientific Machine Learning (ML) workflow proposed by Freitas et al. (2021), based on a Physics-guided Physics-guided Convolutional Neural Networks (CNN) as defined by Zhu and Zabaras (2018). DenseED uses the Physics-guided CNN as a surrogate model for Reverse Time Migration (RTM) calculations to quantify uncertainties. Solving

RTM equations is a CPU-intensive and time-consuming task. To quantify uncertainties, RTM has to be calculated many times. The architecture, presented in Figure 2, shows convolutional layers and dense blocks in an Encoder-Decoder arrangement to manage high-dimensional inputs and outputs.

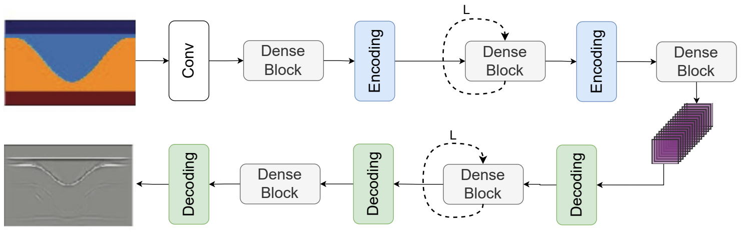

Figure 2. DenseED architecture [Freitas et al. 2021].

DenseED provenance was previously modeled in the provenance database of DfAnalyzer, which captures workflow provenance during execution for runtime hyperparameter tuning [Silva et al. 2020]. This provenance includes hyperparameters and training/test metrics like R^2 and RMSE. Specifically, K determines the DenseED growth rate, and L the number of layers in the dense block, both stored by the model. The system relies on TensorFlow, which requires specific compatibility with Python, the C compiler, Bazel, CUDA, and cuDNN when using GPUs. TensorFlow is available through containers, simplifying this process by enabling GPU usage with specific flags. Users can explore alternative container images from public registries like NGC (NVIDIA GPU Cloud), Docker, and Binder if a workflow's container image is incompatible with the available GPU or the kernel architecture.

Additionally, containers ease the exploration of heterogeneous CPU hardware with minimal effort. Public registries like NGC provide users with optimized TensorFlow container images for different NVIDIA GPU series, allowing users to explore multiple TensorFlow versions and features on different GPU devices. This exploration is not simple, and container provenance helps to identify the best version for reusing the container images, in addition to tracking performance changes. To ensure the reproducibility of these executions over time, container provenance provides relevant information about the original images and, if necessary, helps identify suitable replacement images or rebuild new ones.

4.2. Environment setup

DenseED was deployed with `ProvDeploy` in Santos Dumont (SDumont)[6] Supercomputer. SDumont has an installed processing capacity of around 5.1 Petaflop/s (5.1 x 1015 float-point operations per second), presenting a hybrid configuration of computational nodes, in terms of the available parallel processing architecture. We used two different computational nodes a CPU node and a GPU node. The CPU node has two CPUs with an Intel Xeon E5-2695v2 Ivy Bridge 2.4GHZ processor, 24 cores (12 per CPU), and 64GB DDR3 RAM. The GPU node is part of SDumont expanded partition BullSequana X that has two CPUs with Intel Xeon Skylake 2.1 GHz processor, 48 cores (24 per CPU), 384GB RAM, and four GPUs NVIDIA Volta V100. In both, we used Linux RedHat 7.6 operating system, and Singularity 3.8 for the container engine. We have used a personal computer (CPU Intel Core

[6]`https://sdumont.lncc.br/`

i7-10700K processor, 3.80GHz, 16 cores, and 15,5 GiB) to execute DenseED and generate the container images to be deployed in SDumont. Its operating system is Ubuntu 20.04.6 LTS.

4.3. Analyzing container aware workflow provenance in DenseED

To find the best container composition for DenseED, the workflow was executed with three containerization compositions as shown in Table 3. Querying container provenance helps to choose the best composition and fine-tune containerization decisions according to the target execution environment. `ProvDeploy` supports all the queries listed in Table 2 and this section discusses some of these queries in the context of DenseED.

Table 3. Exploring different containerization compositions with DenseED.

