Popper 2.0: A Container-Native Workflow Execution Engine For Testing Complex Applications and Reproducing Scientific Explorations

Jayjeet Chakraborty, Carlos Maltzahn, Ivo Jimenez
UC Santa Cruz

***Abstract*—Researchers and developers working on computational science research often suffer from the problem of reproducibility in which they fail to reproduce experiments from their artifacts like code, data, configuration, etc. left behind by previous researchers. Factors like broken dependencies, version discrepancies, outdated links, OS differences generally give rise to this problem. Even if they are able to reproduce the experiments, the process generally involves interactively running a long list of commands in proper order. Therefore, to capture the high-level workflow that a researcher generally follows for reproducing experiments in a sustainable way, there is a need for a workflow engine that allows running containerized scripts following a prescribed order. In this paper, we introduce Popper, a container-native workflow engine that executes each step of a workflow inside containers on different computing environments. With Popper, researchers can easily build and run workflows on frequently used computing environments like local machines, HPC clusters, CI services, or Kubernetes.**

I. Keywords: Reproducibility, SciOps, Docker

II. Overview

A. Introduction

Nearly 48.6% of scientists and researchers working in various domains of computational science, upload experimental artifacts like code, figures, datasets, configuration files, etc. on open-access repositories like Zenodo [1], Figshare [2] or GitHub [3]. Unfortunately, only 1.1% of the artifacts available online are fully reproducible and 0.6% of them are partially reproducible [4]. This problem occurs mostly due to the lack of proper documentation, missing artifacts, broken software dependencies, etc. Scientific workflow engines like Nextflow, Pegasus, and Taverna which are well-known have been a predominant solution [5] [6] for handling the reproducibility problem by organizing the steps of a complex scientific workflow as the nodes of a directed acyclic graph (DAG) and executing them in the correct order [7].

One of the primary reasons behind this problem is the disparity in the environment where the workflows are developed and where they are reproduced [8]. Virtual Machines (VMs) were used to address this problem for some time due to their high isolation and environmental consistency guarantees, where every step of a workflow ran inside a separate VM [9] [10]. Since VM's had large resource footprints, researchers replaced VM's with software containers, i.e. light-weight virtualization technologies to provide platform-independent reproducibility [11] [12]. Although software (Linux) containers are a relatively old technology [13], it was not until recently, with the rise of Docker, that they entered mainstream territory [14]. Since then, this technology has transformed the way applications get deployed in shared infrastructures, with 25% of companies using this form of software deployment [15], and a market size projected to be close to 5B by 2023 [16]. Docker has been the de facto container runtime, with other container runtimes such as Singularity [17], Rkt [18], Charliecloud [19], and Podman [20] having emerged.

With Docker, the container-native software development paradigm emerged, which promotes the building, testing, and deployment of software in containers, so that users do not need to install and maintain packages on their machines, rather they can build or fetch container images which have all the dependencies present. Since these container runtimes are available for almost every well known operating system and architecture, experiments can be reproduced easily using containerized workflows in almost any environment [21] [22]. Although there are different container engines available, switching between them is often difficult as they have different API's, image formats, CLI interfaces, among many others. Also, there is an absence of tools that allow running containerized workflows 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 [23] [24]. Currently, different container-based workflow engines are available but all of them assume the presence of a fully provisioned Kubernetes [25] cluster. The difference between cloud-native and container-native is that, in the former, a Kubernetes cluster is required, while in the latter, only a container engine is required. Argo [26], Pachyderm [27], and Brigade [28] are popular examples of cloud-native workflow execution engines. It would be more convenient for researchers if workflow engines provide the flexibility of running workflows in a wide range of computing environments including those of their choice.

