2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC
(CANOPIE-HPC)

Enabling Seamless Execution of Computational and Data Science Workflows on HPC and Cloud with the Popper Container-native Automation Engine

Jayjeet Chakraborty
UC Santa Cruz
orcid.org/0000-0003-1647-2494

Carlos Maltzahn
UC Santa Cruz
orcid.org/0000-0001-8305-0748

Ivo Jimenez
UC Santa Cruz
orcid.org/0000-0002-2222-1985

***Abstract*—The problem of reproducibility and replication in scientific research is quite prevalent to date. Researchers working in fields of computational science often find it difficult to reproduce experiments from artifacts like code, data, diagrams, and results which are left behind by the previous researchers. The code developed on one machine often fails to run on other machines due to differences in hardware architecture, OS, software dependencies, among others. This is accompanied by the difficulty in understanding how artifacts are organized, as well as in using them in the correct order. Software containers (also known as Linux containers) can be used to address some of these problems, and thus researchers and developers have built scientific workflow engines that execute the steps of a workflow in separate containers. Existing container-native workflow engines assume the availability of infrastructure deployed in the cloud or HPC centers. In this paper, we present Popper, a container-native workflow engine that does not assume the presence of a Kubernetes cluster or any service other than a container engine such as Docker or Podman. We introduce the design and architecture of Popper and describe how it abstracts away the complexity of multiple container engines and resource managers, enabling users to focus only on writing workflow logic. With Popper, researchers can build and validate workflows easily in almost any environment of their choice including local machines, Slurm based HPC clusters, CI services, or Kubernetes based cloud computing environments. To exemplify the suitability of this workflow engine, we present a case study where we take an example from machine learning and seamlessly execute it in multiple environments by implementing a Popper workflow for it.**

I. Introduction

Researchers working in various domains related to computational and data-intensive science upload experimental artifacts like code, figures, datasets, and configuration files, to open-access repositories like Zenodo [1], Figshare [2], or GitHub [3]. According to [4], approximately 1% of the artifacts available online are fully reproducible and 0.6% of them are partially reproducible. A 2016 study by Nature found that from a group of 1576 scientists, around 70% of them failed to reproduce each other's experiments [5]. This problem occurs mostly due to the lack of proper documentation, missing artifacts, or encountering broken software dependencies. This results in other researchers wasting time trying to figure out how to reproduce those experiments from the archived artifacts, ultimately making this process inefficient, cumbersome, and error-prone [6].

Numerous existing research has tried to address the problem of reproducibility [7] by different means; for example, logging and tracing system calls, using workflow engines, using correctly provisioned shared and public testbeds, by recording and replaying changes from a stable initial state, among many others [8]. These approaches have led to the development of various tools and frameworks to address these problems of reproducibility [9,10], with scientific workflow engines being a predominant one [11–13]. A workflow engine organizes the steps of a scientific experiment as the nodes of a directed acyclic graph (DAG) and executes them in the correct order [14]. Nextflow [15], Pegasus [16] and Taverna [17] are examples of widely used scientific workflow engines. But some phenomena like unavailability of third-party services, missing example input data, changes in the execution environment, insufficient documentation of workflows make it difficult for scientists to reuse workflows, resulting in *workflow decay* [18].

One of the main reasons behind *workflow decay* is the difficulty in reproducing the environment where a workflow is developed and originally executed [19]. Virtual machines (VM's) can be used to address this problem, as its isolation guarantees make it suitable for running steps or the entirety of a workflow inside a separate VM [20,21]. A VM is typically associated with large resource utilization (e.g. long start times and high memory usage), making OS-level virtualization technologies a better-suited tool for reproducing computational environments with fewer overheads [22,23]. Although software (Linux) containers are a relatively old technology [24], it was not until recently, with the rise of Docker, that they entered mainstream territory [25].

Docker has been a popular container runtime for a long time, with other container runtimes such as Singularity [26], Rkt [27], Charliecloud [28], and Podman [29] having emerged. With containers, the container-native software development paradigm emerged, which promotes the building, testing, and deployment of software in containers, simultaneously giving rise to the practice of running scientific experiments inside containers to make them platform independent and reproducible

978-0-7381-3062-0/20/$31.00 ©2020 IEEE
DOI 10.1109/CANOPIEHPC51917.2020.00007

[30–32]. Differences among container engines stem from the need to serve distinct use cases, manifesting in user experience (UX) differences such as those found in their command-line interfaces (CLIs), container image formats; security requirement and environmental differences such as Podman for enhanced security and Singularity for its use in HPC. In practice, for users attempting to make use of container technology, these differences can be overwhelming, especially if they are only familiar with the basic concepts of how containers work. Based on our analysis of the container tooling landscape, we found that there is an absence of tools for allowing users to work with containers in an engine-agnostic way. It has also been found that as scientific workflows become increasingly complex, continuous validation of the workflows which is critical to ensuring good reproducibility, becomes difficult [33,34].

Different container-based workflow engines are available but most of them assume the presence of a fully provisioned Kubernetes cluster [35]. The presence of a Kubernetes cluster or a cloud computing environment reduces the likelihood of researchers to adopt container technology for reproducing any experiment since it is often costly [36] to get access to one and this, in turn, makes reproducibility complex. It would be more convenient for researchers if workflow engines provided the flexibility of running workflows in a wide range of computing environments including those that are readily available for them to use.

Popper[1] is a lightweight workflow execution engine that allows users to follow the container-native paradigm for building and running reproducible workflows from archived experimental artifacts. This paper makes the following contributions:

1. The design and architecture of a container-native workflow engine that abstracts over container engines, image builders, and resource managers, giving users the ability to focus on Dockerfiles (software dependencies) and workflow logic, without having to invest time in runtime specific details.
2. Popper, an implementation of the above design that allows running workflows inside containers in different computing environments like local machines, Kubernetes clusters, or HPC environments.
3. A case study on how Popper can be used to quickly reproduce complex workflows in different computing environments. We show how an entire machine learning workflow can be executed on a local machine during development, and how it can be scaled up by reproducing it in a Kubernetes cluster with nodes that have GPUs available in them. We also show how the same workflow can be reproduced easily in a Slurm [37] cluster in order to scale it up further.

[1]The version of Popper described in this article is a major overhaul of an earlier version of Popper. See Section V-D for more.

II. Popper

In this section, we describe the motivation behind Popper, provide background, and then introduce its architectural design and implementation.

