Enabling seamless execution of Computational and Data science workflows on HPC and Cloud with the Popper Container-native Automation Engine

Jayjeet Chakraborty, Carlos Maltzahn, Ivo Jimenez
UC Santa Cruz

***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 three case studies where we take examples from machine learning and high-performance computing and turn them into Popper workflows.**

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 [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 use in HPC, etc. 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].

Currently, different container-based workflow engines are available but all 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 [37] is a light-weight workflow execution engine that allows users to follow the container-native paradigm for building 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 multiple resource managers and container engines giving users the ability to focus only on Dockerfiles, i.e. software dependencies and workflow logic, i.e. correct order of execution, and ignore the 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 [38] environments.
3. Three case studies 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 run on a local machine during development and how it can be reproduced in a Kubernetes cluster with GPUs to scale up and collect results. We also show how an HPC workflow developed on the local machine can be reproduced easily in a Slurm [39] cluster.

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, etc. 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 a uniform interface that allows users to write workflows once and reuse them on different environments with tweaks to the configuration file.

B. 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 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) [40] specifications. However, it was not designed for use in multi-user HPC environments and also has significant security issues [41], 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. Also, Docker uses cgroups [42] to isolate

containers, which conflicts with the Slurm scheduler since it also uses cgroups to allocate resources to jobs and enforce limits [43].

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 [44] 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 [45]. 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 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 [46]. Several hosted CI services like Travis [47], Circle [48], and Jenkins [49] make continuous integration and continuous validation easily accessible.

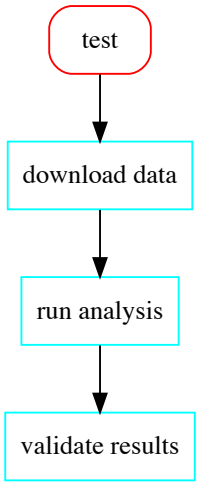

Figure 1: DOT diagram of a Popper workflow DAG

C. Workflow Definition Language

YAML [50] 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 [51], 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.

Listing 1 A three-step 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]
```

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

D. Workflow Execution Engine

The Popper workflow execution engine is composed of several components that talk to each other during 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. 2;

1) Command Line Interface (CLI): Besides allowing users to communicate with the workflow runner, the CLI allows visualizing workflows by generating DOT diagrams [52] like the one shown in Fig. 1; 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.

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, SGE, HTCondor) and cluster managers (e.g. Kubernetes, Mesos, YARN). 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

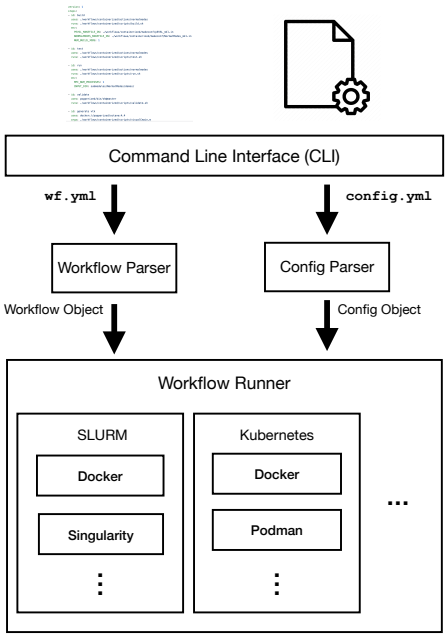

Figure 2: Architecture of the Popper workflow engine

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 [53] and Pyxis [54]. 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.

E. Workflow Exporter

Popper allows exporting a workflow to other workflow specification formats such as CWL and WDL, 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, plugins for CI services like TravisCI, CircleCI, Jenkins and Gitlab-CI are implemented.

III. Case Study

In this section, we present three case studies demonstrating how the Popper workflow engine allows reproducing and scaling workflows in different computing environments. These case studies aim to emphasize on how Popper can help in mitigating these reproducibility issues and make life easier for researchers and developers. For these case studies, we built an image classification workflow that runs the training using Keras [56] over the MNIST [57] dataset having 3 steps; download; verify; and train. The workflow used for the case studies is depicted in Lst. 2. The code that the workflow references can be found here.[1]

Listing 2 Workflow used in the case studies.

