Popper 2.0: A Multi-container Workflow Execution Engine For Testing Complex Applications and Validating Scientific Explorations

Ivo Jimenez, Jayjeet Chakraborty, Arshul Mansoori, Quincy Wofford and Carlos Maltzahn

***Abstract*—Software containers allow users to "bring their own environment" to shared computing platforms, reducing the friction between system administrators and their users. In recent years, multiple container runtimes have arised, each addressing distinct needs (e.g. Singularity, Podman, rkt, among others), and an ongoing effort from the Linux Foundation (Open Container Innitiative) is standardizing the specification of Linux container runtimes. While containers solve a big part of the "dependency hell" problem, there are scenarios where multi-container workflows are not fully addressed by existing runtimes or workflow engines. Current alternatives require a full scheduler (e.g. Kubernetes), a scientific workflow engine (e.g. Pegasus), or are constrained in the type of logic that users can express (e.g. Docker-compose). Ideally, users should be able to express workflows with the same user-friendliness and portability of `Dockerfiles` (write once, run anywhere). In this article, we introduce "Popper 2.0" a multi-container workflow execution engine that allows users to express complex workflows similarly to how they do it in other scientific workflow languages, but with the advantage of running in container runtimes, bringing portability and ease of use to HPC scenarios. Popper 2.0 cleanly separates the three main concerns that are common in HPC scenarios: experimentation logic, environment preparation, and system configuration. To exemplify the suitability of the tool, we present a case study where we take the experimentation pipeline defined for the SC19 Reproducibility Challenge and turn it into a Popper workflow.**

I. Introduction

Although Software (Linux) containers are a relatively old technology [1], it was not until recently, with the rise of Docker, that they entered mainstream territory [2]. Since then, this technology has transformed the way applications get deployed in shared infrastructures, with 25% of companies using this form of software deployment [3], and a market size projected to be close to 5B by 2023 [4]. Docker has been the *de facto* container runtime, with other container runtimes such as Singularity [5], Charliecloud [6] and Podman[1] having emerged. The Linux Foundation bootstrapped the Open Container Initiative (OCI) [7] and is close to releasing version 1.0 of a container image and runtime specifications.

While containers solve a big part of the "dependency hell" problem [8], there are scenarios where a single container image is not suitable for implementing workflows associated to complex application testing or validating scientific explorations [9]. For example, a workflow might involve executing two stages, both of them requiring conflicting versions of a language runtime (e.g. Python 2.7 **and** Python 3.6). In this scenario, users could resort to solving such conflicts with the use of package managers, but this defeats the purpose of containers, which is to *not* have to do this sort of thing inside a container. More generally, the more complex a container image definition gets (a more complex `Dockerfile`), the more "monolithic" it gets, and thus the less maintainable and reusable it is. On the other hand, if an experimentation pipeline can be broken down into finer granularity units, we end up having pieces of logic that are easier to maintain and reuse.

```
workflow "scc19 challenge" {
  resolves = "generate vtk output"
}

action "build" {
  uses = "./workflows/containerized/actions/normalmodes"
  runs = "./workflows/containerized/scripts/build.sh"
  env = {
    # Makefiles using MKL
    PEVSL_MAKEFILE_IN = "./workflows/containerized/makeconf/pEVSL_
    NORMALMODES_MAKEFILE_IN = "./workflows/containerized/makeconf/
    NUM_BUILD_JOBS = "1"
  }
}

action "run quick test" {
  needs = "build"
  uses = "./workflows/containerized/actions/normalmodes"
  runs = [
    "sh", "-c", "cd ./submodules/pEVSL/TESTS/Lap/run-test.sh"
  ]
}

action "execute" {
  needs = "run quick test"
  uses = "./workflows/containerized/actions/normalmodes"
  runs = "./workflows/containerized/scripts/run.sh"
  env = {
    MPI_NUM_PROCESSES = "1"
    # input parameters defined in global_conf file of this direct
    INPUT_DIR = "submodules/NormalModes/demos/"
  }
}

action "validate execution" {
  needs = "execute"
  uses = "actions/bin/sh@master"
  runs = "./workflows/containerized/scripts/validate.sh"
}

# input parameters defined in visualCmain.m file
action "generate vtk output" {
  needs = "validate execution"
  uses = "docker://popperized/octave:4.4"
  args = "./workflows/containerized/scripts/visualCmain.m"
```