A. Motivation

Let us take a relatively simple scenario where users have a list of single-purpose tasks in the form of scripts and they want to automate running them in containers in some sequence. To accomplish this goal of running a list of containerized tasks using existing workflow engines, users need to learn a specific workflow language, deploy a workflow engine service, and learn to execute workflows on that service. These tasks may not be always trivial to accomplish if we assume the only thing users should care about is writing experimentation scripts and running them inside containers. Assume we have three scripts `download_dataset.py`, `verify_dataset.sh`, and `run_training.sh` to download a dataset, verify its contents and run a computational step. In practice, when developers work following the container-native paradigm they end up interactively executing multiple Docker commands to build containers, compile code, test applications, or deploy software. Keeping track of which commands were executed, in which order, and which flags were passed to each, can quickly become unmanageable, difficult to document, error-prone, and hard to reproduce.

The goal of Popper is to bring order to this chaotic scenario by providing a framework for clearly and explicitly defining container-native tasks and to launch them, and track their completion. Running workflows on dissimilar environments like Kubernetes and Slurm incurs multiple operational overheads like adopting environment-specific commands, writing job scripts and definitions, dealing with different image formats like the flat image format of Singularity, which are peculiar to a specific computing environment. For example, running a containerized step on Kubernetes would require writing pod and volume specifications and creating them using a Kubernetes client. Likewise, running an MPI workload inside a Singularity container on Slurm would require creating job scripts and starting the job with `sbatch`. Popper mitigates these environment-specific overheads by abstracting the different implementation details and provides an uniform interface that allows users to write workflows once and reuse them on different environments with tweaks to the configuration file.

B. Design principles

The design of Popper is based on the following principles:

- **Fully embrace the container-native paradigm**. Every step of a workflow executes inside a container.
- **Dockerfile and OCI images as lowest common denominators**. All available container engines support generating Open Container Initiative (OCI) images from Dockerfiles,

as well as importing or exporting these to other formats; most support the Docker registry API as well. This common denominator opens the possibility to create an abstraction layer over runtimes, engines, builders, and orchestrators.
- **Be as lightweight as possible but not lighter**. The syntax for defining container-native workflow languages should be simple enough that it can be learned in seconds, and meant to be as close to shell scripts as possible.
- **Codify the workflow at the highest level**. Automate the workflow that we currently observe in the form of README files; documentation that explains how to compile, run, and test a software project; or artifact descriptions in academic articles describing how to reproduce results. In other words, the goal is *not* to replace existing, more comprehensive workflow engines such as scientific workflow engines, configuration management frameworks, or domain-specific experiment management tools; rather, the goal is to have a thin wrapper around them.

C. Background

In this subsection, we provide background on the different tools and technologies that Popper leverages in order to provide the ability of running engine- and resource manager-agnostic container-native workflows.

1) Docker: Docker is an OS-level virtualization technology that was released in early 2013. It uses various Linux kernel features like namespaces and cgroups [38] to segregate processes so that they can run independently. It provides state of the art isolation guarantees and makes it easy to build, deploy, and run applications using containers following the OCI (Open Container Initiative) [39] specifications. However, it was not designed for use in multi-user HPC environments and also has significant security issues [40], which might enable a user inside a Docker container to have root access to the host system's network and filesystem, thus making it unsuitable for use in HPC systems. Docker uses Linux's cgroups to isolate containers, which conflicts with the Slurm scheduler since it also uses cgroups to allocate resources to jobs and enforce limits [41].

2) Singularity: Singularity is a daemon-less scientific container technology built by LBNL (Lawrence Berkley National Laboratory) and first released in 2016. It is designed to be simple, fast, secure, and provides containerized solutions for HPC systems supporting several HPC components such as resource managers, job schedulers and contains native MPI [42] features. One of the main goals of Singularity is to bring container technology and reproducibility to the High-Performance Computing world. The key feature that differentiates it from Docker is that it can be used in non-privileged computing environments like the compute nodes of HPC clusters, without any modifications to the software. It also provides an abstraction that enables using container images from different image registries interchangeably like Docker Hub, Singularity Hub, and Sylabs Cloud. These features make Singularity increasingly useful in areas of Machine learning, Deep learning, and other data-intensive applications where the workloads benefit from HPC systems.

3) Slurm: Slurm is an open-source cluster resource management and job scheduling system developed by LLNL (Lawrence Livermore National Laboratory) for Linux clusters ranging from a few nodes to thousands of nodes. It is simple, scalable, portable, fault-tolerant, secure, and interconnect agnostic. It is used as a workload manager by almost 60% of the world's top 500 supercomputers [43]. Slurm provides a plugin-based mechanism for simplifying its use across different computing infrastructures. It enables both exclusive and non-exclusive allocation of resources like compute nodes to the users. It provides a framework for starting, executing, and monitoring parallel jobs on a set of allocated nodes and arbitrate conflicting requests for resources by managing a queue of pending work. Slurm runs as a daemon in the compute nodes and also provides an easy to use CLI interface.

4) Kubernetes: Kubernetes is a production-grade open-source container orchestration system written in Golang that automates many of the manual processes involved in deploying, scaling, and managing of containerized applications across a cluster of hosts. A cluster can span hosts across public, private, or hybrid clouds. This makes Kubernetes an ideal platform for hosting cloud-native applications. Kubernetes supports a wide range of container runtimes including Docker, Rkt, and Podman. It was originally developed and designed by engineers at Google and it is hosted and maintained by the CNCF (Cloud Native Computing Foundation). Many cloud providers like GCP, AWS, and Azure provide a completely managed and secure hosted Kubernetes platform.

5) Continuous integration: Continuous Integration (CI) is a software development paradigm where developers commit code into a shared repository frequently, ideally several times a day. Each integration is verified by automated builds and tests of the corresponding commits. This helps in detecting errors and anomalies quickly and shortens the debugging time [44]. Several hosted CI services like Travis [45], Circle [46], and Jenkins [47] make continuous integration and continuous validation easily accessible.

D. Workflow definition language

YAML [48] is a human-readable data-serialization language. It is commonly used in writing configuration files and in applications where data is stored or transmitted. Due to its simplicity and wide adoption [49], we chose YAML for defining Popper workflows and for specifying the configuration for the execution engine. An example Popper workflow is shown in Lst. 1 which downloads a dataset in CSV format and generates its transpose.