Compositions	Description
Coarse-grained	A single container encompassing all dependencies for provenance data capture, and running DenseED.
Partial modular	Two containers: A container with DenseED and a second container with the DBMS and provenance data, and the provenance service.
Provenance modular	Three containers: the first with DenseED, the second with the provenance service, and the third with the DBMS and provenance data.

Table 4 shows how query Q5 explores container images created by the user ‘Liliane’. In this scenario, Liliane has a database with container images from various workflows. The images tagged *dfanalyzer*, *py-readseq-modelGenerator*, and *java-readseq-modelGenerator* are all based on a Java image that was not created by Liliane and share the amd64 kernel architecture. When reproducing this workflow with any of these container images, it is beneficial to check if the base image has a version compatible with the new target architecture.

Table 4. Q5 - Container images generated by user ‘Liliane’;

id	author	arch	vendor	image tags	description	Env name	image type	Base Image
8	Liliane	amd64	Singularity	denseed	DenseED with DfAnalyzer and MonetDB	liliane-imac20	application	tensorflow
9	Liliane	amd64	Singularity	provData	DfAnalyzer and MonetDB	liliane-imac20	provenance	dfanalyzer
1	Liliane	amd64	Docker	icc	Intel compiler with dfa-lib-cpp	liliane-ubuntu	application	N/A
3	Liliane	amd64	Singularity	dfanalyzer	Container of dfanalyzer	liliane-ubuntu	provCollector	java
4	Liliane	amd64	Singularity	monetdb	Container of provdeploy database	liliane-imac20	database	N/A
5	Liliane	amd64	Singularity	py-readseq-modelGenerator	ReadSeq, python2, java, raxml, dfa-lib-python with telemetry, and psutils for python applications	liliane-iMac20	application	java
6	Liliane	N/A	N/A	java-readseq-modelGenerator	ReadSeq, python2, java, raxml, dfa-lib-python with telemetry, and psutils for Java applications	liliane-iMac20	application	java

DenseED explores variations of its CNN architecture (convolutional, discriminator/generator, conditional, etc.), training metrics, and parallelization techniques. It also explores multiple execution environments with different TensorFlow and CUDA versions (*i.e.*, multiple containers), and these changes have an impact on the training execution time.

Analyzing all these different details over time becomes increasingly complex, leading to a trial-and-error process. Without provenance, it is difficult to reproduce the same results or match DenseED with its compatible containers. For instance, even using a single environment like SDumont, users face this challenge since there are multiple GPU partitions with distinct devices available (e.g., K40 and V100). These devices require different combinations of cuDNN, CUDA, TensorFlow, and other software, resulting in distinct container images. Also, these combinations are affected by the image origins (e.g., Docker Hub, NCG, etc), which will ignore the GPUs if deployed in an incompatible partition, in the best scenario.

Using `ProvDeploy`, this information is captured and becomes available through the proposed data model implemented at the workflow provenance database, filling gaps in the execution information, as required in query Q7. Table 5 shows the results of query Q7 for each environment with different container compositions. We present only the container composition with the shortest elapsed time in each environment and the lowest R^2 for the DenseED execution. Query Q7 requires joining the entities *execution*, *env_partition*, *workflow_execution*, and *workflow* with those from the DenseED provenance data model, including hyperparameters and metrics within a constrained interval from the *execution* entity.

Table 5. Q7 - Container composition with the smallest elapsed time per environment and the best R^2.

job_id	Composition	Elapsed time(m)	Env Name	R^2
10898072	Partial modular	4.01	sequana_gpu	0.9979
10905992	Provenance Modular	20.55	cpu	0.9976
N/A	Coarse-grained	32.78	liliane-iMac20-1	0.9975

In query Q7, the differences in R^2 values are small, but we observe that, as expected, the elapsed time increases according to the type of environment. Notably, the best workflow container composition varies with the environment specification, especially in resource-limited scenarios such as the 'liliane-iMac20-1', which is a personal computer. In this environment, the coarse-grained composition was the best when considering only elapsed time. This occurs because a coarse-grained composition involves a single container image, enabling it to expand and freely use available resources. In resource-limited scenarios, other compositions tend to experience resource competition between containers. Additionally, they require extra time to start and stop containers, which accumulates over time, increasing the overall elapsed time.