Popper [29] 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:

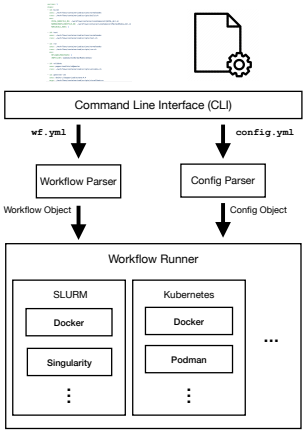

Figure 1: Architecture of the Popper workflow engine

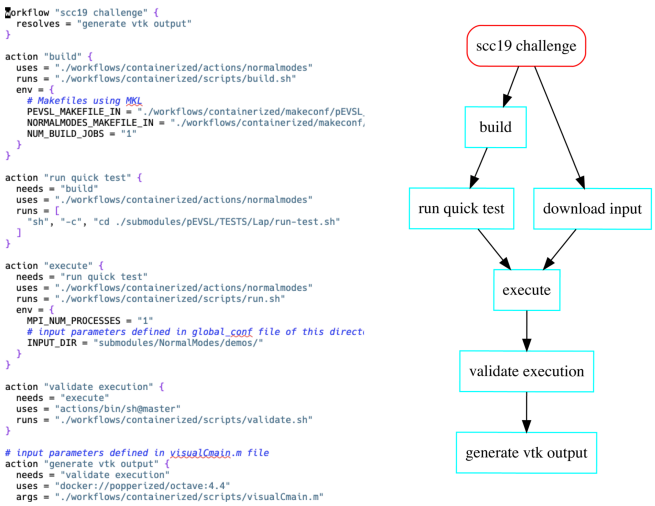

Figure 2: Casestudy

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. This arrangement also provides built-in support for continuous validation and portability of workflows which empowers researchers to develop workflows once and run interchangeably between CI services like Travis, Jenkins, etc. and the local machine without any modifications.
2. Popper CLI, an implementation of the above design that allows running workflows inside containers in different computing environments like local machines, Kubernetes clusters, or HPC [30] environments.

B. Implementation and Architecture

1) Workflow Definition Language: YAML [31] is a human-readable data-serialization language. It is primarily used in writing configuration files and in applications where data is stored or transmitted. Due to its simplicity and wide adoption [32], we chose YAML for defining popper workflows and for specifying the configuration for the execution engine. Another reason for adopting YAML as our workflow definition language was the availability of stable YAML parsers like PyYAML. An example popper workflow definition is shown below.

```
steps:
- id: download-data
  uses: docker://byrnedo/alpine-curl:0.1.8
  args: [
    "--create-dirs",
    "-Lo data/global.csv",
    "https://github.com/datasets/co2-fossil-global/raw/master/global.csv"
  ]

- id: run-analysis
  uses: docker://python:alpine
  args: [
    "scripts/get_mean_by_group.py",
    "data/global.csv", "5"
  ]

- id: validate-results
  uses: docker://python:alpine
  args: [
    "scripts/validate_output.py",
    "data/global_per_capita_mean.csv"
  ]
```

A popper workflow consists of a series of syntactical elements 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 inside a container can be defined by the `args` and `runs` attributes. Popper also supports running steps interactively, like for example running a Jupyter Notebook. Secrets and environment variables needed by a step can be specified through 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. A global `options` attribute is also available which can be used to define workflow engine options that are common to all the steps. An exporter module for Popper is being planned currently which would enable users to export Popper workflows in vanilla YAML to formats like Python code or CWL that are recognized by other well-known workflow engines like Apache Airflow, etc.

2) 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. 1;

a) Command Line Interface (CLI):

