Building Trust in Earth Science Findings through Data Traceability and Results Explainability

Paula Olaya, Dominic Kennedy, Ricardo Llamas, Leobardo Valera, Rodrigo Vargas, Jay Lofstead, *Senior Member, IEEE*, and Michela Taufer, *Senior Member, IEEE*

Abstract—To trust findings in computational science, scientists need workflows that trace the data provenance and support results explainability. As workflows become more complex, tracing data provenance and explaining results become harder to achieve. In this paper, we propose a computational environment that automatically creates a workflow execution's record trail and invisibly attaches it to the workflow's output, enabling data traceability and results explainability. Our solution transforms existing container technology, includes tools for automatically annotating provenance metadata, and allows effective movement of data and metadata across the workflow execution. We demonstrate the capabilities of our environment with the study of SOMOSPIE, an earth science workflow. Through a suite of machine learning modeling techniques, this workflow predicts soil moisture values from the 27 km resolution satellite data down to higher resolutions necessary for policy making and precision agriculture. By running the workflow in our environment, we can identify the causes of different accuracy measurements for predicted soil moisture values in different resolutions of the input data and link different results to different machine learning methods used during the soil moisture downscaling, all without requiring scientists to know aspects of workflow design and implementation.

Index Terms—Scientific workflows, scientific computing, provenance, reproducibility, replicability, soil moisture predictions

1 Introduction

Computational workflows play a key role in scientific discovery. These workflows are growing more complex: they consist of different modeling, analysis, and visualization modules; they run on increasingly heterogeneous systems; and they use machine learning (ML) methods with limited transparency. For scientists using these workflows to study scientific phenomena, trusting data, methods, software, and hardware becomes more necessary than ever. Scientists can trust their findings only through in-depth data lineage and the complete record trail of the methods generating the results. The data lineage and record trail combined enables scientists to trace data back to its sources and explain computational methods and their output.

Provenance collection techniques [1], [2], [3] and container technologies [4], [5], [6], [7] are promising approaches to achieve data traceability through data lineage and result explainability through record trails. Provenance provides data lineage with a thorough description of the history of the data evolution, allowing the scientist to trace the data back to its origin and observe interactions between data and applications. However, there are two main limitations of current provenance solutions. First, the provenance metadata is separate from the dataflow, so any effort to match metadata to workflow components (i.e., data and applications) requires manual work. For example, solutions such as PASS [8], Pachyderm [9], and REANA [10] use separate metadata databases. Because the metadata is in a different location than the workflow components, it is harder for the scientists to query metadata and match with the components to do any provenance analysis. Second, scientists track provenance with custom systems that do not offer portability across heterogeneous platforms. For example, work in [8], [11], [12] builds on custom file systems, and work in [13], [14], [15], [16], [17] builds on custom software packaging. Containers offer a lightweight solution to track provenance across platforms by encapsulating applications and dependencies into an isolated environment [18], [19], [20], [21]. However, there are three main limitations of current container solutions. First, containers are used for services that do not save long-term states; state storage is relegated to existing, shared storage infrastructure. Second, container solutions do not automatically create data lineage and record trails. Third, current container technologies lack an effective way to move data through a containerized workflow.

To overcome these limitations, we propose a computational environment that is seamlessly integrated with container technology, automatically creates a workflow execution's record trail, and invisibly attaches the trail to the workflow's intermediate and output data. To this end, we decouple data and applications of traditionally tightly coupled workflows and encapsulate data and applications into

- *Paula Olaya, Dominic Kennedy, Leobardo Valera, and Michela Taufer are with the University of Tennessee, Knoxville, TN 37996 USA. E-mail: {polaya, dkenne15}@vols.utk.edu, leobardovalera@gmail.com, taufer@utk.edu.*
- *Ricardo Llamas and Rodrigo Vargas are with the University of Delaware, Newark, DE 19716 USA. E-mail: {rllamas, rvargas}@udel.edu.*
- *Jay Lofstead is with Sandia National Laboratories, Albuquerque, NM 87123 USA. E-mail: gflofst@sandia.gov.*

Manuscript received 19 February 2022; revised 25 October 2022; accepted 26 October 2022. Date of publication 8 November 2022; date of current version 27 December 2022.
This work supported in part by Sandia National Laboratories, in part by National Science Foundation through under Grants 1841758, 1941443, 2028923, 2103845, 2103836, and 2138811, in part by IBM through a Shared University Research Award, and in part by XSEDE program through the NSF under Grant 1548562.
(Corresponding author: Michela Taufer.)
Recommended for acceptance by R. Prodan.
Digital Object Identifier no. 10.1109/TPDS.2022.3220539

individual fine-grained containers. We augment both data and application containers to expose provenance metadata and to move data across the containerized workflow effectively. Additionally, we create an interface for visualizing and studying the metadata that scientists can use to understand data lineage and computational methods. We demonstrate how our environment enables data traceability and results explainability for the SOMOSPIE (Soil Moisture Spatial Inference Engine) workflow [22]. SOMOSPIE uses a suite of ML modeling techniques to downscale the 27 km resolution satellite data from the ESA-CCI soil moisture database [23] to higher resolutions necessary for practical use in earth sciences including precision forestry and agriculture, hydrology for landscape ecology, and regeneration dynamics [24], [25].

Our environment enables scientists to link differences in scientific results back to different input data sources (data lineage for traceability); and link different results to specific methods used, without necessarily mastering all the aspects of implementation and execution of the workflow (record trail for explainability). For example, with SOMOSPIE, our environment enables scientists to identify different predicted soil moisture values caused by different input data sources and their resolutions; or to connect different results to specific ML methods used during the soil moisture downscaling. We measure the performance of our environment in terms of execution time, storage space, and IO bandwidth. We demonstrate how our environment has limited overhead and is effective for establishing trustworthiness in the SOMOSPIE workflow. While we demonstrate the benefits of our environment with soil moisture prediction, our solution is application-agnostic: our environment can be easily adapted to general workflows consisting of self-contained applications.

This work makes the following contributions:

- An environment based on fine-grained containerization of both data and applications which automatically creates data lineage and record trail of workflow executions, enabling traceability of data and explainability of results.
- A Singularity/Apptainer based implementation of our fine-grained containerized environment which automatically annotates each container with its data lineage and record trail and effectively supports zero-copy movement of data across containers.
- A demonstration of the environment capabilities to trace data sources and explain ML results for an exemplary earth science workflow, SOMOSPIE [22].

The paper is organized as follows. Section 2 describes the methodology to model a workflow into a fine-grained containerized environment. The implementation of the environment using Singularity/Apptainer as the selected container technology is explained in Section 3. Section 4 demonstrates the use of our containerized environment for an earth science workflow for traceability and explainability. We quantify the impact on overhead and performance in Section 5. We discuss the interoperability and adaptation of our environment in distributed systems in Section 6. Section 7 discusses the related work. Finally, Section 8 concludes with a summary of the findings and directions for future work.

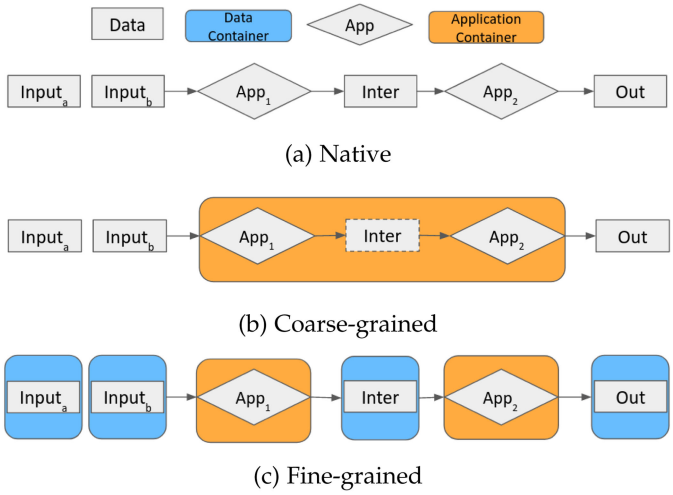


Fig. 1. Composition of scientific workflows on (a) native environment (i.e., original workflow execution on backbone HPC or cloud platforms), (b) using a coarse-grained containerized environment with a single monolithic container and (c) using our proposed fine-grained containerized environment (i.e., decoupled workflow components in data and application independent containers).

2 Modeling Containerized Workflows

We describe how a scientific workflow can be modeled using our fine-grained containerized environment. We present our solution in terms of the decoupling of workflows into application and data components, the communication between these components, the automatic annotation of each component, and the visualization of the associated metadata.

