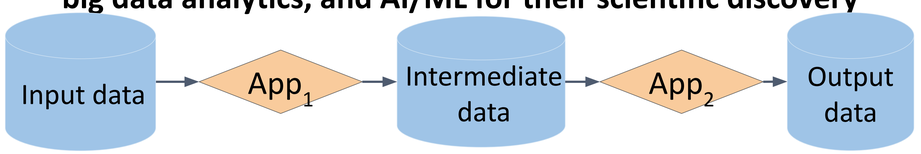
Enabling Reproducibility and Scalability of Scientific Workflows in HPC and Cloud

Paula Olaya[1] and Michela Taufer[1] (Advisor)
University of Tennessee, Knoxville

Scientific Workflows' Complexity

The scientific community relies on the execution of complex workflows that sit at the intersection of HPC, cloud computing big data analytics, and AI/ML for their scientific discovery



Workflows include many interoperable components (data and applications) that are hard to trace and reuse to reproduce results and integrate AI/ML methods with limited transparency

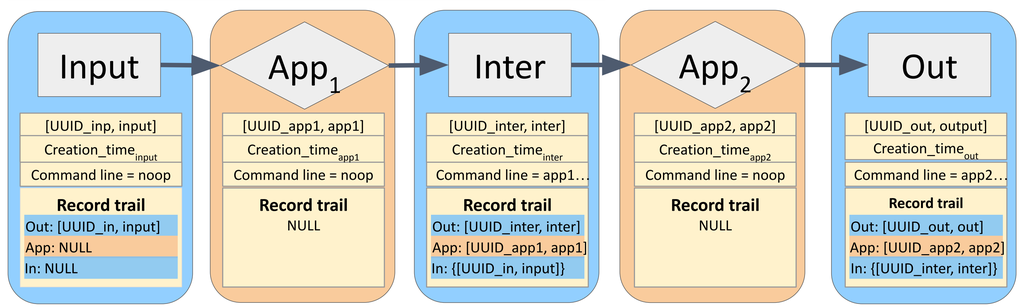
Workflows hide the complexity of large intermediate data and their overall execution can be affected by the I/O bandwidth of the underlying infrastructure

Workflows run on heterogeneous and distributed infrastructure with data and application dependencies that require efficient data management and resource allocation

Challenge 1: Traceability and Explainability of Scientific Workflows

Scientists need execution environments that automatically **trace data provenance and explain results** through an in-depth data lineage and an execution trail

We create **a fine-grained containerized environment that enables data traceability and results explainability:** Our environment automatically creates a record trail and data lineage of a workflow execution and seamlessly attaches it to the workflow components



A **data container** follows a file-system-in-a-file model and includes an individual dataset (i.e., input, intermediary, or output data)

The **application container** includes the executable or script with the respective software stack (i.e., OS, libraries, and packages)

The **provenance metadata** exposes unique hash code (UUID), container name, creation time, command line and record trail

1. We **decouple data and applications** of traditionally tightly-coupled workflows and **encapsulate them into individual fine-grained containers**
2. We **augment all containers** to move data across the containerized workflow effectively and to **expose provenance metadata**
3. We provide an **interface** to visualize and **study the metadata** so scientists understand the data lineage and the computational methods

Challenge 2: Scalability of Scientific Workflows

Scientists need an infrastructure that **efficiently writes and reads large intermediate data** and **automatically scales** their scientific workflows' execution

We **leverage cloud technology to integrate scientific workflows** in cloud-based HPC services (LSF and Kubernetes) using Cloud Object Storage, **enabling better I/O and data scalability**

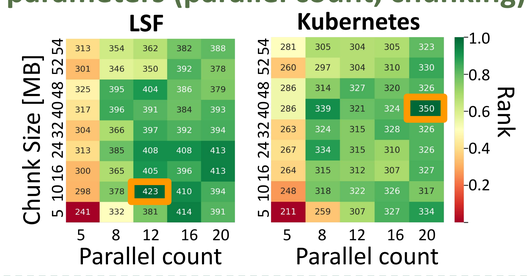
Cloud Object Storage (COS)
Data stored as objects in buckets

S3FS
Map COS into POSIX namespaces

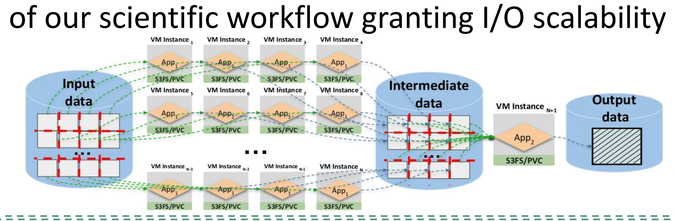
LSF
HPC as a service on the cloud

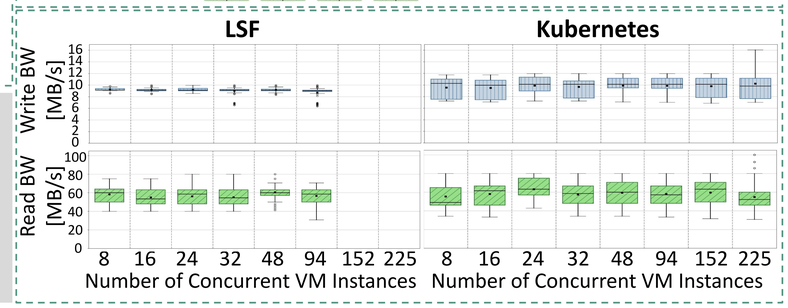
Kubernetes
Cloud-native service

We tune the advanced **S3FS's I/O parameters (parallel count, chunking)**



We **map our infrastructure** to the parallel data nature of our scientific workflow granting I/O scalability



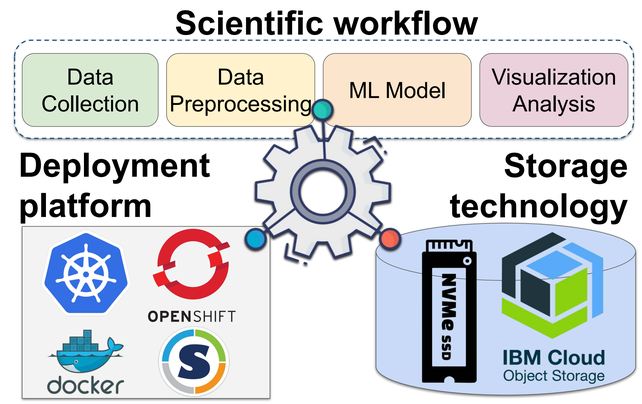


We observe **no I/O performance degradation in the object storage as we increase the number of VM instances** of writing and reading in parallel for LSF and Kubernetes (K8s)

Challenge 3: Orchestration of Scientific Workflows

Scientists require tools that enable **iterative development, orchestration, and deployment of end-to-end scientific workflows**

We use Open Data Hub Pipelines to **orchestrate the end-to-end execution of workflows** in cloud-native clusters (i.e., Kubernetes and OpenShift) with Cloud Object Storage. We ensure i) automated orchestration of the workflows, ii) efficient allocation of infrastructure resources, and iii) reproducibility and reusability of workflows' executions



User defines the pipeline using Kubeflow Pipelines DSL in Python and compiles it to Tekton

Our orchestration ensures efficient intermediate data allocation on NVMe and performance monitoring using Grafana

```
def app1():
  ...
def app2(model):
  ...

dsl.pipeline(workflow):
  for i in N:
    model=app1()
    app2(model)

TektonCompiler().
compile(pipeline,
"pipeline.yaml")
```