A Popper workflow consists of a series of syntactical components called steps, where each step represents a node in the

Listing 1 A two-step example Popper workflow.

```
steps:
# download CSV file with data on global CO2 emissions
- id: download
  uses: docker://byrnedo/alpine-curl:0.1.8
  args: [-L, https://git.io/JUcRU, -o, global.csv]

# obtain the transpose of the global CO2 emissions table
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -o, global_transposed.csv]
```

workflow DAG, with a `uses` attribute specifying the required container image. The `uses` attribute can reference Docker images hosted in container image registries; filesystem paths for locally defined container images (Dockerfiles); or publicly accessible GitHub repositories that contain Dockerfiles. The commands or scripts that need to be executed in a container can be defined by the `args` and `runs` attributes. Secrets and environment variables needed by a step can be specified by the `secrets` and `env` attributes respectively for making them available inside the container associated with a step. The steps in a workflow are executed sequentially in the order in which they are defined. As can be seen in Lst. 1, Popper workflows are extremely simple with a linear sequence of steps without any loops, conditionals, retries, or waits.

E. Workflow execution engine

The Popper workflow execution engine[2] is composed of several components that talk to each other during a workflow execution. The vital architectural components of the system are described in detail throughout this section. The architecture of the Popper workflow engine is shown in Fig. 1;

1) Command line interface (CLI): Besides allowing users to communicate with the workflow runner, the CLI generates configuration files for continuous integration systems such as Travis or Jenkins, so that users can continuously validate their workflows; provides dynamic workflow variable substitution capabilities, among others.

2) Workflow definition and configuration parsers: The workflow file and the configuration file are parsed by their respective parser plugins at the initial stages of the workflow execution. The parsers are responsible for reading and parsing the YAML files into an internal format; running syntactic and semantic validation checks; normalizing the various attributes and generating a workflow DAG. The workflow parser has a pluggable architecture that allows adding support to other workflow languages.

3) Workflow runner: The Workflow runner is in charge of taking a parsed workflow representation as input and executing it. It also downloads actions referenced by the steps in a workflow, checks the presence of secrets that are required by a workflow, and routes the execution of a step to the configured container engine through the requested resource manager. The runner also maintains a cache directory to optimize multiple aspects of execution such as avoid cloning repositories if they have been already cloned previously. Thus, this component orchestrates the entire workflow execution process.

[2]`https://github.com/getpopper/popper`

4) Resource manager and container engine API: Popper supports running containers in both single-node and multi-node cluster environments. Each of these different environments has a very specific job and process scheduling policies. The resource manager API is a pluggable interface that allows the creation of plugins (also referred to as runners) for distinct job schedulers (e.g. Slurm, HTCondor) and cluster managers (e.g. Kubernetes, Mesos [50]). Currently, plugins for Slurm and Kubernetes exist, as well as the default local runner that executes workflows on the local machine where Popper is executed. Resource manager plugins provide abstractions for different container engines which allows a particular resource manager to support new container engines through plugins. For example, in the case of Slurm, it currently supports running Docker and Singularity containers but other container engines can also be integrated like Charliecloud [51] and Pyxis [52]. The container engine plugins abstract generic operations that all engines support such as creating an image from a `Dockerfile`; downloading images from a registry and converting them to their internal format; and container-level operations such as creation, deletion, and renaming. Currently, there are plugins for Docker, Podman, and Singularity, with others planned by the Popper community.

The behavior of a resource manager and a container engine can be customized by passing specific configuration through the configuration file. This enables the users to take advantage of engine and resource manager specific features in a transparent way. In the presence of a `Dockerfile` and a workflow file, a workflow can be reproduced easily in different computing environments only by tweaking the configuration file. For example, a workflow developed on the local machine can be run on an HPC cluster using Singularity containers by specifying information like the available MPI library and the number of nodes and CPUs to use for running a job in the configuration file. The configuration file can be passed through the CLI interface and can be shared among different workflows. It can either be created by users or provided by system administrators.

5) Workflow exporter: Popper allows exporting a workflow to other workflow specification formats such as CWL [53] and WDL [54], as well as those associated with a CI service (e.g. Travis) or workflow engine (e.g. Airflow [55]). In most cases, the workflow specification syntax for these formats is more complex from that one of Popper's, mainly due to the fact that Popper workflow's syntax is fairly minimal and high-level, so it is always the case that a Popper workflow can be written in another existing format that supports containerized workflows. This prevents lock-in of workflows by Popper as workflows that are written initially for Popper can be exported to other formats and executed on other workflow engines or CI tools. Exporting to other formats is handled by an extensible Workflow Exporter module that allows creating exporters for an arbitrary list of workflow specification formats. Currently,

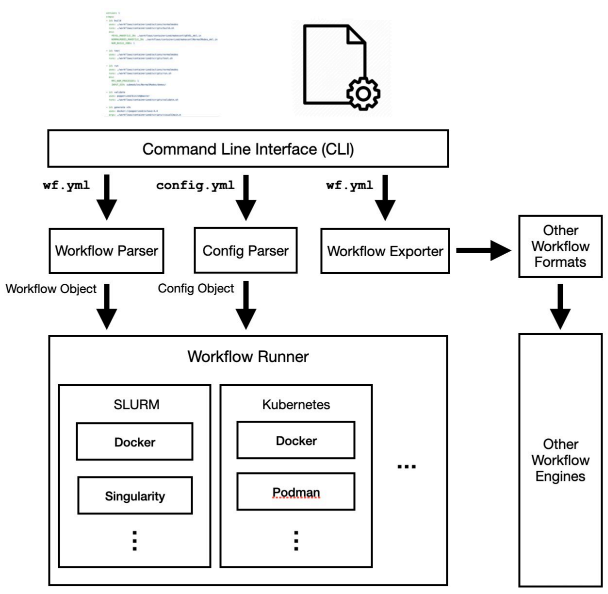

Figure 1: Architecture of the Popper workflow engine

plugins for CI services like TravisCI, CircleCI, Jenkins, and Gitlab-CI are implemented.

III. Case study

In this section, we present a case study demonstrating how the Popper workflow engine allows reproducing and scaling workflows in different computing environments. This study aims to emphasize on how Popper can help in mitigating the reproducibility issues and make life easier for researchers and developers.

The high-level procedure for building reproducible workflows with Popper usually consists of the following steps:

1. Thinking of the logical steps of the workflow.
2. Finding the relevant software packages required for the implementation of these steps.
   - Finding images containing the required software from remote image registries such as DockerHub.
   - If a prebuilt image is not available, a `Dockerfile`, which is a file containing specifications for building Docker images, can be used to build an image manually.
3. Running the workflow and refining it.