2.1 From Native to Fine-grained Workflow Modeling

A workflow is composed of one or multiple interoperable, self-contained applications, each with its own software stack and input and output data. These applications can range from data generation, data collection and merging, data preprocessing and feature extraction to data analysis, modeling, and visualization. Such a workflow can be executed on native HPC and cloud platforms (Fig. 1a). When container technologies are used in HPC or cloud platforms, the whole workflow is usually deployed in a single coarse-grained container (Fig. 1b). This coarse-grained containerization enables easy deployment and management. However, the coarse-grained approach makes it difficult to exactly track executions or identify all the workflow components and their interactions for building an in-depth data lineage and record trail. In addition, such an approach does not enable reusability and composability of individual workflow components. Instead, we decouple the workflow into its components (i.e., applications and data) and use a fine-grained containerized workflow approach that encapsulates each workflow component into its own container (Fig. 1c). Each container serves as an immutable object with a unique hash code for permanent identification, enabling easy data lineage and record trails creation. Our approach is applicable to workflows that can be modeled as directed acyclic graphs (DAGs) whose nodes (applications) and vertices (data in movement from one application to another) can be both containerized. In

theory any workflow with such features can be abstracted into a fine-grained set of interconnected containers.

2.2 Designing Application and Data Containers

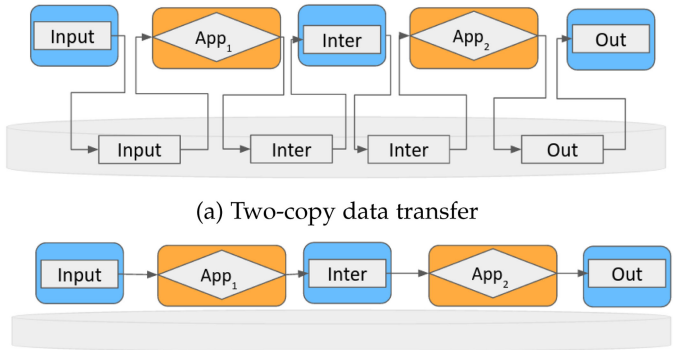
Given an application, we containerize it by encapsulating the executable or script with the respective software stack (i.e., OS, libraries, and software packages). By doing so, scientists can reuse the application container across different workflows and reduce the overall storage requirement. Furthermore, in our environment all application containers are annotated with provenance information including their unique identifier and creation time. The containerization of data is unique to our approach. Because containers are isolated systems, they do not support persistence of data; consequently, only applications are normally containerized, while the data is hosted in local databases, storage volumes, or images [26], [27], [28]. We move away from this implementation by leveraging the work of Lofstead and co-authors called Data Pallets [29] that defines data as a separate and immutable object once created. To create this object, we define a data container that follows a file-system-in-a-file model. Given an individual dataset (i.e., input, intermediary, or output data), we containerize it into a single and independent data container. Similar to the application container, data containers are augmented to expose provenance information such as the unique hash code, creation time, execution task and record trail. By doing so, data containers provide trustworthiness, portability, and shareability of their content to users and across workflows. Two important principles inform our fine-grained containerized workflow approach. First, we separate applications and data into their own containers allowing specificity and unique identification of the components in a workflow for traceability and explainability purposes. Second, our approach values intermediate data; rather than discharging intermediate data, it encapsulates this data in order to provide a complete preservation of the data lineage for traceability and reusability.

2.3 Designing Communication Between Containers

The fine-grained containerization of the workflow components introduces a new challenge in the execution: data has to be moved from one container to the next in an efficient way. When the movement is done through standard container technology, it requires the usage of the host node's storage to serve as a buffer in a two-copy data transfer as shown in Fig. 2a. We propose a new communication approach that enables a zero-copy data transfer: we bind mount direct paths inside the data and application containers through their namespaces, thus avoiding data sharing via the host. This allows containers to directly exchange data without creating extra copies or using external storage. Ultimately, our approach reduces the time and space needed to transfer data between containers. Fig. 2b shows the same example of data transfer for the two-copy approach in Fig. 2a but with our zero-copy data transfer implementation. In essence, an application container can read and write directly from one or multiple data containers.

2.4 Designing Annotated Containers

The fine-grained containerization of a workflow allows us to annotate its executions, capturing metadata at a fine-grained level. To deploy the annotation for different workflow executions, the metadata collection has to be automatic. There are three open questions when automatically annotating workflow executions:



(a) Two-copy data transfer

(b) Zero-copy data transfer

Fig. 2. Communication between the workflow components in two-copy (a) and zero-copy (b) data transfer for our fine-grained containerized environment.

1) *Where should the metadata be allocated?* We augment each container with an extra partition, within which we allocate the metadata. Our approach enables the tight integration of each workflow component and its metadata, facilitating the traceability of both the individual components and the workflow as a whole. While tightly coupling the metadata partition with the workflow components, our environment effectively stashes the metadata information from regular workflow executions. This means that the data or application itself in the container can never be contaminated by managing the metadata partition. The metadata information can still be accessed at any time through standard container partition access commands.
2) *What metadata should be captured?* We capture the container's identification, creation time, execution task, and record trail. The container identification is a tuple (i.e., `[UUID, name]`) composed of a universally unique identifier (i.e., `UUID`) and a name assigned when the container is created. While the `UUID` is unique, the name can be modified. The creation time captures the point in time when the container is written to disk. The execution task is the set of instructions to run the application. This information includes parameters such as initial conditions, random seeds, and other setting values. The container's record trail is a set of ordered tuples (i.e., `[UUID, name]`) that defines the pipeline of containers preceding the current one (i.e., data and application containers used before the current container and triggering its use).
3) *How and when should the metadata be collected?* For each container, we expose information embedded in the container environment, which is otherwise hidden to the workflow execution, to collect our metadata. For both data and application containers, the structure of the metadata partition (i.e., `[container's identification, creation time, execution execution task, and record trail]`) is

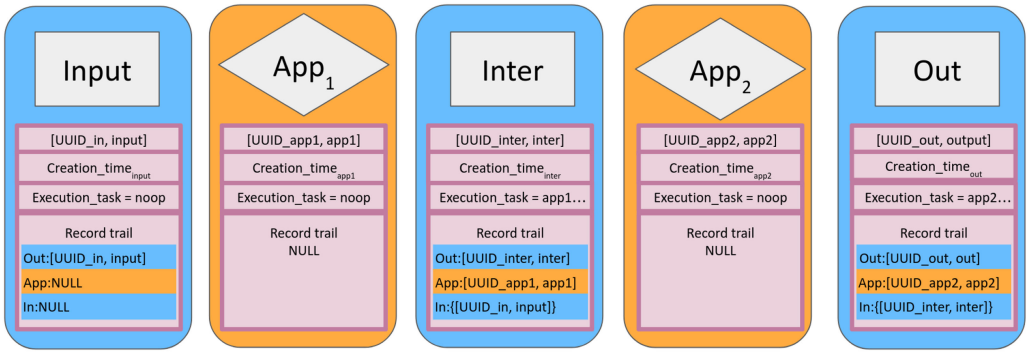


Fig. 3. Example of the metadata partitions for a workflow in our fine-grained containerized environment, composed of once executed three data containers serving as input ($Input$), intermediate ($Inter$), and output (Out) respectively, and two application containers (App_1 and App_2). The partitions are populated with both static and dynamic information.

identical. The content of the partitions depends on the type of container and is collected both statically and dynamically. The static metadata is collected when a container is created and includes the container identification and creation time. The pool of dynamic metadata includes the execution task and record trail. For all containers, (a) the execution task is initialized with `noop` meaning that there is no operation; and (b) the record trail is initialized with `NULL` meaning that there are not predecessor containers because an execution has not happened yet. The dynamic metadata is populated only at the time a workflow is executed. For purposes of reusability across workflows, the static metadata of application containers remains as initialized. On the other hand, the execution task and record trail of the data containers are updated at the execution time to capture the data generation.

Fig. 3 shows an example of automatic metadata collection for a workflow with three data containers serving as input ($Input$), intermediate ($Inter$), and output (Out) respectively, and two application containers (App_1 and App_2). The static metadata is initialized with the container identification tuple and container creation time. During the execution of the workflow, the execution task and record trail are updated for the data containers. The figure presents a snapshot of when the whole workflow has finished its execution, meaning that App_1 and App_2 have written the results into the intermediate and output containers respectively. The record trail of the output container includes the output, App_2, and intermediate containers' identifications. With the identification of the intermediate, we can retrieve its record trail which includes the intermediate, application1, and input containers' identifications. We merge the two containers' record trails and define all the containers used in the workflow pipeline. The combination of the static and dynamic metadata enables building the in-depth data lineage and the complete record trail of the applications generating the results.