```
steps:
- id: download-dataset
  uses: docker://gw000/keras
  args: ["python", "./scripts/download_dataset.py"]

- id: verify-dataset
  uses: docker://alpine:3.9.5
  args: ["./scripts/verify_dataset.sh"]

- id: run-training
  uses: docker://gw000/keras
  args: ["./scripts/run_training.sh"]
  env:
    NUM_EPOCHS: '10'
    BATCH_SIZE: '128'
```

The `download` step downloads the MNIST dataset in the workspace. The `verify` step verifies the downloaded archives against precomputed checksums. The `train` step then starts training the model on this downloaded dataset and records the duration of the training. The download and train steps use a Keras docker 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 otherwise it could make dependency management complex, hence defeating the purpose of containers.

The general paradigm 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. a. Finding images containing the required software from remote image registries like DockerHub, Quay.io, Google Container Registry, etc. b. If a prebuilt image is not available, a `Dockerfile` can be used to build an image manually which is a file containing specifications for building Docker images. 3. Running the workflow and refining it.

1) 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. We used an Apple Macbook Pro Laptop with a 2.4GHz quad-core Intel Core i5 64-bit processor and 8 Gb LPDDR3 RAM for this case study. The image classification workflow was built and run on the MNIST dataset [58] using the Docker 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. The results obtained over 5 executions have been shown in Fig. 3.

To achieve lower training durations, the training should ideally be done on GPUs in the cloud which in turn require these workflows to be easily portable to multi-node cloud environments. In the next section, we will look at how we ran the workflow developed on the local machine efficiently on the Kubernetes using Popper.

2) Workflow execution in the Cloud using Kubernetes: In this section, we discuss how we reduced the training duration in the above workflow by reproducing it on a GPU enabled Kubernetes cluster. On Kubernetes clusters, 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 like DockerHub, Google Container Registry, etc. Then a `PersistentVolumeClaim` is created to claim persistent storage space from a shared filesystem like NFS [59] for the different step pods to share. After the pod is created, the workflow context consisting of the scripts, configs, etc. is copied into the shared volume mounted inside the pod and executed. Although any Kubernetes cluster can be used, for this case study, 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 the workflow developed on the local machine in the 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, the image registry credentials, etc. The training was configured with an overfitting patience of 5 and was allowed to run till it overfits similar to what was done for the local machine case study.

As we can see from Fig. 3, the average training duration was

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

almost 1/4th of what it took to train on the local machine. This shows how Popper helps improve the performance of scientific workflows drastically by allowing easy reproduction in cloud infrastructure.

3) Workflow execution in Slurm clusters: For this case study, we modified our training script to use the Horovod [61] distributed deep learning framework to facilitate training with MPI [62] on a Slurm cluster. For running workflows on Slurm clusters, container engines that are HPC aware, like Singularity, Charliecloud, and Pyxis need to be used. The MPI based workflows can be made compatible between Slurm clusters with different MPI implementations if the Singularity images used in the workflows are built on the cluster by binding to the local MPI libraries. Also, the programs and scripts need to be MPI compatible to utilize the total compute capacity of multiple nodes in HPC clusters. We recommend using a shared filesystem like NFS or AFS [63] mounted on each node and placing the workflow context in there to keep the workspace consistent across all the nodes. We used 3 VMs from Azure each with the same NVIDIA 12GB PCI P100 GPU running Ubuntu 18.04 for this experiment and used Singularity as the container engine for running this workflow. 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 run using MPI on 2 compute nodes having a GPU each and the training parameters were the same as in the previous case studies. As we can see from Fig. 3, Popper allowed us to run the workflow in a Slurm cluster with MPI and hence utilize the processing power of multiple GPUs and drastically reduce the training duration.

4) Workflow execution on CI: We used the workflow exporter to generate a Travis config file, pushed our MNIST project to GitHub with the config, and activated the repository on Travis to run our workflows on CI. For long-running workflows like those consisting of ML/AI or BigData workloads, it is recommended to scale down various parameters like dataset size, epochs, etc. with the help of environment variables to reduce the CI running time and iterate quickly. 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` file used by our case study is shown in Lst. 3. 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.

Listing 3 Popper generated Travis config.