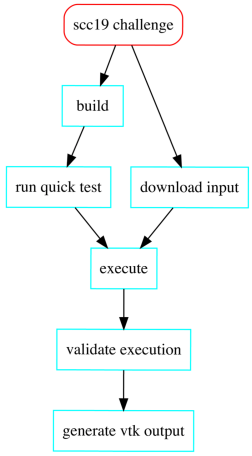

Figure 1: An end-to-end example of a workflow. On the left we have the `.workflow` file that defines the workflow. On the right, a pictorial representation of it.

Thus, we would like to break workflows into subunits, ideally having one container image per node in the directed acyclic graph (DAG) associated to the workflow. From the point of view of UX design, this opens the possibility for devising languages to express multi-container workflows such as the ones implemented in application testing and scientific study validations.

II. Popper 2.0

The goal of the Popper project [10] is to aid users in the implementation of workflows following a DevOps approach. Last

[1] https://github.com/containers/libpod

year, the Popper team released version 1.0 of the command line tool, which allows users to specify workflows in a lightweight YAML syntax. In Popper 1.0, the nodes in a workflow DAG represent arbitrary Bash shell commands, leaving the burden of ensuring that this commands are portable to users. Additionally, as the project team kept incorporating user feedback, the YAML-based workflow definition syntax kept evolving and, over time, started to look like a workflow specification language. Thus the team decided that it was time to embrace workflow languages properly. Around this time, Github released Github Actions [11] (referred from this point on as GHA), a workflow language and code execution platform. The GHA workflow language is a subset of the HashiCorp Configuration Language (HCL)[2], a popular configuration language used in the DevOps community [12]. The GHA workflow language is one of the simplest workflow formats publicly and openly available; simplicity in this context being defined in terms of the number of syntactic elements of the language. This characteristic, along with the fact that the language specification assumes a containerized environment, made the GHA workflow language a great option for implementing Popper version 2.0.

In the remaining of this section we briefly expand on the GHA workflow language, as well as the design and implementation of the workflow execution engine (Popper 2.0); in particular, we dive into the container runtime API layer.

A. Github Actions Workflow Language

The GHA language is an (open) workflow specification based on containers [13]. Being a workflow language, it specifies a set of tasks and the order in which they should be executed. An example is shown in Fig. 1. A `.workflow` file is made of only two syntactical components: `workflow` and `action` blocks, with either having attributes associated to them. In the case of a `workflow` block, there's only a single attribute (`resolves`) that specifies the list of actions that are to be executed at the end of the workflow. Action blocks define the nodes in the workflow DAG, with a `needs` attribute denoting dependencies among actions, and a `uses` attribute specifying the container that is to be executed. The `uses` attribute can reference Docker images hosted in container image registries; filesystem paths for actions defined locally; or publicly accessible github repositories that contain actions.[3]

In practical terms, an action is a container image that is expected to behave in a constrained manner. The logic within an action has to assume that it is given access to the project directory (the "workspace") where the workflow file is stored and that relative paths are with respect to this workspace folder. In addition, environment variables defined at runtime allow an action to access environmental information such as the absolute path to the workflow on the machine the workflow is running; the commit in the associated Git repository storing the

[2]https://github.com/hashicorp/hcl

[3]See https://developer.github.com/actions for more.

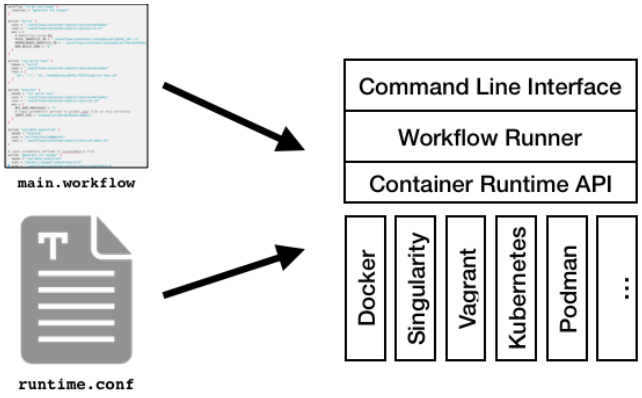

Figure 2: Architecture of the Popper workflow execution engine

`.workflow` file; among others. One of the greatest advantages of this one-container-per-action approach is that, given that the GHA specification is open, anyone can implement actions and publish them on Git repositories, allowing others to reuse them in distinct contexts, creating a bast catalog of reusable actions. The GHA community is continuously growing this catalog where one can find anything from installing python packages, to configuring infrastructure, to manage datasets in data repositories.