2.5 User Interface

We provide a user interface to facilitate the study of collected metadata. This interface reads the metadata of one or multiple containers. From this metadata, the interface constructs a workflow visual representation as a directed graph. Scientists can interact and adapt the graph to their needs for further analysis. The nodes of the graph represent each container in the workflow which are labeled by the universally unique identifiers (i.e., `UUID`) and include their respective metadata as their description. Additionally, nodes are distinguished between data and application containers. For each container, the interface backtraces its record trail and connects the nodes in the order of the lineage of that execution. Subsequently, the interface constructs individual subgraphs of each execution. These subgraphs are merged to find common patterns and containers shared across multiple executions, providing a graph with unique nodes (data and application containers) connected based on the data lineage constructed from the descriptions of the metadata.

3 Singularity/Apptainer Implementation

We build our computational environment by augmenting Singularity/Apptainer [5] container technology. We develop a Jupyter Notebook [30] interface for the metadata visualization and analysis.

3.1 Selecting the Container Technology

While there are many different container technologies (e.g., Docker [4], Singularity/Apptainer [5], Charliecloud [6], Podman [7]), in our work, we use Singularity/Apptainer for three main reasons. First, it does not require administrative privileges that are challenging to obtain in tightly controlled environments such as the US National Laboratories. Second, the use of the SIF (Singularity Image File) format container allows us to customize the content of each container with different types of partitions (i.e., metadata partition, application partition, data partition), where each container can have one or more partitions. Last, Singularity/Apptainer supports user-defined add-on functionalities through plugins. These plugins are packages that can be dynamically loaded by the Singularity/Apptainer at runtime, augmenting Singularity/Apptainer with experimental, non-standard, and vendor-specific functionalities. Some of these functionalities allow users to add commands and flags for the container's creation and execution; the functionalities can serve as an interface with more complex subsystems (i.e, compute and storage devices) at runtime. We extend Singularity/Apptainer to feature three functionalities needed to implement the designed fine-

grained containerized environment in Sections 2.2, 2.3, and 2.4. First, we use the SIF format container to automatically create the individual application and data containers. We initialize each container with a metadata partition, providing users with access to information otherwise hidden to them. Second, we design a zero-copy data transfer mechanism for the Singularity/Apptainer's technology which facilitates data movement across containers. This zero-copy functionality is now part of the Singularity/Apptainer code [31]. Last, we automatically annotate the workflow with provenance information. Both automatic creation and annotation functionalities are integrated in a plugin that is part of our software release [32], [33]. Furthermore, we implement a Jupyter notebook that serves as the user interface designed in Section 2.5.

3.2 Creating Application and Data Containers

We augment the Singularity/Apptainer runtime with a plugin that supports the automatic creation and execution of fine-grained containerized workflows. The plugin can work with any workflow that can be modeled as a DAG. Given a workflow, our plugin has the ability to automatically create a fine-grained sequence of data and application containers using `apptainer workflow -create`. We use a web service on the local machine at port `5000` to facilitate the user with the generation of the fine-grained workflow. Through the web service, the user provides information about the workflow, such as the definition file of an application (i.e., application executable and software stack); the number, location, and size of each input data; and the expected size of the output data. The plugin uses this information to create the workflow with its individual application and data containers. Specifically, the plugin encapsulates individual datasets and application executables or scripts into independent file system partitions. For application containers, the plugin encapsulates the application executable or script together with the software system stack in a squashFS partition. For data containers, the plugin compresses the data in an Ext3 file system partition. Both types of containers include metadata in a JSON generic file system partition.

3.3 Zero-copy Communication Between Containers

We extend the Singularity/Apptainer technology to support direct transferring data between containers without going through host or external storage (i.e., zero-copy data transfer). We use the bind mount functionality to define a bind path that directly links a directory from the source container to a directory in the destination container. During an execution, the bind path is parsed, capturing which containers and directories to use. The source and destination containers are loaded in the environment. The source directory is replicated in the destination container; any change on the directory inside the destination container is also reflected in the source container and vice versa.

3.4 Implementing Annotated Containers

We define a second functionality in our plugin `apptainer workflow -run [workflow_description].json` that grants the user the ability to execute a fine-grained containerized workflow while also annotating containers with metadata. A user executes the fine-grained containerized workflows by running the command `apptainer workflow -run [workflow_description].json`. This command triggers the Singularity/Apptainer API callback `clicallback` that activates our plugin. Once the plugin is active, it starts the automatic collection of metadata in the workflow. To this end, for each application container's execution, the plugin collects two pieces of information. First, it collects the application's execution task settings (e.g., initial conditions, random seeds, and other setting values). Second, it collects the bind path that lists all input and output data containers for that application container. Following the data containers listed in a bind path, the plugin uses the SIF API to extract each container's identification and creation time. The plugin appends that information to the record trail of output data containers for that bind path. Given an output container, its execution task and record trail are transformed into JSON format and added as a new file descriptor in its metadata partition.

3.5 Jupyter Notebook User Interface

We develop a Jupyter notebook [30] that serves as a user interface for inspecting and gaining insights from the collected metadata. It allows the user to select the metadata of one or more containers and to backtrace the execution dataflow, which is represented through a directed graph. We use the package *NetworkX* expanding the open-source function *NetworkX Viewer* [34] with a customized node token class to tailor the interactive visualization of the workflow graph. Specifically, our interface enables users to (i) create the nodes based on the list of containers in the record trail where the attributes of each node follow the structure of the metadata; (ii) assign different colors to distinguish between data and application containers; (iii) build independent subgraphs based on the dataflow stated in the metadata of each container; (iv) merge the subgraphs to build larger graphs by using common patterns and unique components; (v) visualize the graphs in an interactive session where the user can reorganize it; and (vi) obtain detailed provenance information about any container.

4 Traceability and Explainability in a Real Use Case in Earth Science

We demonstrate how our fine-grained containerized approach is used to study two cases on an earth science workflow, SOMOSPIE [22]. In the first case, the workflow has to be traced back to the input data to explain different levels of predictions' resolutions (a case of missing data traceability). In the second case, the annotations are used to discover which different ML methods are the reason for different soil moisture predictions when using the same data (a case of missing result explainability).

4.1 A Data-driven Workflow for Soil Moisture Prediction

Soil moisture, the percentage of water by weight or volume in soil, is critical for linking climate dynamics to ecosystem functioning, playing a key role in the Earth's water and carbon cycles. The current availability of spatial soil moisture information across large areas (e.g., continents) comes from satellite-based remote sensing sources (e.g., ESA CCI [23], NASA

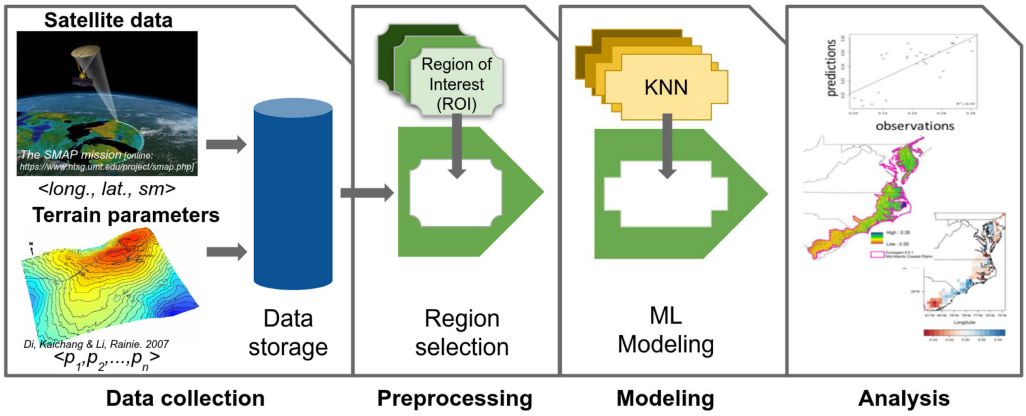


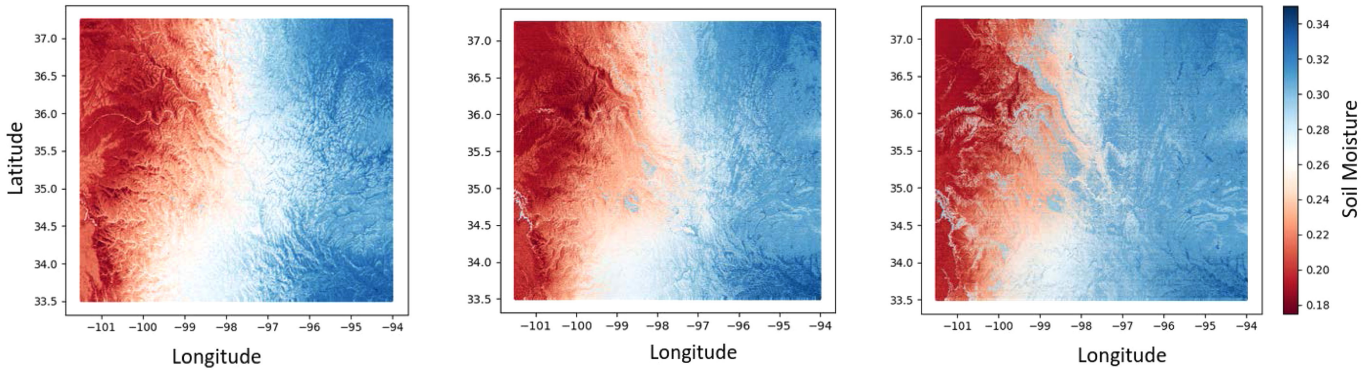
Fig. 4. SOMOPIE's modular workflow for predicting soil moisture composed of four modules (i.e., data collation, preprocessing, modeling, and analysis).

