Towards Shareable and Reproducible Cloud Computing Experiments

Tanu Malik
School of Computing
DePaul University
Chicago, IL USA
tanu.malik@depaul.edu

Samee U. Khan
Department of Electrical and Computer Engineering
Mississippi State University
Starkville, MS, USA
skhan@ece.msstate.edu

***Abstract*—Containerization has emerged as a systematic way of sharing experiments comprising of code, data, and environment. Containerization isolates dependencies of an experiment and allows the computational results to be regenerated. Several new advancements within containerization make it further easy to encapsulate applications and share lighter-weight containers. However, using containerization for cloud computing experiments requires further improvements both at the container runtime level and the infrastructure-level. In this paper, we lay a vision for using containers as a dominant method for efficient sharing and improved reproducibility of cloud computing experiments. We advocate use of container-compliant cloud infrastructures, inclusion of performance profiles of the application or system architecture on which experiments were performed, and methods for statistical comparison across different container executions. We also outline challenges in the achieving this vision and propose existing solutions that can be adapted and propose new methods that can help with automation.**

I. Introduction

In recent years, there has also been a growing concern over the reproducibility of experiments in computer systems research [6]. Reproducibility is obtaining consistent results when using the same or similar input data, computational steps, analysis conditions, etc [12]. Several obstacles to reproducibility have been identified, such as lack of complete descriptions of experimental design, lack of a complete artifact consisting of code, data and environment, poor design of experiments with experiments lacking either sufficient number of runs or workload coverage or both, and poor use of statistical measures for reporting of results [8, 15, 22].

Given the widely noted concerns, some notable practices have emerged that decrease the time and effort required to support the reproducible evaluation. One of the dominant practices is the use of containers, which isolate an experiment's dependencies *viz.* code, data, and environment and ensure the portability of the experiment [1, 5, 11, 13]. Users typically describe the experiment's dependencies, and the container runtime builds and installs the dependencies. The runtime isolates the resulting lightweight binary image from the host environment. The description becomes a shareable transfer unit between interested parties who can build and install the same experiment without much manual work of seeking dependencies and ensuring host environment compatibility. Further, multiple containers can be chained together and used within workflow orchestration.

While containerization is a necessary condition for sharing experiments, it is not sufficient for reproducibility since reproducibility also requires being able to effectively evaluate and verify containerized computations [11, 23]. In the context of cloud computing experiments, this often requires choosing a cloud infrastructure on which to execute and orchestrate containers. Lack of a scalable, reliable and economical infrastructure leads to poor replication of results.

Given a viable infrastructure, the next reproducibility step is to regenerate results. While containers can be built and re-executed, often in system-related experiments, results rely on metrics and measures generated by profiling and measuring the system-under-test (SUT), where the system represents the original cloud infrastructure or a distributed setup. When reproducing the containerized experiment, the new results under the new SUT must be regenerated and compared with past results. However, more often than not, reviewers do not have access to the same SUT. Thus reviewers, to validate, typically simply repeat experiments and regenerate measures and metrics from traces that authors collect and provide within the containerized experiment. While such a validation satisfies the measure's or metric's correctness, they do not establish reproducibility, which often involves checking for consistency under variable SUTs.

Finally, despite regeneration of results, reproducible evaluation entails sound and meaningful performance comparisons. Making fair comparisons between different techniques of the same class, or to benchmark fairly across competing products, especially when the SUT has also undergone a change, is challenging.

As conferences increasingly adapt artifact description and evaluation practices [9, 10], we believe that sharing and reproducing via containers, despite challenges, will be on the rise. For example, The SuperComputing 2023 reproducibility effort mandates use of containers when submitting computational artifacts [20]. Given that the cloud is an integral component of computer systems research and a vast majority of experiments that advance computer systems research are performed on the cloud, there is an urgent need to understand how to

containerize cloud-related experiments share and reproduce them. Cloud-computing solutions such as ChameleonCloud [4] and CloudLab [2] provide bare, metal-as-a-service infrastructure for reproducible evaluation, but currently, do not provide services for container execution and container orchestration: they have to be manually performed by the user.