```
---
dist: xenial
language: python
python: 3.7
services: docker
install:
- git clone https://github.com/systemslab/popper /tmp/popper
- export PYTHONUNBUFFERED=1
- pip install /tmp/popper/src
script: popper run -f wf.yml
```

IV. Discussion

A summary of the training duration and accuracy obtained by running the workflow in three different computing environment is shown in Fig. 3. As one would expect, running the same workflow on better, larger hardware resources reduces the amount of time needed to train the models.

This case study showcases the benefits of using Popper: 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. To expand on this point, we analyzed the GitHub repository[2] of MLPerf [64], 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; etc. Popper can solve much of the problems generally noticed in reproducing research artifacts like these we found.

As exemplified in the use cases, Popper helps build workflows that can be run on Cloud and HPC environments besides the local machine with minimal changes in configuration in a sustainable fashion. The adjustments that users need to make to reproduce workflows on Kubernetes and Slurm is described below.

1. 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. 4. 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.

2. Similarly, for running 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 worker nodes to use, etc. as shown in Lst. 5.

[2] https://github.com/mlperf/training

Listing 4 Config file for running on Kubernetes.

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

Listing 5 Config file for running on Slurm.

```
engine:
  name: singularity

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

It can be seen that with few tweaks like changing the resource manager options in the configuration file, a workflow developed on a local machine can be executed in Kubernetes and Slurm. In this way, Popper allows researchers and developers to build and test workflows in different computing environments with relatively minimal effort.

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 [65], WDL [66], and YAWL [67] 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, etc. Most of these languages support syntax for integration with various computing backends like container engines (e.g. Docker, uDocker [68], Singularity), HPC clusters (e.g. HTCondor [69], LSF [70], Slurm), cloud providers (e.g. AWS, GCP, Azure), Kubernetes, etc. 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.

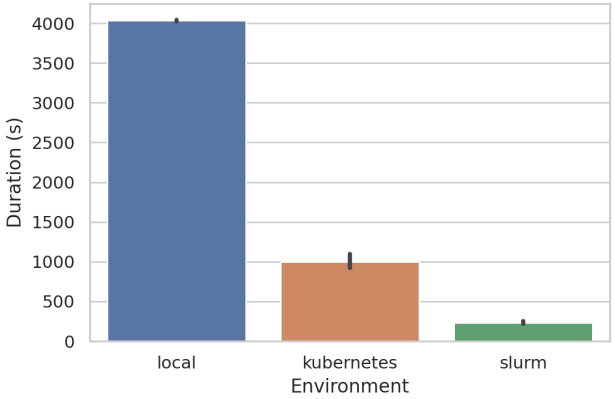

Figure 3: Comparison of training durations in 3 different computing environments with Popper.

B. Workflow execution engines

Workflow execution engines can be categorized in several different categories. In this section, we have discussed three frequently used categories of workflow execution engines namely Generic, Cloud Native, and Container Native and compared 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 running steps in software containers; Also, tools like the Collective Knowledge Framework [71] helps researchers in sharing their research in the form of reusable workflows that consists of artifacts like algorithms, datasets, models, scripts, and experimental results and provide APIs, CLI, meta descriptions, and common automation actions for the related artifacts. Popular workflow engines like Airflow and Luigi [72] 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 [73] uses Java, Dagr [74] uses Scala and SciPipe [75] 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 for workflow execution seems like an exaggeration. 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: 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 [76], Flyte [77], and Dray [78] 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 [79], Pachyderm [80], Brigade [81] have come into existence. These workflow engines facilitate running workflows in 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 in 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 [82], 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. Related work on Popper

Popper aids in making experiments reproducible not only from ML/AI but also from other areas of computer science like Networking, Storage systems, Computational genomics, etc. Reproducibility issues in Systems experiments like experiments with the GassyFS file system were explored and tackled by Popper [83]. This experiment used the `1.x` version of Popper, which assumed a workflow from a hardcoded folder layout consisting of bash scripts representing each step of the workflow. Few network simulation experiments that run on the Cooja network simulation platform were made reproducible with the use of Popper [84]. Additionally, tutorials on how Popper helps in producing reproducible research from different domains of computational science were held in various workshops and talks [85]. This paper introduces the `2.x` version of Popper that assumes a workflow defined in a YAML format and adds support for more container engines like Podman and Singularity along with resource managers like Slurm and Kubernetes. This paper also evaluates the usefulness of having support for running workflows in Cloud and HPC by providing a case study.