SMAP [35]). There are however two major limitations of satellite-based soil moisture information: (i) large areas have spatial information gaps (e.g., where there is high canopy density, frozen soil, or extremely dry conditions); and (ii) they have coarse granularity (around 27×27 km grids). There is a pressing need to improve the spatial representation of soil moisture for applications in earth sciences (e.g., ecological niche modeling, carbon monitoring systems, and other Earth system models). Predictions can be used in policy making and precision agriculture (e.g., optimizing irrigation practices and other land management decisions). We use the open-source workflow for fine-grained soil moisture predictions called SOMOPIE [22]. The workflow fills missing spatial information and increases spatial resolution of the satellite information. The workflow consists of three steps: (i) satellite data and terrain parameters are input to a sequence of execution steps; (ii) ML methods transform the satellite data into higher resolution and gap-free predictions using K-Nearest Neighbors (KNN), Random Forest (RF), and Surrogate Based Modeling (SBM); and (iii) visualization methods rebuild predictions into formats suitable for further study. Fig. 4 shows an abstraction of the workflow. For scientists, the entire workflow execution can be opaque, preventing easy data traceability and results explainability. For example, different resolutions of the terrain parameters data (different input data) used for generating fine-grained predictions are not easy to trace from the output. Results obtained using different ML methods (different executables) are not easy to link to the specific method used. We show how our fine-grained containerized approach can be applied to the SOMOSPIE workflow to enable traceability and explainability in two cases. In the first case, the different input data fed into the workflow has to be traced back in order to explain the different levels of details in the predictions (a case of missing traceability). In the second case, different ML methods are the reason for different soil moisture predictions when using the same data (a case of missing explainability).

4.2 Integrating Traceability

Fig. 5 shows an example of missing data traceability. SOMOSPIE is used with different input data resulting in different levels of details for the prediction of a region centered around Oklahoma (a rich agricultural area). The longitude is on the x-axis, the latitude is on the y-axis, and each of the pixels/coordinates represents a soil moisture (SM) value. Each one of the three figures represents the same area of Oklahoma where the longitude runs from -101.5 to -94.0 and the latitude runs from 33.5 to 37.0. The soil moisture ranges from 0.175 to 0.35 and is mapped into a color gradient where the lowest and driest SM value is red and the highest and most moisturized SM value is blue.

The terrain parameters are selected from different datasets with different resolutions: 1 km, 250 m, and 90 m. The ML method is fed with the input datasets and generates predictions from the satellite resolution (27 km) down to the terrain parameters resolution. However, the figures do not reveal the different resolutions of the input data to the scientists using the same workflow. Our approach annotates the workflow to capture the data provenance in the metadata, providing scientists with full transparency in the data transformations from input to output in the workflow. Fig. 6 shows the output of our interface for the predicted soil moisture values in Fig. 5. The interface is fed with the metadata of the containers. Based on the metadata, the interface builds and represents the workflow as a graph. The graph shows four data containers (nodes in blue), symbolizing the input data containers connected to an application container (first orange node). This application container corresponds to the ML method that generates the three data containers with the soil moisture predictions at a higher resolution. These predictions are then visualized in the second application container (second orange node), which are encapsulated in three independent output data containers (shown in Fig. 5). Based on the graph representation, the scientist can interact with any container and obtain its lineage. Fig. 6 presents the metadata partition of each of the containers executed in the workflow. This metadata partition includes the `UUID` (simplified in the figure), container name, creation time, execution task, and the record trail. The record trail shows the lineage of containers that were used to generate the current component. Starting from the three output data containers (09,10,11) the record trail shows in bold that the same visualization application (08) was executed on three independent predictions' datasets (05, 06, 07). Based on the metadata of the intermediate data containers (05, 06, 07), the record trail reveals (in bold) that each of these predictions were the result of executing the KNN application (04) with the same training data (00) and three independent evaluation datasets with different resolutions (01,02,03). Finally, scientists can trace the three different outputs back to the data sources and explain



(a) 1 km (b) 250 m (c) 90 m

Fig. 5. Example of soil moisture output visualization for Oklahoma predicted on three different resolutions (i.e., 1 km, 250 m, and 90 m) generated with SOMOPIE.

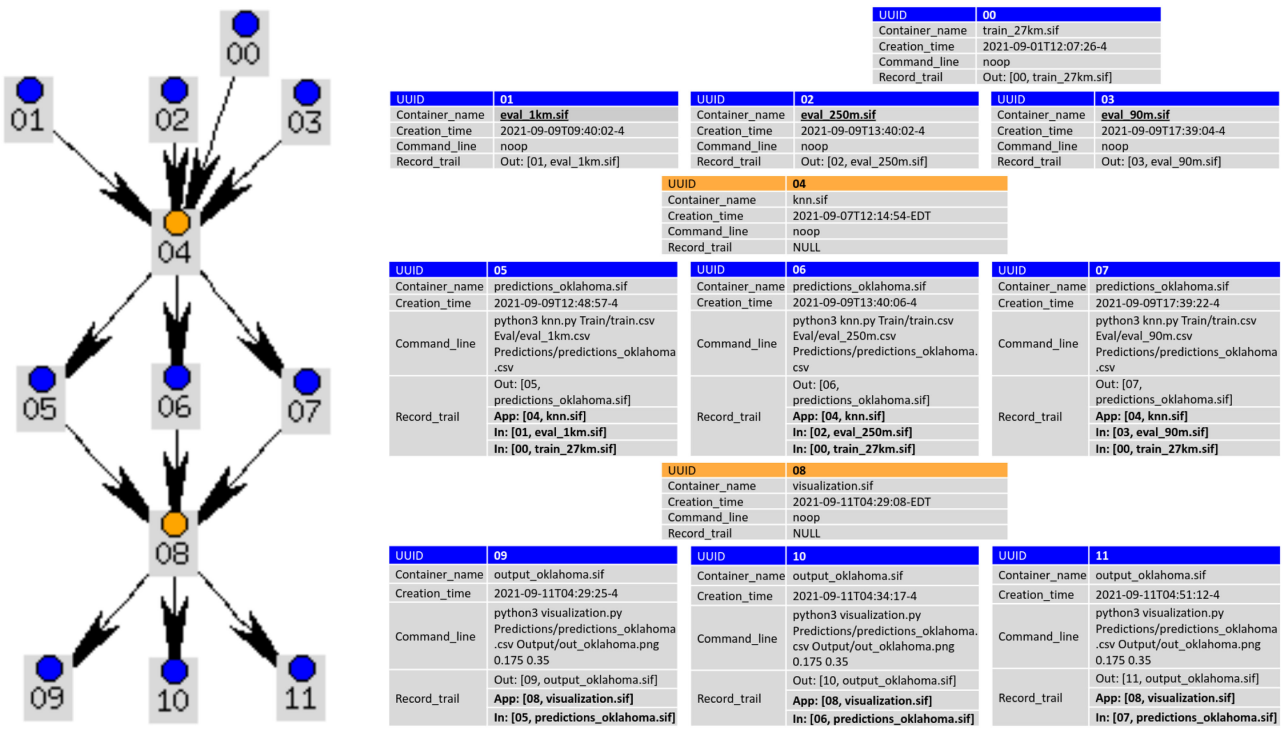
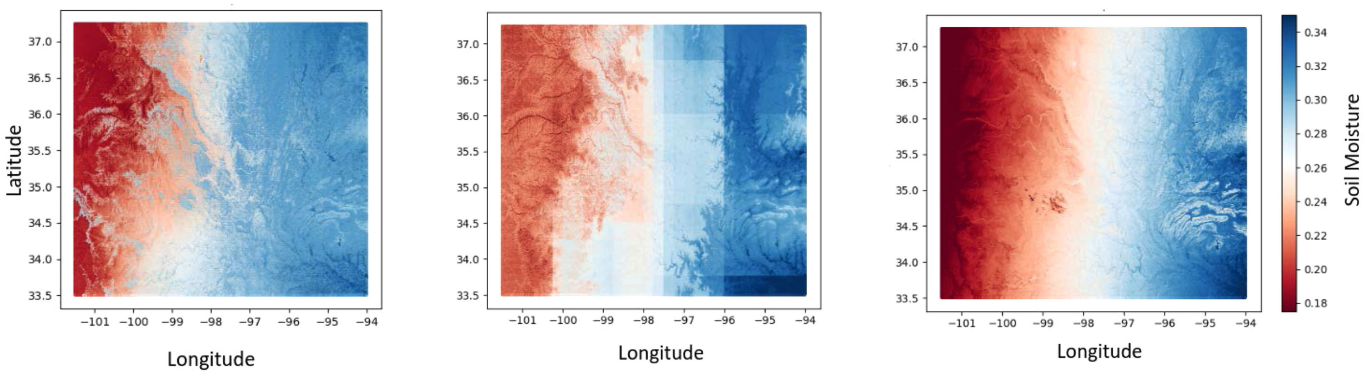


Fig. 6. Graph representation of the SOMOSPIE workflow executions for Oklahoma on three resolutions (i.e., 1 km, 250 m, and 90 m) generated by the Jupyter notebook interface. It includes the metadata visualization for each containerized component.

how the observed differences come from the three input datasets, each with a different resolution. Furthermore, because our interface directly maps any difference to specific containers used during the execution, it makes it easier for the scientists to retrieve those containers and use them to reproduce the results or generate new studies.