This paper provides a vision for operating a container-based infrastructure for reproducible analysis of cloud-computing experiments. We begin with simplified creation of containers via reproducible packages with reproducible containers such as Sciunit [23], and namespace-based containers such as Docker and KVM and describe ways how to include performance characteristics within them. We specifically consider experiments in which measurement and analysis itself are not affected by containerization, and in which appropriate statistical measures to measure performance are used. Limiting to these experiments is necessary as it is known that containerization does affect disk and network-I/O intensive applications, especially when there is random disk I/O, and improper use of statistical measures can lead to poor explanability of results.

The rest of the paper is described as follows: We describe the state of reproducible science in computer systems research and review containerization methods and performance tools in the next Section (Section II). We then (in Section III) describe a three-step vision of including container services within existing cloud-based infrastructure, inclusion of performance characterization within containers, and use of statistical measures to compare performance results. We conclude in Section IV.

II. Current State-of-art in Reproducible Evaluation

Definitions. For reproducibility, an experiment is typically defined as a program, an application, or a benchmark. An experiment produces some numerical results that may measure a metric relating to accuracy, or efficiency or involve an aggregate measure. For example, the numerical results of an experiment may measure its accuracy in terms of a loss function measure and its performance in terms of resource utilization (e.g., CPU), latency (e.g., subtracting the start timestamp from the end timestamp), and throughput (e.g., average number of executions in a time window). A computational artifact refers to all the files necessary and sufficient for reproducibility. These often include the software implementation of the experiment, associated workflows, and details about platforms. An artifact may also include established artifacts such as libraries, existing software, and other computational experiments.

According to the National Science Report on Reproducibility and Replicability [12], the reproducibility of such an experiment, implies obtaining consistent computation output and performance measurement results using an equivalent workload and setup. We note that the above definition does not unequivocally define what constitutes an 'experiment' but does define 'conditions of the experiment' as necessary part of an experiment.

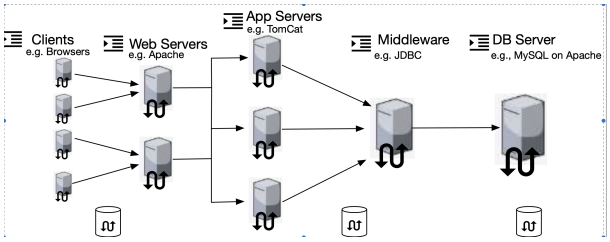

Fig. 1: Performance metric collection.

Given the definitions, we describe a typical setup of a cloud experiment, which aims to analyze complex inter-server interactions. We enumerate components of this experiment necessary and sufficient for reproducible evaluation and distinguish between components that achieve repeatable evaluation versus those that achieve reproducible evaluation.

Consider a n-tier benchmark such as RuBBoS (Rice University Bulletin Board System), which is a representative e-commerce application modeled after bulletin board news sites such as Slashdot. The workload of RUBBoS consists of 25 user operations on the bulletin board database. A typical implementation of RUBBoS consists of the LAMP stack [19]: a Linux server with a Apache web server, a MySQL database server and application and middleware servers such as Tomcat application server, C-JDBC clustering server. Figure 1 describes a representative configuration. There are several such benchmarks viz. RUBiS [18], Cloudstone [3], YCSB [24] and often used for cloud-related experiments.

To analyze complex inter-server interactions resulting from such benchmarks often hosted on n nodes of a cloud, authors typically deploy a variety of fine-grain resource and event monitors, such as `promotheus`, `collectl`, `sar`, `iostat`. These monitors often generate measurement results that include the response time of each query (captured by the client nodes), the resource utilization levels at 50ms intervals for all hardware (and some software) resources such as CPU and memory, and event logs such as the arrival and departure timestamps of every message at every server. Using these monitors complex inter-server interactions, such as queuing of requests [7], configuration effects on bugs [17], and response-time long tail problem can be analyzed.

In a typical setup, authors automate the monitoring procedures via scripts but often the monitors are probes and hooks that interact at the kernel-level. Thus even though the benchmark itself can be containerized and guaranteed to run on different machines, the install of the monitoring procedures must be manually verified since they work at the kernel level. In addition measurements from each of these monitors conforms to different formats and granularities. Scoping the measurements with respect to the n-tier benchmark requires configurations that are application specific. Resolving the timestamp measurements requires log parsing and reconciliation based on variability.

