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 arisen, each addressing distinct needs (e.g. Singularity, Podman, rkt, among others), and an ongoing effort from the Linux Foundation (Open Container Initiative) 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"
```

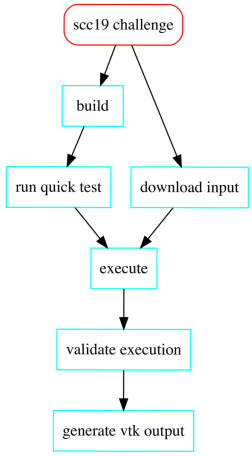

Figure 1: An end-to-end example of a workflow. On the left we have the `.workflow` file that defines the workflow. On the right, a pictorial representation of it.

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

A. Architecture

Here we describe the architecute or the dfd of popper

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

B. YAML as the workflow defination language

Here we show how yml serves better than other config languages

C. Workflow execution engine

Here we describe components of the workflow execution engine

1) ***Command line interface (PopperCLI)****:* Here we talk about the features that PopperCLI provides.

2) ***Workflow runner****:* Here we talk about the work of the Workflow Runner

3) ***Internal representation of the workflow****:* Here we talk about how is the workflow interpreted internally

4) ***Resource managers****:* Here we talk about the functions of the resource manager

5) ***Container runtimes****:* Here we talk about the container runtimes.

6) Continuous evaluation: we talk about continuous evaluation of workflows

III. Case Study

A. Background

1) Docker: Here we talk about docker

2) Singularity: here we talk about singularity

3) SLURM: Here we talk about slurm

4) Kubernetes: Here we talk about Kubernetes

5) CI Services: Here we talk about some CI services

B. Execution Scenarios

Here we describe the 3 different execution scenarios

1) ***Single-Node local workflow execution****:* Single node workflow execution in the local machine for development purposes

2) ***Workflow execution in the Cloud using Kubernetes****:* Escalating the workflow execution to a GPU enabled cluster in the cloud for production grade execution

3) ***Exascale workflow execution in SLURM clusters****:* Planetscale workflow execution is super computing environments

IV. Results

A. Complexity

B. System Resource Usage

C. Overheads

V. Conclusion

A. Benefits

B. Challenges

C. Learning Curve

D. Related Work

Here we talk about other workflow execution solutions and how popper differentiates to them.

1) ***Generic workflow execution engines****:*

2) ***Container native workflow execution engines****:*

3) ***Cloud native workflow execution engines****:*

4) ***Workflow defination languages****:*

VI. Future Work

References

[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.