4.3 Integrating Explainability

Fig. 7 shows an example of lack of result explainability for the same region. The figure shows three visualizations of the fine-grained soil moisture predictions with the same resolution of 250 m. The three predictions are generated using the same input data but different prediction methods (i.e., KNN, RF, SBM). The figures and associated fine-grained predictions do not reveal any information about the reasoning beyond the sharpness and high mixture of dry and moist values in Fig. 7a, the tile-like values in Fig. 7b, or the smoother transition between dry to moisturized values in Fig. 7c. Once again, our automatic annotation of the workflow generates metadata revealing the differences in the output data, providing the scientists with an explanation of the results in terms of the methodology used. Fig. 8 shows the output of our interface for the predicted soil moisture in Fig. 7. As in the previous case, the interface is fed with the metadata of the containers, and based on the metadata it builds the workflow as a graph. The output graph of the interface shows that there are three possible paths that start from two input data containers (both blue nodes). A fork in the execution occurs at the first stage of the workflow where there are three different application containers corresponding to the three different ML methods (first three orange nodes). Each of the ML methods generates an intermediate data container with the soil moisture predictions. Finally, the three predictions are visualized, generating three independent



(a) K-Nearest Neighbors (KNN) (b) Random Forest (RF) (c) Surrogate-based Modeling (SBM)

Fig. 7. Example of soil moisture output visualization for Oklahoma predicted with three different ML methods (i.e., KNN, RF, and SBM) generated with SOMOSPIE.

Fig. 8. Graph representation of the SOMOSPIE workflow executions for Oklahoma with three ML methods (KNN, RF, and SBM) generated by the Jupyter notebook interface. It includes the metadata visualization for each containerized component.

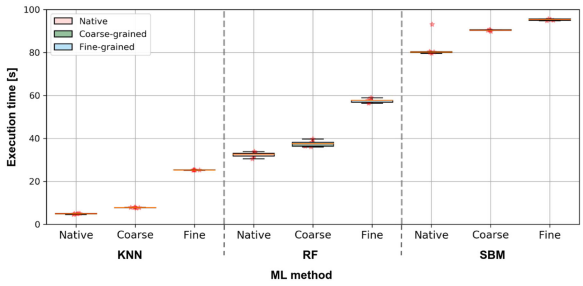
output data containers including the three visualizations (last three blue nodes). We also present the metadata partition for each of the containerized components. By backtracking the record trail, it is possible to reveal that the output containers (10,16,17) were generated by using the soil moisture predictions (06,14,15) from three different ML methods (04,12,13). By providing scientists with the workflow annotations automatically generated by our environment, we enable scientists to explain the different results by linking them to the ML methods used during downscaling. Furthermore, the application containers can be reused to generate new workflows without re-writing the application or re-installing the software stack.

5 Measuring Overheads and Performance

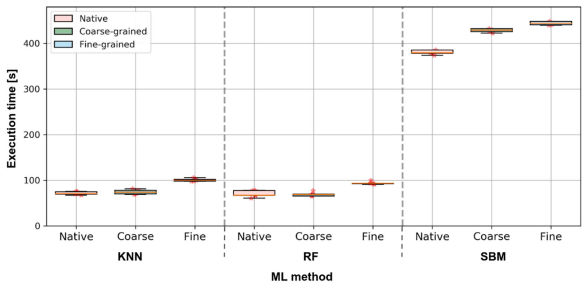
We collect the performance measurements of our fine-grained containerized environment and compare them with both a native environment (without containerization) and a coarse-grained containerized environment (for which we containerize all the applications inside a single container and store the data on the native memory). The three environments are part of a broader set of environment configurations with our and the native settings at the two ends of the testing spectrum and the coarse-grained environment as a trade-off in between the other two. In other words, the coarse-grained environment (single container) serves as the link between the native (no containers) and the fine-grained (multiple independent containers) environments.

5.1 Experimentation Platform Settings

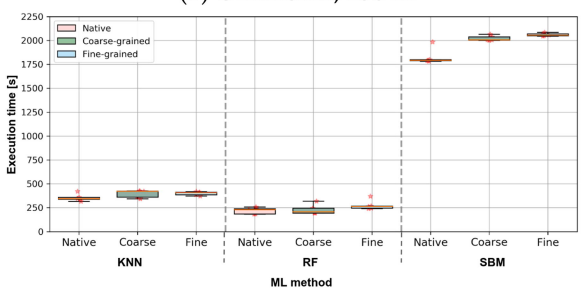
We run a diverse set of tests for SOMOSPIE and collect execution time in seconds and storage space in MBs for the different environments. Furthermore, we measure the bandwidth in MB/s for different data read and write workloads. We run the tests on XSEDE Jetstream2 [36] configured as a virtual



(a) Oklahoma, 1 km



(b) Oklahoma, 250 m



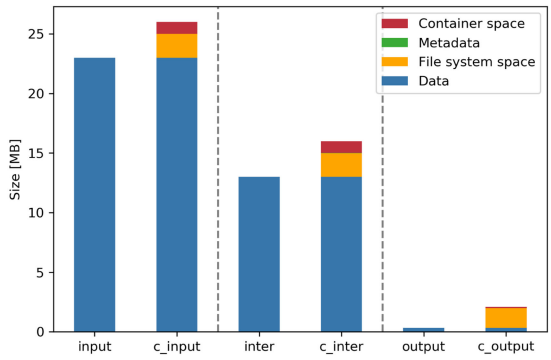
(c) Oklahoma, 90 m

Fig. 9. Execution time of SOMOSPIE, running on Oklahoma with three resolutions: (a) 1 km, (b) 250 m, and (c) 90m, with three ML methods comparing the native (no containers), coarse (single container), and fine-grained containerized (multiple containers) environments.

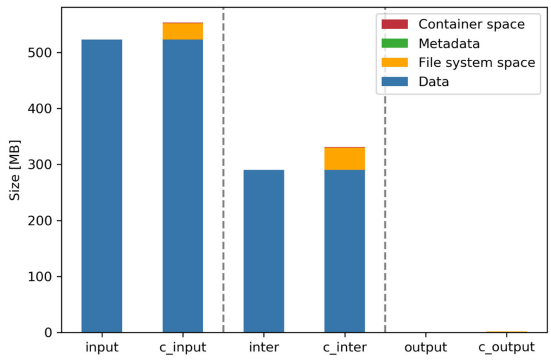
machine with 16 cores with AMD Milan 7713 CPU type, 60 GB of RAM (DIMM), and 60 GB local disk (HDD) space with an attached volume of 100 GB. In terms of software, the virtual machine has Ubuntu 20.04 as the OS, Apptainer 1.1, Go 1.19, and Python 3.8.10 using the next packages: numpy-1.23.2, pandas-1.4.3, scipy-1.9.0, and scikit-learn-1.1.2.

5.2 Execution Times

We measure the execution times of all three environments (i.e., the native, the coarse-grained, and the fine-grained). Specifically, we measure the execution times of SOMOSPIE on Oklahoma with three resolutions: (a) 1 km, (b) 250 m, and (c) 90 m. We run each resolution 5 times using the three ML methods (i.e., KNN, RF, and SBM) for the different environments on top of Jetstream2. Fig. 9 shows the results. In the figure, we observe that the smaller the data and the more inexpensive the application is, the more time overhead is seen when deploying our fine-grained containerized environment compared to the native (77%) and the coarse-grained (53%) environments. As the data increases and the application becomes more complex and time consuming, the overhead drops to 10% compared to native and 1.5% compared to coarse-grained. Overhead is always expected



(a) Oklahoma, 1 km



(b) Oklahoma, 250 m

Fig. 10. Comparison between the native and the fine-grained containerized environment for the data storage including the input, intermediate (inter), and output data of the earth science workflow running in Oklahoma with two resolutions: (a) 1 km, and (b) 250 m.

given the extra layers of virtualization and the metadata management. Still, as applications are more complex and are deploying larger data, the observed overhead is an acceptable trade-off for the gained traceability of data and explainability of results.