Query Q8 can be successfully executed, identifying the best result in DenseED with the highest R^2 and the lowest RMSE. Query Q8 joins the entities *execution*, *env_partition*, *workflow_execution*, *container_image*, and *workflow* with those of the DenseED provenance, like hyperparameters and metrics within a restricted interval of DenseED training. Considering that DenseED was executed with various compositions and on different hosts, container provenance is crucial. Without it, the user would only know the values of K and L that led DenseED to the presented R^2 and RMSE. The user would lack information about the container images (tensorflow, dfanalyzer, monetdb) and the environment in which this model was executed (sequana_gpu), making it much harder to replicate or reproduce the execution.

5. Conclusion

Using containers to execute workflows in HPC environments can improve the portability and reproducibility of workflows. However, while containers enhance portability, they lack provenance data to assess reproducibility. In this paper, we explored container provenance aspects and introduced a container-aware provenance data model that extends an existing workflow provenance data model. Our goal is to provide container-aware provenance to improve workflow data analysis in HPC environments and improve trust in the use of containers. We evaluated this model using queries to help analyze and reproduce the workflow.

The proposed model is implemented in `ProvDeploy`. For its provenance to be successfully integrated with provenance from other services, certain requirements must be met by those services. For example, they must have a key attribute that identifies the workflow, which is necessary for integrating with the container-aware provenance data model. Despite such limitations, the proposed model represents a first step towards capturing container provenance associated with workflow provenance. In future work, we plan to integrate the proposed model with other provenance capture mechanisms and evaluate it with other types of workflows.

Acknowledgments

This paper was funded in part by CNPq, FAPERJ, and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. We acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer.

References

Abbas, M., Khan, S., Monum, A., Zaffar, F., et al. (2022). Paced: Provenance-based automated container escape detection. In *2022 IEEE IC2E*, pages 261–272. IEEE.

Ahmad, R., Nakamura, Y., Manne, N. N., and Malik, T. (2020). `PROV-CRT`: Provenance support for container runtimes. In *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*.

Campagna, D., da Silva, A., and Braganholo, V. (2020). Achieving gdpr compliance through provenance: An extended model. In *Anais do XXXV Simpósio Brasileiro de Bancos de Dados*, pages 13–24, Porto Alegre, RS, Brasil. SBC.

Canon, R. S. (2020). The role of containers in reproducibility. In *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 19–25. IEEE.

Chen, X., Irshad, H., Chen, Y., Gehani, A., and Yegneswaran, V. (2021). `CLARION`: Sound and clear provenance tracking for microservice deployments. In *30th USENIX Security*, pages 3989–4006.

Costa, F., Silva, V., de Oliveira, D., Ocaña, K. A. C. S., Ogasawara, E. S., Dias, J., and Mattoso, M. (2013). Capturing and querying workflow runtime provenance with PROV: a practical approach. In Guerrini, G., editor, *EDBT/ICDT '13*, pages 282–289.

Datta, P., Polinsky, I., Inam, M. A., Bates, A., and Enck, W. (2022). `ALASTOR`: Reconstructing the provenance of serverless intrusions. In *31st USENIX Security*, pages 2443–2460.

de Oliveira, D. C. M., 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.

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.

Freitas, R. S., Barbosa, C. H., Guerra, G. M., Coutinho, A. L., and Rochinha, F. A. (2021). An encoder-decoder deep surrogate for reverse time migration in seismic imaging under uncertainty. *Computational Geosciences*, 25:1229–1250.

Gruening, B., Sallou, O., Moreno, P., da Veiga Leprevost, F., Ménager, H., Søndergaard, D., Röst, H., Sachsenberg, T., O'Connor, B., Madeira, F., Dominguez Del Angel, V., Crusoe, M. R., Varma, S., Blankenberg, D., Jimenez, R. C., BioContainers Community, and Perez-Riverol, Y. (2018). Recommendations for the packaging and containerizing of bioinformatics software. *F1000Res*, 7.

Han, R., Zheng, M., Byna, S., Tang, H., Dong, B., Dai, D., Chen, Y., Kim, D., Hassoun, J., and Thorsley, D. (2024). `PROV-IO`$^+$: A cross-platform provenance framework for scientific data on hpc systems. *IEEE Transactions on Parallel and Distributed Systems*.