For this case study, we built an image classification workflow that trains a model using Horovod [56] and Keras [57] over the MNIST [58] dataset. The workflow consists of three steps: the `download-dataset` step downloads the MNIST dataset in the workspace; the `verify-dataset` step verifies the downloaded archives against precomputed checksums; the `run-training` step then starts training the model on this downloaded dataset and records the duration of the training.

Listing 2 Workflow used in the case study.

```
steps:
# download the MNIST dataset
- id: download-dataset
  uses: docker://horovod/horovod:0.19.3-tf2.1.0-torch-mxnet1.6.0-py3.6-gpu
  runs: [python]
  args: [./workflows/mnist/scripts/download_dataset.py]

# verify the dataset against checksums
- id: verify-dataset
  uses: docker://alpine:3.9.5
  args: [./workflows/mnist/scripts/verify_dataset.sh]

# run training on the dataset
- id: run-training
  uses: docker://horovod/horovod:0.19.3-tf2.1.0-torch-mxnet1.6.0-py3.6-gpu
  args: [./workflows/mnist/scripts/run_training.sh]
  env:
    NUM_EPOCHS: '1'
    BATCH_SIZE: '128'
    DATASET_REDUCTION: '0.1'
```

The workflow used for the case study is depicted in Lst. 2. The download and train steps use a Horovod image and the verify step uses a lightweight alpine image. Although a single Docker image can be used in all the steps of a workflow, we recommend using images specific to the purpose of a step; doing otherwise eventually complicates dependency management, hence defeating the purpose of containers. The code that the workflow references can be found in the repository[3] associated with this paper.

A. *Workflow execution on the local machine*

Popper aids researchers in writing, testing, and debugging workflows on their local development machines. Researchers can iterate quickly by making changes and executing the `popper run` command to see the effect of their changes immediately. In our case, we used a laptop with a 2.4GHz quad-core Intel Core i5 64-bit processor and 8GB of LPDDR3 RAM. The image classification workflow was executed against the MNIST dataset using Docker as the container engine. On single node machines, Popper leaves the job of scheduling the containerized steps to the host machines OS. We ran the workflow 5 times with an overfitting patience of 5 on the laptop's CPU. To reduce the runtime of this training ML workflow, it can be executed on machines with GPUs available to them. An alternative for this is to make use of the cloud, which in turn requires the workflow to be ported to this environment.

B. *Workflow execution in the cloud using Kubernetes*

On Kubernetes, steps of a Popper workflow run in separate pods that can get scheduled on any node of the cluster in a separate namespace. Popper first builds the images required by the workflow and pushes them to an online image registry. Then a `PersistentVolumeClaim` is created to claim persistent storage space from a shared filesystem like NFS [59] for the different step pods to share and the workflow context (scripts, configuration files, etc.) is copied into it. Finally, for each step of the workflow, a pod is created that binds to the shared volume and the corresponding scripts or commands are executed inside the pod.

Listing 3 Configuration file for running on Kubernetes.

```
resource_manager:
  name: kubernetes
  options:
    volume_size: 4Gi
    namespace: mynamespace
```

Listing 4 Configuration file for running on Slurm.

```
engine:
  name: singularity
  options:
    bind: [/path/to/mpi/lib:/usr/lib/,
           /path/to/mpi/bin:/usr/bin/]

resource_manager:
  name: slurm
  options:
    run-training:
      nodes: 2
      nodelist: worker1,worker2
      cpus-per-task: 2
```

Although any Kubernetes cluster can be used, for this case, we used a 3-node Kubernetes cluster on CloudLab [60] each with an NVIDIA 12GB PCI P100 GPU. The training pod used the single GPU of the node on which it was scheduled. Reproducing a locally developed workflow on a Kubernetes cluster only requires changing the resource manager specifications in the configuration file like specifying Kubernetes as the requested resource manager, specifying the `PersistentVolumeClaim` size and the image registry credentials. In our case, we configured the training with an overfitting patience of 5, similar to what was done for the local machine execution. The training can be speed up further by scheduling the workflow on multiple nodes to utilize the processing power of multiple GPUs.

C. *Workflow execution in Slurm clusters*

For running workflows on Slurm clusters, container engines that are HPC-aware, like Singularity, Charliecloud, and Pyxis need to be used. Currently, Popper supports running MPI workloads only through Slurm using Singularity as the container engine. To run workflows on a Slurm cluster, a configuration file containing Slurm specific options like the number of CPUs or nodes to use for running a job needs to be supplied to the `popper run` command. The user should be aware of the availability of resources like the nodes, CPUs, and GPUs on a shared cluster to write the Popper configuration file accordingly.

Popper's Slurm runner builds or pulls an image on the login node of a Slurm cluster and then starts executing the containerized MPI workload on the compute nodes using the `sbatch` command, which creates and schedules a job for running the container on Slurm. With Singularity as the container engine, both the hybrid and bind method can be used to run a Singularity container with MPI available inside it. Using the bind approach provides the benefit of making the same MPI based Singularity image compatible between Slurm clusters with different MPI versions.

For this case study, we used 3 VMs from Azure each with an NVIDIA 12GB PCI P100 GPU running Ubuntu 18.04 to create a Slurm cluster with 1 login and 2 compute nodes. We used `mpich` which is a popular implementation of MPI, with Singularity following the bind approach, where we install MPI on the host and then bind mount the `/path/to/mpi/bin` and `/path/to/mpi/lib` of the MPI package inside the Singularity container for the MPI version in the host and the container to stay consistent. The training step was executed on 2 compute nodes having a GPU each and the training parameters were the same as in the previous executions.

[3]`https://github.com/ivotron/popper-canopie-paper`

Listing 5 Travis configuration generated by Popper.

```
dist: "xenial"
language: "python"
python: "3.8"
services: "docker"
install: "pip install popper"
script: "popper run -f wf.yml"
```

D. *Workflow execution on CI*

We used the workflow exporter to generate a Travis configuration file, pushed our MNIST project to GitHub with the configuration, and activated the repository on Travis to run our workflows on CI. For long-running workflows like those consisting of ML/AI or big data workloads, it is recommended to scale down various parameters like dataset size or number of epochs, with the help of environment variables to reduce the CI running time and iterate quickly. In this case, we declared environment variables like `NUM_EPOCHS`, `DATASET_REDUCTION`, and `BATCH_SIZE` to control the number of epochs, size of training data, and batch size respectively in our workflow. Using the above variables we used only 10% of the dataset and configured the training for a single epoch, thus effectively reducing our CI running time by approx. 75%. The `.travis.yml` configuration file used in our case is shown in Lst. 5. It can be generated by running `popper ci travis` from the command line. By setting up CI for Popper workflows, users can continuously validate changes made to their workflows and also protect their workflows from getting outdated due to reasons such as outdated dependencies, outdated container images, and broken links.