B. Workflow Execution Engine

The architecture of the Popper GHA workflow execution engine is shown in Fig. 2. The source code is available at https://github.com/systemslab/popper, released with an open source license. Users interact with the execution engine via its command line interface (CLI). They provide a `.workflow` file and optionally a runtime configuration file. The engine has three main components:

*1) **Command line interface (CLI)**:* Besides allowing users to execute workflows, the CLI provides with search capabilities so that users can search for existing actions implemented by others; allows to visualize workflows by generating diagrams such as the one shown in Fig. 1; generates configuration files for continuous integration systems (e.g. TravisCI, Jenkins, Gitlab-CI, etc.) so that users can continuously validate the workflows they implement.

*2) **Workflow runner**:* The workflow runner is in charge of parsing the `.workflow` file and creating an internal representation for it. It also downloads actions referenced by the workflow, and routes the execution of an action to its corresponding runtime plugin. The runner also creates a cache directory to optimize multiple aspects of execution such (e.g. avoid cloning repositories if they have already cloned previously).

*3) **Container runtime API and plugins**:* This component abstracts the runtime from the. The API exposes generic operations that all runtimes support such as creating a container

image from a `Dockerfile`; downloading images from a registry and converting them to their internal format; and operations on containers such as creation, deletion, renaming, etc. Currently, there are plugins for Docker and Singularity, with others planned by the Popper community.

In addition, the container runtime abstraction layer supports the specialization of a runtime by allowing users to provide a runtime-specific configuration file. This enables users to take advantage of runtime-specific features in a transparent way. For example, in the case of Singularity, this file can contain specific information about the batch scheduler, such that a container can connect to it and launch MPI jobs. This file can either be created by users or be provided by system administrators.

III. Case Study

We now present a case study that exemplifies how to create a workflow as part of a scientific exploration[4]. The example shows an end-to-end workflow corresponding to a parameter sweep execution of the NormalModes software package [14] from the SC19 Student Cluster Competition (SCC) Reproducibility Challenge [15]. This workflow (Fig. 1) goes through the stages of downloading input datasets, building the code associated to the study, running the parameter sweep, validating claims that are made in the original study and producing visualizations of the output.

In the context of the SCC Reproducibility Challenge, having a fully functional workflow upfront reduces significantly the barrier for students to begin being productive. Given that the purpose of the SCC is to have students optimize as much as possible the code for a particular platform, having fully a functional end-to-end workflow allows them to focus right away on what it matters: optimizing compilation, swapping libraries, etc. Compare this to the alternative scenario where students need to start from a tarball and a `README` file. Most of the time in this scenario is spent on tasks that do not provide value to their final goal.

IV. Discussion

A. Benefits

1) ***Tool suitability****:* Fits well the researcher workflow: shell/python/R scripts are still used. Dockerfiles allow them to specify how their environment looks like and GHA allows them to reuse as much as possible.

2) ***Workflow portability****:* From the point of view of users, given that `Dockerfiles` are "universal" (all runtimes support it), a GHA workflow can serve as an abstraction between them and the multitude of runtimes available to them. For example, GHA workflows can be the bridge between HPC and cloud infrastructures. The roadmap of Popper envisions executing seamlessly on Kubernetes by handling all the logic of deploying a scheduler under the hood, or connecting the application to an existing Kubernetes-hosted batch scheduler. This means that from the point of view of a user, they can run the same workflow regardless of the backend infrastructure.

3) ***User friendliness****:* An extension of the above; container-based workflow language such as GHA hides complexities and runtime-specific setups of a container container runtime. The researcher or developer sees `Dockerfile` and `.workflow` files; system administrators configure the runtime and provide a configuration file that users pass to Popper (e.g. GPU, batch schedulers, etc.).