5.3 Storage Space

We measure the storage space usage for the two environments at each end of our testing spectrum (i.e., native and our fine-grained). We do not present results for the coarse-grained containerized environment because we expect its storage space to match the storage of the native for data, and to match the fine-grained for applications. As we encapsulate the workflow components (data and applications) in containers, the containerization adds extra space for the encapsulation format. Depending on the type of containers, this extra space is allocated for different purposes. We distinguish between data and application containers and compare the storage space used in the native environment versus our fine-grained containerized environment.

Data containers: In the native environment we have data, and in our fine-grained containerized environment we have data containers. A data container includes the data encapsulated in an Ext3 file system, metadata partition, and the container dependencies. We measure the size of all data containers in SOMOSPIE when executing the three ML methods for the different input data resolutions (1 km, 250 m, 90 m). Fig. 10 presents the size comparison between the

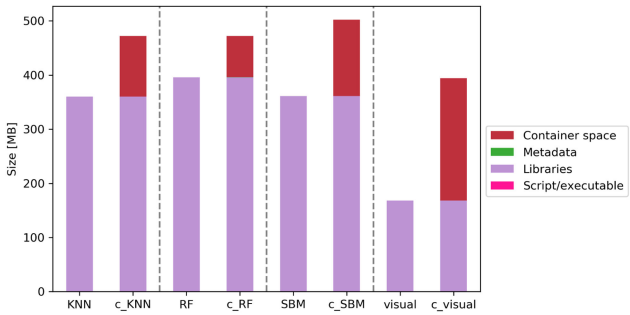


Fig. 11. Comparison between the native and the fine-grained containerized environment for the applications storage in SOMOSPIE.

native environment and our environment for Oklahoma with two resolutions: (a) 1 km and (b) 250 m. On the x-axis we have the different data (data) and its containerization (c_data): input, c_input, intermediate (inter), c_inter, output, and c_output. Each is represented by a bar indicating their size in MBs, as stated on the y-axis. First, we observe that the container dependencies' space (red bar) is constant. As the data increases in size, the container dependencies' space becomes imperceptible. Second, for all containers, we observe that the space of the metadata partition (green bar) is negligible and always in the order of KBs. Last, we see that the extra space in the data container is mostly part of the Ext3 file system space (orange bar) and is used to encapsulate the data (blue bar) that has the same size in both native and containerized environments. The main cause of the large space used by the Ext3 file system is that it reserves space for the journaling information and 5% space for root processes. Ways of reducing the reserved space have been explored, and Ext4 [37] incorporates scalability and performance enhancements. However, Singularity/Apptainer chose Ext3 for their data containers as it is the most efficient and modifiable file system currently available on a wide variety of systems.

Application containers: In the native environment there are applications in the form of scripts or executables which require libraries and dependencies to run. In our fine-grained containerized environment we have application containers that include a script or executable, libraries, a metadata partition, and the container space which corresponds to the OS and software dependencies. Fig. 11 presents the comparison between native and fine-grained containerized applications in SOMOSPIE. On the x-axis, we have the three ML methods (application) and their containerization (c_application): KNN, c_KNN, RF, c_RF, SBM, c_SBM, and the visualization (visual) with its containerization (c_visual), all written in Python. On the y-axis we have the size in MBs. We observe these key properties: first, the scripts and libraries occupy the same space for the native and our containerized environment; second, as in the data containers, the metadata partition is negligible; and last, the extra space in the application container comes from the container space which includes the OS and software dependencies. The identification of libraries, software dependencies, and OS by the application container ensures replicability and transparency of the software system by guaranteeing that the user will always have the same versions of OS, libraries, and software dependencies, regardless of the platform on which the workflow is executed.

5.4 IO Bandwidth

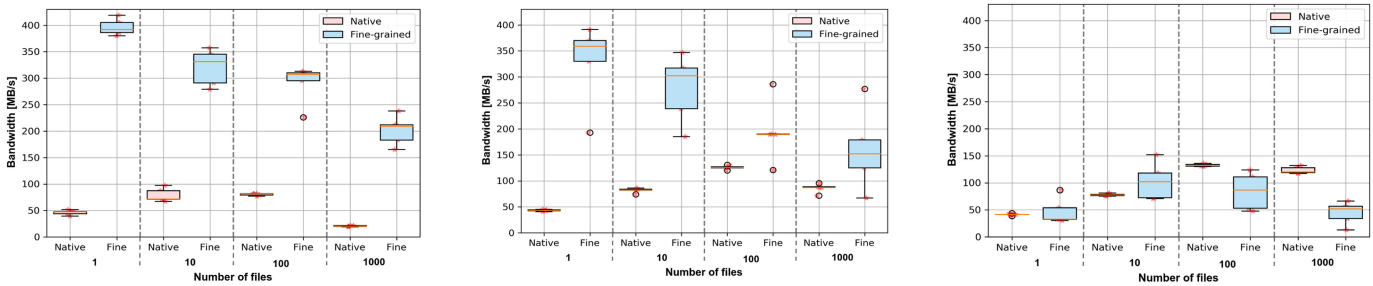
We benchmark the IO bandwidth for the two environments at each end of our testing spectrum (i.e., native and fine-grained) using FIO, a Flexible IO tester. We do not present results for the coarse-grained because its bandwidth is similar to the native, as shown in [38]. In the native environment, we measure the bandwidth of the benchmark to read from and write to the virtual volume on Jetstream2. In the fine-grained containerized environment, we measure the bandwidth of the benchmark encapsulated in an application container to read from and write to a data container stored in the virtual volume. The raw size of the virtual volume is 100 GB. Because of the virtual volume file system overhead (Ext4), the virtual volume is 90 GB. The storage for the container adds additional overhead, thus the data container has 80 GB of usable capacity.

FIO has the flexibility to select different IO settings including number of files, IO size, and sequential or random reads and writes. FIO spawns a number of files on a particular location doing a type of IO action. The location and the type of IO are specified by the user. We select four numbers of files (i.e., 1, 10, 100, and 1000) and three IO sizes (i.e., 1 GB, 10 GB, and 80 GB) for each IO size. Depending on the IO size and number of files, each test has a different configuration. They are as follows:

- 1 file of size 1 GB, 10 GB, and 80 GB
- 10 files of size 0.1 GB (100 MB), 1 GB, and 8 GB each
- 100 files of size 0.01 GB (10 MB), 0.1 GB (100 MB), and 1 GB each
- 1000 files of size 0.001 GB (1 MB), 0.01 GB (10 MB), and 0.08 GB (80 MB) each

For all the listed tests, we set the block size equal to 4KB (the default block size for the Ext3 and Ext4 file systems). We select a sequential mix of read and write IO patterns, mimicking the IO pattern of SOMOSPIE.

Fig. 12 shows the results of the bandwidth for the three IO sizes: (a) 1 GB, (b) 10 GB, and (c) 80 GB. The x-axis shows the number of files [1, 10, 100, 1000] and the y-axis shows the measured bandwidth in MB/s. For each number of files we measure the read and write IO 5 times for both environments. The measurements are represented in boxplots where the native measurements are in pink and the fine-grained containerized ones are in blue. We can extract two main observations from this figure. First, we observe that, as the number of files increases from 1 file to 100 files, the bandwidth of the native environment also increases, ranging from 45 MB/s to 140 MB/s. This trend is observed for all IO sizes. For the larger number of files (i.e., 1000 files), the bandwidth drops for the native environment. The bandwidth is measured from the start of the metadata operation to the end of the data transfer. Thus, while the metadata operation time per file is fixed, the IO time grows proportionally with the data size. In other words, when the files are many and small, the metadata operation time significantly impacts the IO time. Second, we compare IO bandwidth when the data fits in the DRAM (in Figs. 12a and 12b) versus when it does not (in Fig. 12c). In the figures, we observe that for 1 GB and 10 GB IO sizes our fine-grained containerized environment has higher IO performance than native. As we reach 80 GB in IO size, the performance for



(a) IO size of **1 GB** (b) IO size of **10 GB** (c) IO size of **80 GB**

Fig. 12. Bandwidth comparison for the native (no containers) and the fine-grained containerized (multiple containers) environments for different file counts and sizes.

both environments are comparable. The higher performance is due to the fact that the data container fits in the DRAM, allowing faster data access. This is not the case for the larger IO size for which the container must interact with the virtual volume regularly, thus degrading performance back to the native implementation. When running this test we deal with two main constraints. First, the size of the DRAM (60 GB) is fixed and defined by the Jetstream2 virtual machine. Second, users cannot control how the DRAM is managed. It is the kernel that manages the loading of the containers' content to the DRAM based on the availability of the resources. A modification of the kernel goes beyond the scope of our work.