E. *Results*

The results obtained from executing the workflow in the different computing environments have been shown in Fig. 2. We can see that the training duration drastically reduced, by almost 75% as we went from running the workflow on the local machine to running on Kubernetes. We achieved further speedup by moving the workflow execution to Slurm where we used 2 GPUs from 2 different nodes instead of 1 GPU as in the case of Kubernetes. Also, Popper allowed us to scale our workflow down and run it on CI to continuously validate the workflow. These case studies show how Popper helps improve the performance of scientific workflows and boost developer productivity by allowing seamless reproduction on cloud and HPC infrastructure.

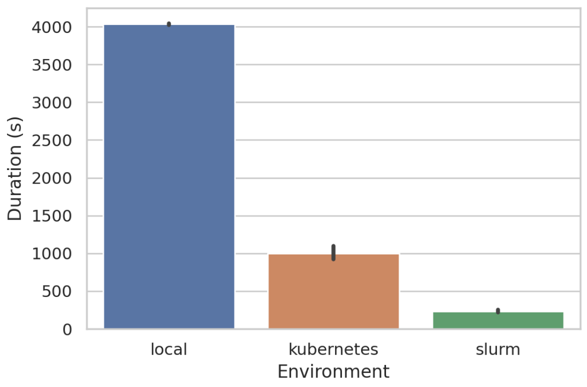

Figure 2: Comparison of training runtime in 3 different computing environments with Popper.

IV. Discussion

Having portable workflows drastically reduces software development and debugging time by enabling developers and researchers to quickly iterate and test the same workflow logic in different computing environments. As anecdotal evidence for this claim, we analyzed the GitHub repository[4] of MLPerf [61], a benchmark suite that measures how fast a system can train ML models. From a total of 123 issues, 67 were related to problems of reproducibility: missing or outdated versions of dependencies, documentation not aligning with the code, missing or broken links for datasets. Popper can solve much of the problems generally noticed while reproducing research artifacts like the ones we found.

The configuration files are orthogonal to the workflow and are only meant to provide environment-specific options to allow running the same workflow in different environments. In general, the adjustments that need to be made in order to reproduce workflows on Kubernetes and Slurm are the following:

- To run workflows on Kubernetes clusters, users need to pass some configuration options through a YAML file with contents similar to the one shown in Lst. 3. The `volume_size` and `namespace` options are not required if the defaults are suitable for running the workflow but we show it here to depict some ways in which the Kubernetes resource manager can be customized.
- Similarly, for running workflows on Slurm, users need to specify a few configuration options like the number of nodes to use for running the job concurrently, the number of CPUs to allocate to each task, the MPI library to bind to, as shown in Lst. 4.

With few tweaks, a workflow developed on a local machine can be executed on Kubernetes and Slurm clusters. More generally, this feature allows Popper to cleanly separate concerns: experimentation logic is encoded in the workflow file; runtime or infrastructure information is contained in a configuration file. Depending on the scenario, users might not even need to be aware of the contents of a configuration file, and instead rely on system administrators to provide this information. In this way, Popper allows researchers and developers to build and test workflows in different computing environments with relatively minimal effort.

[4]`https://github.com/mlperf/training`

Relying on Dockerfiles and OCI images as common denominators (Section II-B) allows to raise the level of abstraction across multiple container runtimes and orchestrators. Instead of having to deal with low level engine-specific issues, users can create and share workflows without worrying about what container infrastructure is available in the environments they have access to. We believe these invariants will hold in the future, with Dockerfiles likely being standardized as well.

V. Related Work

The problem of implementing multi-container workflows as described in Section I is addressed by several existing tools. We briefly survey some of these tools and technologies and compare them with Popper by grouping them in categories.

A. *Workflow definition languages*

Standard workflow definition languages like CWL, WDL, and YAWL [62] provide an engine agnostic interface for specifying workflows declaratively. Being engine agnostic, different workflow engines can adopt these languages as these workflow languages provide a plethora of useful syntactic elements to support a wide range of workflow engine features. Some of these workflow definition languages provide syntax for fine-grained control of resources by the users like defining the amount of CPU and memory to be allocated to each step, specifying scheduling policies, among others. Most of these languages support syntax for integration with various computing backends like container engines (e.g. Docker, uDocker [63], Singularity), HPC clusters (e.g. HTCondor [64] or LSF [65]) and Kubernetes (e.g. AWS, GCP, Azure or baremetal). For a user whose primary goal is to automate running a set of containerized scripts in sequence, learning these new workflow languages and syntaxes might add to the overall complexity. One of Popper's primary goals is to minimize the workflow language overhead as much as possible by allowing users to specify workflows using vanilla YAML syntax, thus keeping the learning curve flat and preventing sources of confusion.

B. *Workflow execution engines*

Workflow execution engines can be categorized in several different categories. In this section, we discuss a few frequently used categories of workflow execution engines namely Generic, Cloud Native, Container Native, and CI and compare their pros and cons with Popper.

1) Generic workflow execution engines: Few examples of this category are stable and mature scientific workflow engines like Nextflow, Pegasus, and Taverna which have recently introduced support for running steps in software containers. Also in this category are frameworks that support, in addition to defining workflows, executing and sharing them, for example, CK [66], REANA [67], OpenML [68], among others [69]. Popular workflow engines like Airflow and Luigi [70] require specifying workflows using programming languages and also provide pluggable interfaces that require the installation of separate plugins. For example, Airflow and Luigi use Python, Copper [71] uses Java, Dagr [72] uses Scala and SciPipe [73] uses Go as their workflow definition language. The goal with Popper is to minimize overhead both in terms of workflow language syntax and infrastructural requirements for running workflows and hence allow users to focus solely on writing the workflows. The first issue is already addressed in the previous subsection, but it's also relevant here because not all engines support standard workflow languages such as CWL, and also learning specific programming languages in order to implement a workflow might be overwhelming for some users. Most of these popular workflow engines like Pegasus, Airflow, and Luigi also require a standalone service that users need to learn how to deploy and interact with before executing workflows, thus adding to the complexity. Popper also mitigates this issue as it can be downloaded and run as a standalone executable and does not assume any service deployment or infrastructural management before running workflows.