Besides allowing users to communicate with the workflow

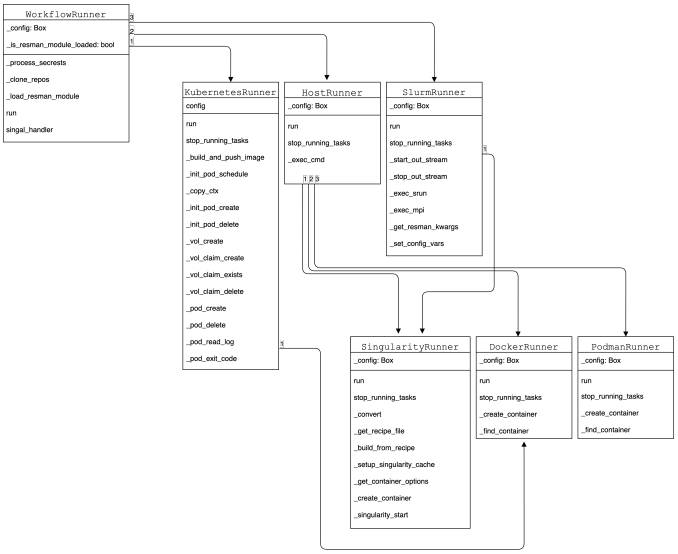

Figure 3: Class diagram for the Workflow runner

runner, the CLI allows visualizing workflows by generating DOT diagrams [33] like the one shown in Fig. 2; generates configuration files for continuous integration systems, e.g. TravisCI, Jenkins, Gitlab-CI, etc. so that users can continuously validate their workflows; provides dynamic workflow variable substitution capabilities, among others. The CLI interface is built using Click which is a Python package for creating command-line interfaces in a composable way with as little code as necessary. The CLI functionalities are defined in the `PopperCLI` abstraction within the `popper.cli` module. Popper also provides an interface for providing a configuration file that contains container and resource manager specific options.

b) 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 YML 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. The parser interface consists of a `WorkflowParser` class in the `popper.parser` module. The parser module uses the pykwalify package to validate a `dict` based on a predefined schema. The `WorkflowParser` has methods like `__apply_substitutions`, `__skip_steps`, `__filter_steps` that provide functionalities to manipulate workflows in different ways. The parser takes an input of a file and other metadata containing information on how to reshape the workflow data (if required) and gives out a Python Box, which is an immutable data container for dot notation access.

c) Workflow Runner:

The Workflow runner is in charge of taking a parsed workflow representation as input and executing it. It is defined as the `WorkflowRunner` class in the `popper.runner` module. It also downloads actions referenced by the steps in a workflow, checks the presence of secret variables that are required by a workflow, dynamically loads the resource manager module and starts the execution of a step using the configured container engine. The runner also maintains a cache directory following the XDG specifications to optimize multiple aspects of execution such as avoid cloning repositories unnecessarily, avoid re-building already built images. Thus, the workflow runner module drives the entire workflow execution. The UML class diagram for the workflow runner module is shown in Fig. 3;

d) Resource Manager and Container Engine Interface:

Popper supports running containers in both single-node and multi-node cluster environments. 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, etc.) and cluster managers (e.g. Kubernetes, Mesos, YARN, etc.). Currently, plugins for SLURM and Kubernetes exist, as well as the default Host runner that executes workflows on the host 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 Singularity containers but other container engines can also be integrated like Charliecloud [34] and Pyxis [35]. 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, renaming, etc. Currently, there are plugins for Docker, Singularity, and Podman, with others planned by the Popper community.

The behavior of a resource manager and a container engine can be customized by passing specific options through the configuration file enabling 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 a Slurm cluster using Singularity containers by specifying the job options in the configuration file. The configuration file can be created by users or provided by system administrators and can be passed through the CLI interface.

The resource manager interface of Popper has 3 parent classes i) `HostRunner`, contains common methods for building and running containers on the local machine. ii) `SlurmRunner`, contains methods for pulling and running containers in Slurm clusters, and iii) `KubernetesRunner`, contains methods for creating and managing `PeristentVolume`, `PersistentVolumeClaim`, and `Pod` in Kubernetes clus-

ters. Each resource manager class is subclassed by the container engine specific subclasses and add the container-specific functionalities. For example, the `DockerRunner` inherits from the `HostRunner` class. Another implementation of the `DockerRunner` inherits from the `KubernetesRunner` for running containers in Kubernetes.

e) Continuous Integration:

Popper allows users to continuously validate their workflows by allowing them to export workflows as CI pipelines for different continuous integration services like TravisCI, Circle, Jenkins, Gitlab-CI, etc. The CLI provides a `popper ci` sub-command that can be used to generate configuration files for different CI services. To set up CI for a project using Popper, it is required to generate a CI configuration file, push the project to Github and enable the repository on the CI provider. Using CI with Popper workflows enhances the reproducibility guarantees as continuous validation helps to keep a check on various breaking changes like outdated dependencies, broken links, deleted docker images, etc. Another benefit of using CI with Popper is that even without changes, jobs can be configured so that they run periodically (e.g. once a week), to ensure that they are in a healthy state. An example of a Travis configuration file generated by Popper is shown below.

```
---
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/cli
script: popper run -f .popper.yml
```

C. Quality Control

The tool has been unit tested using the `unittest` library from the Python standard library. Several plugins like the Slurm Runner has been unit tested by mocking the methods using the `unittest.mock` module which couldn't otherwise be tested in a CI environment. The current unit test coverage of the codebase is around 91%. Unit tests are run for different resource manager and container engine configurations in different Python versions like 3.6, 3.7, 3.8. The code repository is continuously tested by TravisCI on every push to GitHub. Also, the coding style of Popper's source code is kept consistent by the Black code formatter. Popper follows a release cycle of once a month and also has an automated release pipeline in place.