VI. Conclusion and Future Work

In this paper, we present Popper, a container-native workflow execution engine that aims to solve the reproducibility problem in computational science. We first describe and analyze the design of Popper's YAML based workflow syntax and the architecture of the Popper workflow engine. We present a few case studies using an ML workflow to demonstrate how Popper helps developers and researchers build and test workflows in different computing environments like a local machine, Kubernetes, and Slurm quickly and with minimal changes in configuration. Next, we compare Popper with existing state-of-the-art workflow engines illustrating its YAML based workflow syntax that has a relatively low entry barrier and its ability to run containerized workflows without requiring access to any cloud environment.

As future work, we have planned the following improvements for Popper:

- Add support for more container engines like NVIDIA Pyxis, Charliecloud, Shifter and resource managers like HTCondor, TORQUE 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 the STDOUT or a file. This can be extended to have an abstract mechanism to store and export logs to logging drivers like syslog, fluentd, AWS CloudWatch, etc. Similar abstractions can be implemented to support different tracing mechanisms of containers like `dtrace`.
- Add plugins for engines like Kata container and Firecracker will 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 our workflow syntax to make the building and pushing of images container-engine agnostic using tools like Kaniko and BuildKit that supports rootless image builds and makes builds efficient using their remote caching feature.

Acknowledgements: This work was partially funded by the NSF Awards #OAC-1836650 (IRIS-HEP[3]) and #CNS-1705021, as well as by the Center for Research in Open Source Software (CROSS)[4].

VII. References

[1] "Zenodo - research. Shared." Available at: https://zenodo.org/.

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

[3] "Github, the world's leading software development platform." 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.

[3] https://iris-hep.org

[4] https://cross.ucsc.edu

[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, a security-minded, standards-based container engine," 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 Community, "Containers/libpod," Sep. 2019. 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] "Popper, container-native workflow execution engine." Available at: https://github.com/systemslab/popper.

[38] L.T. Yang and M. Guo, *High-performance computing: Paradigm and infrastructure*, John Wiley & Sons, 2005.

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

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

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] "Circle ci." Available at: https://circleci.com/.

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

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

[51] "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.

[52] "The dot language." Available at: https://www.graphviz.org/doc/info/lang.html.

[53] 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.

[54] "Pyxis: Container plugin for slurm workload manager." Available at: https://github.com/NVIDIA/pyxis.

[55] K.U. Allyson Gale, "Apache airflow," *GitHub*, 2020. Available at: https://github.com/apache/airflow.

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

[57] "The mnist database of handwritten digits." Available at: http://yann.lecun.com/exdb/mnist/.

[58] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, 2012, pp. 141–142.

[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] 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.

[62] W. Gropp, W.D. Gropp, E. Lusk, A. Skjellum, and A.D.F.E.E. Lusk, *Using mpi: Portable parallel programming with the message-passing interface*, MIT press, 1999.

[63] J.H. Howard and others, *An overview of the andrew file system*, Carnegie Mellon University, Information Technology Center, 1988.

[64] 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.

[65] 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.

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

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

[68] 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.

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

[70] 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.

[71] 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.

[72] "Spotify/luigi," *GitHub*. Available at: https://github.com/spotify/luigi.

[73] "Copper-engine/copper-engine," *GitHub*. Available at: https://github.com/copper-engine/copper-engine.

[74] "Fulcrumgenomics/dagr," *GitHub*. Available at: https://github.com/fulcrumgenomics/dagr.

[75] 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.

[76] "Alpha-unito/streamflow," *GitHub*. Available at: https://github.com/alpha-unito/streamflow.

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

[78] "Alpha-unito/streamflow," *GitHub*. Available at: https://github.com/CenturyLinkLabs/dray.

[79] Argo Community, "Argo: Get stuff done with kubernetes," Sep. 2019. Available at: https://github.com/argoproj/argo.

[80] 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.

[81] "Brigade, event-driven scripting for kubernetes." Available at: https://brigade.sh/.

[82] "Gitlab-org/gitlab-runner," *GitHub*. Available at: https://gitlab.com/gitlab-org/gitlab-runner.

[83] I. Jimenez, M. Sevilla, N. Watkins, and C. Maltzahn, *Popper: Making reproducible systems performance evaluation practical*, Santa Cruz, CA: UC Santa Cruz, 2016.

[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," *P-recs'19*, 2019.

[85] 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.