2) Container-native workflow execution engines: The container-native paradigm encourages shifting the entire software development lifecycle which includes building, testing, debugging, and deployment to within software containers. The workflow engines that assume running steps of a workflow inside separate containers are usually termed as container-native workflow engines. Streamflow [74], Flyte [75], and Dray [76] are some well-known examples of container-native workflow execution engines. Some of these container-native workflow engines like Flyte and Dray are built around a client-server architecture requiring some service deployment effort before being able to run workflows on them. Popper falls in this category of container-native workflow engines but it does not assume any service deployment before running workflows, hence mitigating any extra service maintenance overhead or cost.

3) Cloud-native workflow execution engines: Cloud-native computing is an emerging paradigm in software development that aims to utilize existing cloud infrastructure to build, run, and scale applications in the cloud. The cloud-native paradigm is a subset of the container-native paradigm since it encourages running applications not only inside containers but also on cloud infrastructure. Container engines like Docker and orchestration tools like Kubernetes have become an integral part of the cloud-native paradigm over the years. With the wide-spread acceptance of the cloud-native paradigm, several cloud-native workflow engines like Argo [77], Pachyderm [78], Brigade [79] have come into existence. These workflow engines facilitate running workflows on the cloud by running an entire workflow in containers managed by Kubernetes. The limitation of these workflow engines is the requirement of having access to a Kubernetes cluster which can block users from running workflows in the absence of one. Although Popper can run workflows on the cloud using Kubernetes, it does not necessarily require access to a Kubernetes cluster for running the containerized steps of a workflow. Popper is not exclusively cloud-native since it does not assume the presence of a Kubernetes cluster for running workflows, but in addition to being able to work as cloud-native i.e. run workflows on Kubernetes, it can also behave as container-native in different computing environments like a local machine, Slurm and cloud VM instances over SSH.

C. Continuous integration tools

Continuous Integration is a DevOps practice that enables building and testing code frequently to prevent the accumulation of broken code and support faster development cycles. Tools like Travis, Circle, Jenkins, and GitLab-CI have become the standard for doing CI, but they were primarily built for running on the Cloud. This results in increased iteration time, especially for quick modify-compile-test loops, where the time spent in booting of VMs, scheduling of CI jobs becomes an overhead. Reproducing experiments on the local machine that originally run on CI services, requires imitating the CI environment locally which makes the entry barrier high. Running on CI tools hosted locally, like using Gitlab-runner [80], requires knowledge and expertise in running and using the specific tools. Popper tackles these problems by providing a workflow abstraction that allows users to write a workflow once and run them interchangeably between different environments like a local machine, CI services, Cloud, and HPC by learning a single tool only. Given the above, Popper is not intended to replace CI tools, but rather serve as an abstraction on top of CI services, helping to bridge the gap between a local and a CI environment.

D. Previous version of popper

Earlier work [81,82] introduced the *Popper Convention*, a set of guidelines for organizing folders and bash scripts inside a Git repository in order to make it easier to reproduce experiments associated to academic articles. In practical terms, the Popper convention prescribes a fixed folder layout and script naming scheme. This results in having a hard-coded workflow, with each step represented by a bash script.

An early version of the CLI tool introduced in Section II-E1 aided in the execution of experiments that followed the Popper convention, and was later labeled "Popper 1.x"[5]. This earlier Popper 1.x version was significantly simpler, as it did not

[5]Available in the branch `v1.x` of the official repository: `https://github.com/getpopper/popper/tree/v1.x`.

assume containers, did not defined a workflow language, and could not be extended in any way. In contrast, the Popper engine presented here fully embraces the container-native paradigm by implementing a principled design (described Section II-B), with a robust architectural design, support for workflow files, as well as all the other features described earlier.

In addition to the Popper convention papers, tutorials that introduce the Popper container-native engine have been held in various workshops [83], introductory talks and seminars. Additionally, a case study using Popper for implementing computer network experiments was presented in [84]. This earlier work did not touch on the principles, architecture or cloud/HPC runners that were introduced in previous sections.

VI. Future work

As future work, we plan the following:

- Add support for more container engines like NVIDIA Pyxis, Charliecloud, Shifter [85] and resource managers like HTCondor, TORQUE [86] to Popper in order to extend the range of the different computing environments currently supported.
- Add more exporter plugins for exporting Popper workflows to advanced workflow syntaxes such as CWL, WDL, and Airflow to enable interoperability between different workflow engines.
- Currently, Popper supports logging to `stdout` or a file. This can be extended to have an abstract mechanism to store and export logs to logging drivers like syslog [87], fluentd [88], or AWS CloudWatch [89]. Similar abstractions can be implemented to support different tracing mechanisms of containers like `dtrace` [90].
- Add plugins for engines like Kata container [91] and Firecracker [92] to allow running containers with the isolation guarantees of a VM.
- Building, caching, and layering of images is currently taken care of by the underlying container engine. We plan to add an `image` attribute to the Popper workflow syntax to abstract the process of building and pushing images to registries. This feature will effectively allow images to be built in an engine- and builder-agnostic way, and allow the use of tools such as Kaniko [93], BuildKit [94] or `img`. These tools support rootless image builds and significantly speed up builds by providing remote caching features.

Acknowledgements

This work was partially funded by the NSF Awards #OAC-1836650 (IRIS-HEP[6]) and #CNS-1705021, as well as by the Center for Research in Open Source Software (CROSS)[7].

[6]`https://iris-hep.org`
[7]`https://cross.ucsc.edu`

References

[1] “Zenodo.” Available at: https://zenodo.org/.

[2] “Figshare.” Available at: https://figshare.com/.

[3] “Github.” Available at: https://github.com/.

[4] J.H. Stagge, D.E. Rosenberg, A.M. Abdallah, H. Akbar, N.A. Attallah, and R. James, “Assessing data availability and research reproducibility in hydrology and water resources,” *Scientific data*, vol. 6, 2019, p. 190030.

[5] M. Baker, “Reproducibility crisis?” *Nature*, vol. 533, 2016, pp. 353–66.

[6] F. Fidler and J. Wilcox, “Reproducibility of scientific results,” *The stanford encyclopedia of philosophy*, E.N. Zalta, ed., https://plato.stanford.edu/archives/win2018/entries/scientific-reproducibility/; Metaphysics Research Lab, Stanford University, 2018.

[7] S.N. Goodman, D. Fanelli, and J.P. Ioannidis, “What does research reproducibility mean?” *Science translational medicine*, vol. 8, 2016, pp. 341ps12–341ps12.