6 Interoperability, Heterogeneity, and Multiple Instances

We discuss our environment's interoperability with other workflows and with resource managers, as well as the adaptation of our environment in distributed heterogeneous systems and with multiple container instances.

6.1 Environment Interoperability

Technology transfer, in which we envision the use of our environment for other applications, has been a key driving factor of our design. Our containerization approach requires that workflows are composed of one or multiple self-contained applications. Furthermore, users should be able to model the workflows as DAGs whose nodes (applications/tasks) and vertices (data in movement from one task to another) can be containerized. Any workflow with such features can be abstracted into a fine-grained set of interconnected containers, making our approach application-agnostic. We extend the Singularity/Apptainer runtime to support the concept of automatic creation and execution of fine-grained workflows that can be described as DAGs. We implement the Singularity/Apptainer plugin described in Section 3 which, given a workflow, has the ability to automatically create a fine-grained sequence of data and application containers, as well as the ability to execute these workflows while also annotating containers with execution metadata.

The integration of our environment into resource managers and orchestrators is possible as long as they manage containerized executions. Such containerized executions are supported by HPC and cloud solutions such as Pegasus [39], REANA [10], Pachyderm [9], and Kubernetes [40]. Because data is containerized, the workflow manager does not have to deal with data transfer from-to local storage, adding additional portability for our containerized workflow across platforms.

6.2 Distributed Heterogeneous Resources

Decomposing the workflow into fine-grained containers enables deploying different components on different nodes in a distributed system as long as the system shares storage (e.g., GPFS, object or block storage solutions). The bind mount across nodes can work through the shared storage connecting the nodes. For example, when using a cloud platform with object storage solution, an application container on a node can write to a data container on a different node; the content of the data container can be read by a second application container on the same node or a different node, using Kubernetes as the orchestrator of the containers' execution. With the increase of GPU usage for ML-based scientific workflows, container technologies such as Docker, Singularity/Apptainer, and Podman support the execution of applications in the scientific workflows on GPUs. Specifically, for our containerized environment, Singularity/Apptainer supports running application containers that use NVIDIA's CUDA GPU compute framework or AMD's ROCm solution. Regardless of the operative system on the host machine, users can run GPU-enabled ML frameworks (e.g., Tensorflow, MXNet, PyTorch). Regarding the data generated by the GPU, we assume that this data is copied back to the CPU, given that GPU-direct is not a widely available technology in most HPC and cloud platforms. When data is copied back to the CPU, our fine-grained containerized environment encapsulates it in a data container and automatically collects the data lineage.

6.3 Metadata from Multiple Container Instances

Our design supports multiple container instances, in which given an application container, we execute n instances of its image as a service. The application container image has a universally unique identifier (`UUID`). The n application instances inherit the same `UUID`, and through the `UUID` they link to the metadata of the image. When running multiple instances of the same application image, the metadata of the application image is not impacted but the metadata of the output data containers is. Such an impact changes based on where the multiple application instances write to. The n instances can write to their own output data container(s) or

to a single, shared container(s). In the first case, the n application instances are writing to their own output data container(s) (i.e., each application instance generates one output data container). For each output data container, the container's identification, creation time, and execution task are unique to the output data container. The record trail is the same across output data containers and links back to the `UUID` of the application image. In the second case, the n application instances are writing to a shared output data container(s) (i.e., the application instances generate one single output data container). For the shared output data container, the container's identification and creation time are unique to the output data container. The execution task is a list that contains n execution tasks including initial conditions, random seeds, and other setting values of the n instances. The single record trail links back to the `UUID` of the application image.

7 Related Work

Annotating a workflow execution with the provenance of its components has been previously used for traceability, reproducibility, and explaining results. Related work for collecting and preserving the provenance of a workflow at the system level include developing custom file systems tracking provenance such as the Lineage File System (LinFS) [11], PASTA in PASS (Provenance-Aware Storage Systems) [8], and Parrot [12]; encapsulating workflows through ad hoc packages such as CDE [13], ReproZip [14], Umbrella [15], and Occam [16] [17]; and encapsulating workflows through existing container technologies such as Pachyderm [9], REANA [10], and Science Capsule [41]. The use of custom file systems and custom ad hoc packages limits the portability and usability of their solutions across systems. The use of existing container technology to encapsulate the workflows overcomes this challenge, offering portable solutions. Contrary to existing containerization solutions, our approach decouples workflows into components (data and applications) at a finer level and maps the components to one-to-one single and independent containers.

Containerizing data facilitates transportation, interpretation, and use. We adapt the premise of the data containerization from Data Pallets [29]. It defines storage as a new container type when running workflows, where the containers include the data and links to the application and the input deck. Even when data is containerized, intermediate data is treated as disposable for solutions like Prune [42], CDE [13], ReproZip [14], Umbrella [15], and Occam [16][17] where they focus on sharing final results of the workflow executions. Only Pachyderm provides a complete audit trail for all data across pipeline stages, including intermediate results. As with Pachyderm, our solution grants first class citizen access to the intermediate data and its metadata. Moreover, we are the first to permanently and portably attach the provenance invisibly to the data and the applications. We achieve this through the use of a second partition in the container structure.

Finally, there has been a significant amount of work in workflow management that targets provenance. Workflow management systems like Pegasus [39], Kepler [43] and DAGMan [44] provide a way to orchestrate workflows while capturing the data provenance. Only Pegasus provides application containers as a solution to package software with complex dependencies. Pegasus currently supports Docker, Singularity/Apptainer, and Shifter. However, the workflow managers use the scientific workflow system to track and store the computational steps and their data dependencies, but information about the environment is rarely gathered. Furthermore, integrating a workflow to the management systems can be complex. It requires the translation of the workflow into the right format: DAX (Directed Acyclic Graph in XML) for Pegasus, XML or KAR files for Kepler, and DAG input file for DAGMan, for example. Finally, not all workflows are in the stage of managing their pipelines with these workflow tools, which makes a case for alternative solutions for hosting workflows and capturing provenance to allow traceability of data and explainability of results. Our work is intended to work together with these systems, using their support for containerized workflows to handle the workflow management tasks while we manage the automatic provenance collection, enabling traceability and explainability and annotation of data containers.

8 Conclusion

In this paper, we present a fine-grained containerized environment using Singularity/Apptainer technology that enables scientists to achieve trust in findings from their workflows by seamlessly providing data traceability and results explainability. We demonstrate the benefits of our environment for SOMOSPIE, an earth science workflow that uses ML-methods to predict satellite soil moisture data to a resolution necessary for policy making and precision agriculture. Specifically, we use our environment for two use cases in which we trace back differences in predictions due to input data and ML-methods. When compared with native and coarse-grained containerized environments, we observe that our environment has limited overhead in terms of time (10%), storage space (5% for data containers and 30% application container), and it has significantly higher IO bandwidth, with a peak of 400 MB/s versus 50 MB/s for native. Our solution is effective for establishing trustworthiness in scientific findings. Future work includes the automatic orchestration of the workflow in our containerized environment and the creation of a catalogue of containers that scientists can extend, share, and use to build new scientific workflows in multiple domains.

Code Availability

The code implementing the augmented Singularity/Apptainer can be found at: https://github.com/TauferLab/ContainerizedEnv.

Acknowledgments

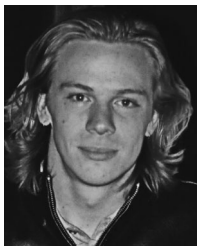
The authors acknowledge the Singularity/Apptainer team, specially Cedric Clerget and Ian Kaneshiro, for their support.

References

[1] R. Souza et al., "Workflow provenance in the lifecycle of scientific machine learning," *Concurrency Comput.: Pract. Experience*, vol. 1, pp. 1–19, 2021.