Kunstmann, L., Pina, D., de Oliveira, D., and Mattoso, M. (2024). `ProvDeploy`: Provenance-oriented containerization of high performance computing scientific workflows. *arXiv preprint arXiv:2403.15324 / Under Review*.

Kunstmann, L., Pina, D., de Oliveira, L. S., de Oliveira, D., and Mattoso, M. (2022). `ProvDeploy`: Explorando alternativas de conteinerização com proveniência para aplicações científicas com pad. In *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 49–60. SBC.

Lampa, S., Dahlö, M., Alvarsson, J., and Spjuth, O. (2019). Scipipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines. *GigaScience*, 8(5):giz044.

Merkel, D. et al. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2.

Modi, A., Reyad, M., Malik, T., and Gehani, A. (2023). Querying container provenance. In *Companion Proceedings of the ACM Web Conference 2023*, pages 1564–1567.

Moreau, L. and Groth, P. (2013). Provenance: an introduction to prov. *Synthesis lectures on the semantic web: theory and technology*, 3(4):1–129.

Murta, L., Braganholo, V., Chirigati, F., Koop, D., and Freire, J. (2015). noworkflow: capturing and analyzing provenance of scripts. In *IPAW 2014*, pages 71–83. Springer.

Novella, J. A., Emami Khoonsari, P., et al. (2019). Container-based bioinformatics with pachyderm. *Bioinformatics*, 35(5):839–846.

Olaya, P., Kennedy, D., et al. (2022). Building trust in earth science findings through data traceability and results explainability. *IEEE TPDS*, 34(2):704–717.

Orzechowski, M., Balis, B., Pawlik, K., Pawlik, M., and Malawski, M. (2018). Transparent deployment of scientific workflows across clouds-kubernetes approach. In *2018 IEEE/ACM UCC Companion*, pages 9–10. IEEE.

Paranhos, R., Lage, M., and de Oliveira, D. (2023). Uso de grafos de proveniência para análise temporal de uso do solo em centros urbanos: uma abordagem prática. In *Anais do XXXVIII Simpósio Brasileiro de Bancos de Dados*, pages 457–462, Porto Alegre, RS, Brasil. SBC.

Pina, D., Chapman, A., Kunstmann, L., de Oliveira, D., and Mattoso, M. (2024). `DLProv`: A data-centric support for deep learning workflow analyses. In *Companion of the 2024 ACM SIGMOD/PODS, Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning.*, DEEM '24, pages 77–85. ACM.

Priedhorsky, R., Canon, R. S., Randles, T., and Younge, A. J. (2021). Minimizing privilege for building hpc containers. In *IEEE/ACM SC*, pages 1–14.

Satapathy, U., Thakur, R., Chattopadhyay, S., and Chakraborty, S. (2023). Disprotrack: Distributed provenance tracking over serverless applications. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pages 1–10. IEEE.

Schlegel, M. and Sattler, K.-U. (2023). Management of machine learning lifecycle artifacts: A survey. *SIGMOD Rec.*, 51(4):18–35.

Shaffer, T., Phung, T. S., Chard, K., and Thain, D. (2023). Landlord: Coordinating dynamic software environments to reduce container sprawl. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1376–1389.

Silva, V., Campos, V., Guedes, T., Camata, J., de Oliveira, D., Coutinho, A. L., Valduriez, P., and Mattoso, M. (2020). Dfanalyzer: runtime dataflow analysis tool for computational science and engineering applications. *SoftwareX*, 12:100592.

Straesser, M., Bauer, A., Leppich, R., Herbst, N., Chard, K., Foster, I., and Kounev, S. (2023). An empirical study of container image configurations and their impact on start times. In *2023 IEEE/ACM 23rd CCGrid*, pages 94–105. IEEE.

Wofford, Q., Hurd, J., Greenberg, H., Bridges, P. G., and Ahrens, J. (2022). Complete provenance for application experiments with containers and hardware interface metadata. In *2022 IEEE/ACM CANOPIE-HPC*, pages 12–24. IEEE.

Zhu, Y. and Zabaras, N. (2018). Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification. *Journal of Computational Physics*, 366:415–447.