[8] P. Ivie and D. Thain, “Reproducibility in scientific computing,” *ACM Comput. Surv.*, vol. 51, Jul. 2018. Available at: https://doi.org/10.1145/3186266.

[9] S.R. Piccolo and M.B. Frampton, “Tools and techniques for computational reproducibility,” *GigaScience*, vol. 5, 2016, pp. s13742–016.

[10] R.D. Peng, “Reproducible research in computational science,” *Science*, vol. 334, 2011, pp. 1226–1227.

[11] J.-L.R. Stevens, M. Elver, and J.A. Bednar, “An automated and reproducible workflow for running and analyzing neural simulations using lancet and ipython notebook,” *Frontiers in neuroinformatics*, vol. 7, 2013, p. 44.

[12] A. Bánáti, P. Kacsuk, and M. Kozlovszky, “Minimal sufficient information about the scientific workflows to create reproducible experiment,” *2015 ieee 19th international conference on intelligent engineering systems (ines)*, IEEE, 2015, pp. 189–194.

[13] R. Qasha, J. Cała, and P. Watson, “A framework for scientific workflow reproducibility in the cloud,” *2016 ieee 12th international conference on e-science (e-science)*, IEEE, 2016, pp. 81–90.

[14] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, “Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids,” *Proceedings of the 1st acm sigmod workshop on scalable workflow execution engines and technologies*, 2012, pp. 1–13.

[15] P. Di Tommaso, M. Chatzou, E.W. Floden, P.P. Barja, E. Palumbo, and C. Notredame, “Nextflow enables reproducible computational workflows,” *Nature biotechnology*, vol. 35, 2017, p. 316.

[16] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, “Pegasus: Mapping scientific workflows onto the grid,” *European Across Grids Conference*, Springer, 2004, pp. 11–20.

[17] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, and A. Wipat, “Taverna: A tool for the composition and enactment of bioinformatics workflows,” *Bioinformatics*, vol. 20, 2004, pp. 3045–3054.

[18] J. Zhao, J.M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble, “Why workflows break — understanding and combating decay in taverna workflows,” *2012 ieee 8th international conference on e-science*, 2012, pp. 1–9.

[19] H. Meng and D. Thain, “Facilitating the reproducibility of scientific workflows with execution environment specifications,” *Procedia Computer Science*, vol. 108, 2017, pp. 705–714.

[20] B. Howe, “Virtual appliances, cloud computing, and reproducible research,” *Computing in Science & Engineering*, vol. 14, 2012, pp. 36–41.

[21] M. Wannous, H. Nakano, and T. Nagai, “Virtualization and nested virtualization for constructing a reproducible online laboratory,” *Proceedings of the 2012 ieee global engineering education conference (educon)*, 2012, pp. 1–4.

[22] R.K. Barik, R.K. Lenka, K.R. Rao, and D. Ghose, “Performance analysis of virtual machines and containers in cloud computing,” *2016 international*

conference on computing, communication and automation (iccca), IEEE, 2016, pp. 1204–1210.

[23] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," *Proceedings of the 17th international middleware conference*, 2016, pp. 1–13.

[24] P.B. Menage, "Adding generic process containers to the linux kernel," *Proceedings of the Linux symposium*, Citeseer, 2007, pp. 45–57.

[25] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, 2014, pp. 81–84.

[26] G.M. Kurtzer, V. Sochat, and M.W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, 2017, p. e0177459.

[27] Rkt Community, "Rkt," Sep. 2019. Available at: https://github.com/rkt/rkt.

[28] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2017, p. 36.

[29] "Podman." Available at: https://github.com/containers/libpod.

[30] I. Jimenez, C. Maltzahn, A. Moody, K. Mohror, J. Lofstead, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "The role of container technology in reproducible computer systems research," *First workshop on containers (woc 2015) (workshop co-located with ieee international conference on cloud engineering - ic2e 2015)*, Tempe, AZ: 2015.

[31] J. Stubbs, S. Talley, W. Moreira, R. Dooley, and A. Stapleton, "Endofday: A container workflow engine for scalable, reproducible computation." *IWSG*, 2016.

[32] C. Zheng and D. Thain, "Integrating containers into workflows: A case study using makeflow, work queue, and docker," *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, ACM, 2015, pp. 31–38.

[33] E. Deelman, T. Peterka, I. Altintas, C.D. Carothers, K.K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, 2018, pp. 159–175.

[34] S. Cohen-Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. Le Bras, F. Lemoine, and others, "Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities," *Future Generation Computer Systems*, vol. 75, 2017, pp. 284–298.

[35] D.K. Rensin, *Kubernetes - scheduling the future at cloud scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472: 2015. Available at: http://www.oreilly.com/webops-perf/free/kubernetes.csp.

[36] M. Rodriguez and R. Buyya, "Container orchestration with cost-efficient autoscaling in cloud computing environments," *Handbook of research on multimedia cyber security*, IGI Global, 2020, pp. 190–213.

[37] "Slurm workload manager." Available at: https://slurm.schedmd.com/overview.html.

[38] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," *Haifux, May*, vol. 186, 2013.

[39] "Opencontainers.org." Available at: https://www.opencontainers.org/.

[40] R. Yasrab, "Mitigating docker security issues," *arXiv preprint arXiv:1804.05039*, 2018.

[41] D. Brayford, S. Vallecorsa, A. Atanasov, F. Baruffa, and W. Riviera, "Deploying ai frameworks on secure hpc systems with containers." *2019 ieee high performance extreme computing conference (hpec)*, IEEE, 2019, pp. 1–6.

[42] C. The MPI Forum, "MPI: A message passing interface," *Proceedings of the 1993 acm/ieee conference on supercomputing*, New York, NY, USA: Association for Computing Machinery, 1993, pp. 878–883. Available at: https://doi.org/10.1145/169627.169855.

[43] S. Ibrahim, K.-K.R. Choo, Z. Yan, and W. Pedrycz, *Algorithms and architectures for parallel processing: 17th international conference, ica3pp 2017, helsinki, finland, august 21-23, 2017, proceedings*, Springer, 2017.

[44] M. Virmani, "Understanding devops & bridging the gap from continuous integration to continuous delivery," *Fifth international conference on the innovative computing technology (intech 2015)*, IEEE, 2015, pp. 78–82.

[45] "Travis CI." Available at: https://travis-ci.org/.

[46] "Circle CI." Available at: https://circleci.com/.

[47] "Jenkins." Available at: https://www.jenkins.io.