To quickly try out the Popper tool and check if it works,

- The Popper CLI tool can be installed by running,

```
$ pip install popper
```

or by using the Docker based installer given here.

- An example workflow can be scaffolded and run by doing,

```
$ popper scaffold
$ popper run
```

- By default, the workflow will use docker as the container runtime and will run the containers on the host. To customize the engine and resource manager,

```
$ popper run -r slurm -e singularity
```

Detailed documentation of Popper can be found here. The Popper repository also has an in-depth guide on performing Computational research tasks through Popper like using building papers in Latex, interactively using Jupyter Notebooks, Writing steps for workflows, etc. The @getpopper organization in GitHub provides a library of several workflows and Dockerfiles ranging from data science research using R and Python, spinning up Kubernetes clusters, Genomics pipelines, MLPerf benchmark workflows, etc.

III. Availability

A. Operating System

Popper only runs on Linux or macOS. On Windows systems, Popper can be executed in the Windows Subsystem for Linux (WSL2). Basically, the minimum requirement of using Popper is the availability of Docker on the system. So, any OS that supports running Docker should inherently support Popper too.

B. Programming Language

The tool is written in Python 3.6+.

C. Dependencies

- Docker
- Singularity (Required if workflows need to run in singularity container engine.)
- Podman (Required if workflows need to run in Podman container engine.)
- Kubernetes (Required for running workflows in a Kubernetes cluster.)
- Slurm (Required for running workflows in HPC environments.)

D. List of contributors

Popper has a long list of amazing contributors. The list of people who helped build Popper can be found here.

E. Language

English

F. Software Location

1) Code repository:

- Name: GitHub
- Identifier: https://github.com/getpopper/popper/
- Licence: MIT
- Date published: Released every month continuously
- Version: 2.7.0
- PyPi Package: https://pypi.org/project/popper/

IV. Reuse Potential

The Popper tool can for running a plethora of computational science workflows. Users can also run workflows that require heavy resources like a larger number of CPU cores or GPU in the cloud using the Kubernetes runner. Workflows involving MPI workloads can make use of the Slurm runner to run workflows in Slurm clusters using Singularity. Due to its pluggable architecture, users can add support for container runtimes and resource managers that fit best for their use cases. The main reason behind Popper being written in Python was to have contributions from both academia and industry and fits a lot of use cases from both worlds. Popper can be used to run workflows from but not limited to computational biology, data science and big data, machine learning and artificial intelligence, statistical inferences, CI/CD workflows, etc. Any high-level interactive workflow like writing a sequence of commands or running a sequence of bash scripts in a terminal can be captured and automated with Popper. Popper workflows can be shared through GitHub and reused by other researchers with the only criteria of having the Popper tool installed along with the required dependencies.

A. Funding Statement

B. Competing Interests

The authors have no competing interests to declare.

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] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] Datadog, "8 surprising facts about real Docker adoption," *8 surprising facts about real Docker adoption*, Jun. 2018. Available at: https://www.datadoghq.com/docker-adoption/.

[16] MarketsAndMarkets, "Application Container Market worth 4.98 Billion USD by 2023," 2018. Available at: https://www.marketsandmarkets.com/PressReleases/application-container.asp.

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

[18] Rkt Community, "Rkt, a security-minded, standards-based container engine," Sep. 2019. Available at: https://github.com/rkt/rkt.

[19] 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.

[20] Podman Community, "Containers/libpod," Sep. 2019. Available at: https://github.com/containers/libpod.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] "Popper, container-native workflow execution engine." Available at: https://github.com/systemslab/popper.

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

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

[32] "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.

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

[34] 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.

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