[2] L. Moreau et al., "Special Issue: The First Provenance Challenge," *Concurrency Comput.: Pract. Experience*, vol. 20, no. 5, pp. 409–418, 2008.
[3] P. Buneman and W.-C. Tan, "Data provenance: What next?," *SIGMOD Rec.*, vol. 47, no. 3, pp. 5–16, 2019.
[4] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014, Art. no. 2.
[5] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLoS One*, vol. 12, 2017, Art. no. e0177459.
[6] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in HPC," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 1–10.
[7] H. Gantikow, S. Walter, and C. Reich, "Rootless containers with podman for HPC," in *Proc. High Perform. Comput.: ISC High Perform. Int. Workshops*, 2020, pp. 343–354.
[8] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2006, pp. 1–4.
[9] P. Inc., "Pachyderm: The data foundation for machine learning." Accessed: Oct. 04, 2021. [Online]. Available: https://www.pachyderm.com
[10] T. Simko, L. Heinrich, H. Hirvonsalo, D. Kousidis, and D. Rodríguez, "REANA: A system for reusable research data analyses," in *Proc. EPJ Web Conf.*, 2019, Art. no. 06034.
[11] C. Sar and P. Cao, "Lineage file system." Accessed: Oct. 04, 2021. [Online]. Available: https://crypto.stanford.edu/cao/lineage
[12] D. Thain and M. Livny, "Parrot: An application environment for data-intensive computing," *Scalable Comput.: Pract. Experience*, vol. 6, pp. 9–18, 2005.
[13] P. J. Guo and D. Engler, "CDE: Using system call interposition to automatically create portable software packages," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 1–2.
[14] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational reproducibility with ease," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 2085–2088.
[15] H. Meng and D. Thain, "Facilitating the reproducibility of scientific workflows with execution environment specifications," *Procedia Comput. Sci.*, vol. 108, pp. 705–714, 2017.
[16] L. Oliveira, D. Wilkinson, D. Mossé, and B. R. Childers, "Occam: Software environment for creating reproducible research," in *Proc. IEEE 14th Int. Conf. E-Sci.*, 2018, pp. 394–395.
[17] D. Wilkinson, L. Oliveira, D. Mossé, and B. Childers, "Software provenance: Track the reality not the virtual machine," in *Proc. 1st Int. Workshop Practical Reproducible Eval. Comput. Syst.*, 2018, pp. 1–6.
[18] J. M. Diaz, A. Landwehr, and M. Taufer, "Dynamic CPU resource allocation in containerized cloud environments," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 535–536.
[19] S. Herbein et al., "Resource management for running HPC applications in container clouds," in *Proc. 31st High Perform. Comput. Int. Supercomputing Conf.*, 2016, pp. 261–278.
[20] S. McDaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in docker containers," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 490–491.
[21] A. Dusia, Y. Yang, and M. Taufer, "Network quality of service in docker containers," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 527–528.
[22] D. Rorabaugh, M. Guevara, R. Llamas, J. Kitson, R. Vargas, and M. Taufer, "SOMOSPIE: A modular soil moisture spatial inference engine based on data-driven decisions," in *Proc. 15th Int. Conf. ESci.*, 2019, pp. 1–10.
[23] Soil moisture CCI. Accessed: Oct. 04, 2021. [Online]. Available: https://www.esa-soilmoisture-cci.org
[24] R. M. Llamas, M. Guevara, D. Rorabaugh, M. Taufer, and R. Vargas, "Spatial gap-filling of ESA CCI satellite-derived soil moisture based on geostatistical techniques and multiple regression," *Remote. Sens.*, vol. 12, no. 4, 2020, Art. no. 665.
[25] M. Guevara, M. Taufer, and R. Vargas, "Gap-free global annual soil moisture: 15 km grids for 1991–2018," *Earth Syst. Sci. Data*, vol. 13, no. 4, pp. 1711–1735, 2021.
[26] K. M. D. Sweeney and D. Thain, "Efficient integration of containers into scientific workflows," in *Proc. 9th Workshop Sci. Cloud Comput.*, 2018, pp. 1–6.
[27] T. Renner, L. Thamsen, and O. Kao, "CoLoc: Distributed data and container colocation for data-intensive applications," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 3008–3015.
[28] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A containerized big data streaming architecture for edge cloud computing on clustered single-board devices," *Closer*, pp. 68–80, 2019.
[29] J. Lofstead, J. Baker, and A. Younge, "Data pallets: Containerizing storage for reproducibility and traceability," in *Proc. Int. Conf. High Perform. Comput.*, 2019, pp. 36–45.
[30] T. Kluyver et al., "Jupyter notebooks – a publishing format for reproducible computational workflows," *Positioning and Power in Academic Publishing: Players, Agents and Agendas.*, Amsterdam, Netherlands: IOS Press, 2016, pp. 87–90.
[31] C. Clerget and I. Kaneshiro, "Singularity image bind support." Accessed: Aug. 20, 2022. [Online]. Available: https://github.com/apptainer/singularity/pull/4948
[32] P. Olaya and D. Kennedy, "Containerized environment." Accessed: Aug. 20, 2022. [Online]. Available: https://github.com/TauferLab/ContainerizedEnv
[33] D. Kennedy, P. Olaya, J. Lofstead, R. Vargas, and M. Taufer, "Augmenting singularity to generate fine-grained workflows, record trails, and data provenance," in *Proc. IEEE 18th Int. Conf. E-Sci.*, 2022, pp. 1–2.
[34] R. P. Paul Walsh, "NetworkX viewer." Accessed: Oct. 11, 2021. [Online]. Available: https://github.com/jsexauer/networkx_viewer
[35] NASA and Jet Propulsion Laboratory and PODAAC, "Soil moisture active passive (SMAP)." Accessed: Oct. 04, 2021. [Online]. Available: https://podaac.jpl.nasa.gov/SMAP
[36] D. Y. Hancock et al., "Jetstream2: Accelerating cloud computing via Jetstream," in *Proc. Pract. Experience Adv. Res. Comput.*, 2021, pp. 1–8.
[37] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, pp. 21–33, 2007.
[38] G. Hu, Y. Zhang, and W. Chen, "Exploring the performance of singularity for high performance computing scenarios," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun.*, 2019, pp. 2587–2593.
[39] E. Deelman et al., "Pegasus: A workflow management system for science automation," *Future Gener. Comput. Syst.*, vol. 46, pp. 17–35, 2015.
[40] Kubernetes. Accessed: Aug. 20, 2022. [Online]. Available: https://github.com/kubernetes/kubernetes
[41] D. Ghoshal, L. Bianchi, A. Essiari, M. Beach, D. Paine, and L. Ramakrishnan, "Science capsule - capturing the data life cycle," *J. Open Source Softw.*, vol. 6, no. 62, 2021, Art. no. 2484.
[42] P. Ivie and D. Thain, "PRUNE: A preserving run environment for reproducible scientific computing," in *Proc. IEEE 12th Int. Conf. E-Sci.*, 2016, pp. 61–70.
[43] The kepler project. Accessed: Apr. 10, 2021. [Online]. Available: https://kepler-project.org
[44] G. Malewicz, I. Foster, A. L. Rosenberg, and M. Wilde, "A tool for prioritizing DAGMan jobs and its evaluation," *J. Grid Comput.*, vol. 5, no. 2, pp. 197–212, 2007.

Paula Olaya received the MS degree in computational science from the University of Tennessee, Knoxville. She is currently working toward the PhD degree in computer science with the University of Tennessee. Her research interests include the convergence between high-performance and cloud computing for scientific applications.



Dominic Kennedy is currently working toward the BS degree in computer science with the University of Tennessee, Knoxville. After completing a cybersecurity internship with Cisco systems, he moved into his role as a research assistant with the Global Computing Lab led by Dr. Michela Taufer.

Ricardo Llamas received the BS degree in geography from the National Autonomous University of Mexico, in 2010. He is currently working toward the PhD degree in water science and policy with the University of Delaware under Dr. Rodrigo Vargas. His research interests include soil moisture modeling across different spatial scales based on remotely sensed data and the use of machine learning and geostatistics.

Leobardo Valera received the PhD degree in computational science from the University of Texas with El Paso. He is currently a postdoc research with the University of Tennessee, Knoxville. His research interests include reliable computation, machine learning, and quantum computing.

Rodrigo Vargas received the PhD degree in environmental sciences from the University of California-Riverside, in 2007. He is a professor of ecosystem ecology and environmental change in the department of Plant and Soil Sciences, University of Delaware. His research interests include data science for environmental applications, design of environmental networks, and climate change.

Jay Lofstead (Senior Member, IEEE) received the PhD degree in computer science from the Georgia Institute of Technology, in 2010. He is the Principal Member of Technical Staff with Sandia National Laboratories. His research interests focus around large scale data management and trusted scientific computing. In particular, he works on storage, IO, metadata, workflows, reproducibility, software engineering, machine learning, and operating system-level support for any of these topics.

Michela Taufer (Senior Member, IEEE) received the PhD degree in computer science from the Swiss Federal Institute of Technology (ETH), in 2002. She is an ACM Distinguished Scientist and holds the Jack Dongarra professorship in high performance computing with the Department of Electrical Engineering and Computer Science, University of Tennessee Knoxville. Her research interests include high-performance computing, volunteer computing, scientific applications, scheduling and reproducibility challenges, and in situ data analytics.