B. Challenges

1) ***Technological barriers****:* Software containers, being a relatively new technology, can be seen a significant time investment given its associated learning curve.

2) ***Cultural barriers****:* Carrying out experimentation or complex application testing by writing workflows is not something practitioners are used to do; following this approach requires a paradigm shift.

3) ***Security and stability concerns****:* Container technology is currently moving relatively fast, with new versions of container runtimes being released every week. In some contexts this might be a deal breaker, as this represents a burden from the point of view of system administrators.

V. Related Work

The problem of implementing multi-container workflows as described in Section I is addressed by existing tools. We briefly survey some of these technologies and compare them with Popper 2.0 by grouping them in categories.

1) ***Workflow languages****:* Workflow languages such as CommonWL [16] and Yadage[5] allow users to specify workflows using a standard, runtime-agnistic language that can be implemented by multiple engines. These languages do not assume a containerized environment, although they support container-based workflows. Compared to the GHA language, their syntax is more complex and give users more control over execution aspects (e.g. specify scheduling policies).

2) ***Kubernetes-based****:* A set of workflow engines that run on Kubernetes, each having their own syntax, allow users to easily submit workflows to a Kubernetes cluster (assuming they have access to one). Examples of these are Pachyderm [17], Argo[6] and Brigade[7]. The main drawback of these when compared to Popper is that the requirement of having access to a Kubernetes cluster.

[4]Code available at https://github.com/popperized/normalmodes-workflows

[5]https://github.com/yadage/yadage

[6]https://github.com/argoproj/argo

[7]https://brigade.sh

*3) **Scientific workflow engines**:* Stable and mature scientific workflow engines such as Pegasus [18], Taverna [19] and NextFlow [20] have introduced native support for containers. Similarly to the previous category, these impose the requirement of needing a workflow engine deployment prior to being able to execute workflows.

*4) **Container-based package managers**:* While not proper workflow execution engines, container-based package managers such as category Snappy[8], Flatpack[9] and AppImage[10] can be employed to implement reproducible workflows. This alternative does not require users to specify their workflows using a workflow language (shell scripts suffice), but assumes that these package managers are available in the Host OS.

BIBLIOGRAPHY

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

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

[3] 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/.

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

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

[6] 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.

[7] Open Container Initiative, "New Image Specification Project for Container Images," *Open Containers Initiative*, Apr. 2016. Available at: https://www.opencontainers.org/blog/2016/04/14/new-image-specification-project-for-container-images.

[8] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, Mar. 2014. Available at: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[9] 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.

[10] 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)*, 2017, pp. 1561–1570.

[11] GitHub, "GitHub Actions: Built by you, run by us," *The GitHub Blog*, Oct. 2018. Available at: https://github.blog/2018-10-17-action-demos/.

[12] Y. Brikman, *Terraform: Up and Running: Writing Infrastructure as Code*, " O'Reilly Media, Inc.", 2017.

[13] GitHub, "An open source parser for GitHub Actions," *The GitHub Blog*, Feb. 2019. Available at: https://github.blog/2019-02-07-an-open-source-parser-for-github-actions/.

[14] J. Shi, R. Li, Y. Xi, Y. Saad, and M.V. de Hoop, "Computing Planetary Interior Normal Modes with a Highly Parallel Polynomial Filtering Eigensolver," *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 71:1–71:13. Available at: https://doi.org/10.1109/SC.2018.00074.

[15] S. Michael and S. Harrel, "From SC Papers to Student Cluster Competition Benchmarks: Joining Forces to Promote Reproducibility in HPC," *SC19*, Apr. 2019. Available at: https://sc19.supercomputing.org.

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

[17] 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.

[18] 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.

[19] 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.

[20] P. Di Tommaso, M. Chatzou, E.W. Floden, P.P. Barja,

[8] https://github.com/snapcore/snapd
[9] https://github.com/flatpak/flatpak
[10] https://github.com/AppImage/AppImageKit

E. Palumbo, and C. Notredame, “Nextflow enables reproducible computational workflows,” *Nature biotechnology*, vol. 35, 2017, p. 316.