[48] O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language (yaml™) version 1.1," *Working Draft 2008-05*, vol. 11, 2009.

[49] "Behold, the world's most popular programming language – and it is...wait, er, yaml?!?" Available at: https://www.theregister.co.uk/2018/11/19/popular_programming_language_yaml.

[50] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." *NSDI*, 2011, pp. 22–22.

[51] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," *Proceedings of the international conference for high performance computing, networking, storage and analysis*, New York, NY, USA: Association for Computing Machinery, 2017. Available at: https://doi.org/10.1145/3126908.3126925.

[52] "Pyxis." Available at: https://github.com/NVIDIA/pyxis.

[53] P. Amstutz, M.R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, and others, "Common workflow language, v1. 0," 2016.

[54] "OpenWDL." Available at: https://openwdl.org/.

[55] "Apache airflow." Available at: https://github.com/apache/airflow.

[56] A. Sergeev and M.D. Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018. Available at: http://arxiv.org/abs/1802.05799.

[57] A. Gulli and S. Pal, *Deep learning with Keras*, Packt Publishing Ltd, 2017.

[58] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 1998. Available at: http://yann.lecun.com/exdb/mnist/.

[59] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the sun network filesystem," *Proceedings of the summer usenix conference*, 1985, pp. 119–130.

[60] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of cloudlab," *2019 USENIX annual technical conference (USENIX ATC 19)*, Renton, WA: USENIX Association, 2019, pp. 1–14. Available at: https://www.usenix.org/conference/atc19/presentation/duplyakin.

[61] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, and others, "Mlperf training benchmark," *arXiv preprint arXiv:1910.01500*, 2019.

[62] W.M. Van Der Aalst and A.H. Ter Hofstede, "YAWL: Yet another workflow language," *Information systems*, vol. 30, 2005, pp. 245–275.

[63] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-Garcı'a, and P. Orviz, "Enabling rootless linux containers in multi-user environments: The udocker tool," *Computer Physics Communications*, vol. 232, 2018, pp. 84–97.

[64] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: A distributed job scheduler," *Beowulf cluster computing with windows*, 2001, pp. 307–350.

[65] X. Wei, W.W. Li, O. Tatebe, G. Xu, L. Hu, and J. Ju, "Implementing data aware scheduling in gfarm (r) using lsf (tm) scheduler plugin mechanism." *GCA*, vol. 5, 2005, pp. 3–5.

[66] G. Fursin, "The collective knowledge project: Closing the gap between ml&systems research and practice with portable workflows, reusable automation actions, and reproducible crowd-benchmarking," May. 2020.

[67] T. Šimko, L. Heinrich, H. Hirvonsalo, D. Kousidis, and D. Rodrı’guez, “REANA: A system for reusable research data analyses,” *EPJ web of conferences*, EDP Sciences, 2019, p. 06034.

[68] J. Vanschoren, J.N. Van Rijn, B. Bischl, and L. Torgo, “OpenML: Networked science in machine learning,” *ACM SIGKDD Explorations Newsletter*, vol. 15, 2014, pp. 49–60.

[69] R. Isdahl and O.E. Gundersen, “Out-of-the-box reproducibility: A survey of machine learning platforms,” *2019 15th international conference on eScience (eScience)*, IEEE, 2019, pp. 86–95.

[70] “Luigi.” Available at: https://github.com/spotify/luigi.

[71] “Copper.” Available at: https://github.com/copper-engine/copper-engine.

[72] “Dagr.” Available at: https://github.com/fulcrumgenomics/dagr.

[73] S. Lampa, M. Dahlö, J. Alvarsson, and O. Spjuth, “SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines,” *GigaScience*, vol. 8, 2019, p. giz044.

[74] “Streamflow.” Available at: https://github.com/alpha-unito/streamflow.

[75] K.U. Allyson Gale, “Introducing flyte: A cloud native machine learning and data processing platform,” *https://eng.lyft.com/*, 2020.

[76] “Dray.” Available at: https://github.com/CenturyLinkLabs/dray.

[77] “Argo.” Available at: https://github.com/argoproj/argo.

[78] J.A. Novella, P. Emami Khoonsari, S. Herman, D. Whitenack, M. Capuccini, J. Burman, K. Kultima, and O. Spjuth, “Container-based bioinformatics with Pachyderm,” *Bioinformatics*, vol. 35, 2018, pp. 839–846.

[79] “Brigade.” Available at: https://brigade.sh/.

[80] “Gitlab Runner.” Available at: https://gitlab.com/gitlab-org/gitlab-runner.

[81] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Standing on the shoulders of giants by managing scientific experiments like software,” *USENIX; login*, vol. 41, 2016, pp. 20–26.

[82] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “The popper convention: Making reproducible systems evaluation practical,” *2017 ieee international parallel and distributed processing symposium workshops (ipdpsw)*, IEEE, 2017, pp. 1561–1570.

[83] I. Jimenez, J. Lofstead, and C. Maltzahn, “Creating repeatable, reusable experimentation pipelines with popper: Tutorial,” *Proceedings of the 24th symposium on principles and practice of parallel programming*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 441–442. Available at: https://doi.org/10.1145/3293883.3302575.

[84] A. David, M. Souppe, I. Jimenez, K. Obraczka, S. Mansfield, K. Veenstra, and C. Maltzahn, “Reproducible computer network experiments: A case study using popper,” *Proceedings of the 2nd international workshop on practical reproducible evaluation of computer systems*, 2019, pp. 29–34.

[85] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, “Shifter: Containers for hpc,” *Journal of physics: Conference series*, 2017, p. 082021.

[86] G. Staples, “Torque resource manager,” *Proceedings of the 2006 acm/ieee conference on supercomputing*, 2006, pp. 8–es.

[87] “Syslog.” Available at: https://en.wikipedia.org/wiki/Syslog.

[88] “Fluentd, an open source data collector for unified logging layer.” Available at: https://www.fluentd.org/.

[89] “Amazon cloudwatch.” Available at: https://aws.amazon.com/cloudwatch/.

[90] “Dtrace.” Available at: http://dtrace.org/.

[91] “Kata containers, the speed of containers, the security of vms.” Available at: https://katacontainers.io/.

[92] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, 2020, pp. 419–434.

[93] “GoogleContainerTools/kaniko,” *GitHub*. Available at: https://github.com/GoogleContainerTools/kaniko.

[94] “Moby/buildkit,” *GitHub*. Available at: https://github.com/moby/buildkit.