For reproducible evaluation, authors will typically con-

tainerize the benchmark and, instead of sharing the entire setup of the experiment, will simply make the reconciled monitoring traces available—the bare minimum necessary for regenerating the performance and metrics published in the paper. One may argue this is a reasonable scoping of the experiment—it assures the correctness of the experiment, especially since compiling and integrating monitoring frameworks such as `mmapfork` in stress-ng [21] require a couple of hours to build and run and yet it may be non-portable and may fail on heterogeneous cloud machines. However, we believe this does not lead to reproducibility on the cloud because the provided artifact can only be evaluated under the cloud SUT on which the n-tier benchmark was installed and not on any cloud. Given the goal of general reproducibility on the cloud, the question becomes how to reduce the burden of installing, and configuring monitoring frameworks and how to provide a reconciled set of measurements that can apply to *any* SUT.

III. Containerizing Performance Profiles—A Vision

We provide three concrete directions under which progress is needed for reproducible evaluation of cloud experiments.

A. A Container-compliant Cloud Infrastructure

A container encapsulates the artifact and includes the details about how to run the artifact, but the encapsulated artifact must be executed in the cloud. To execute containers in the cloud, one needs a supportive workflow language that can declare the steps needed to orchestrate the various steps across and within containers. In current cloud infrastructure, this is achieved manually by users via template such as XML used by RSpec on GENI and CloudLab, YAML used by Heat on Chameleon and other OpenStack clouds, or AWS CloudFormation in the realm of commercial clouds. Clearly, a specification on one cloud may not translate to other, becoming a reproducibility bottleneck itself. But assuming that such bottlenecks do not arise, a container compliant cloud infrastructure must at least be able to translate the scripts encapsulated within the container to a cloud-specific template.

B. Performance profiles based on light-weight monitors and simulators

The state, message sequencing, scheduling, *etc.*, induce variable execution traces in large-scale computer systems. This further exacerbates the challenge of sampling ultra-high-dimensional data with uncertain system dynamics. We envision a performance profile instead of maintaining sampled data. A performance profile is a characterization of the rates at which a machine can (or is projected to) carry out fundamental operations that is abstract from the particular application. Microbenchmarks are typically used to generate data for a performance profile. For example, a single-machine experiment may use DDR4 memory sticks. The description of the experiment reports that sticks have a specified theoretical peak throughput of 10 GB/s. However, the actual memory bandwidth will be practically much less than the theoretical limit. The performance profile will model based on actual performance numbers obtained.

```
do j=1,lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j),rowstr(j+1)-1
   sum = sum + a(k)*p(colidx(k))
 enddo
 w(j) = sum
enddo
```

Fig. 2: Profiling a simple program

Monitoring all the possible resources is generally not an option, since this might result in enormous performance overhead. In typical performance engineering, a selected set of resources are chosen and the monitoring set is gradually changed based on observed results. For example, if the issue is caused by CPU, a researcher might look into additional data like context switches or interrupts. An experiment can also be deployed on heterogeneous platforms (e.g., XEN vs. KVM, 2-core vs. 4-core), hence the structure of monitoring data may differ from one experiment to another. However, for the purposes of reproducibility the user knows the critical subcomponents that matter to an application and thus can profile those. The intuition is that if the performance profile is obtained over a sufficiently large set of machines, that can serve as the foundation for building a prediction model of the performance of an application when executed on new ("unseen") machines, Further, the user may generate complex profiles by combining performance profiles models of simple profiles, i.e., knowing the correlation between different microbenchmarks is equally important.

An application profile are detailed summary of the fundamental operations an application aims to carry out independently of the machine. There are several ways to obtain such profiles by considering their abstract syntax trees or abstract program control flow as obtained by static analysis. Further application profiles can include statistics such as the frequency of traversal of different paths.

Consider the application profile of an example program (Figure 2) in which the inner loop accesses three arrays, two sequentially and one randomly. Thus the application profile indicates that the ratio of floating point operations to memory loads is 2/3. Several application profile simulators, identify the two stride-one accesses and the one stride-random access. Further, they can even map which code segments to the total dynamic instruction count, and can thus determine which part of the application will the code majorly spend its time.

Containerizing these profiles will require concomitant efforts in standardizing how benchmark data is shared and interpreted, and similarly how profiles should be collected and included within containers.

C. Stochastic Assertions and Comparisons

We envision performance and application profiles to identify patterns, trends, and relations, which generally gets obscured by the massive quantities of data. We assume that such profiles will be stored in a datastore. One issue that may still arise is a comparison of profiles at different granularities. Consider comparing a profile of a performance measure that varies in thousands to something that has variations in the range of hundreds. Profile transformations will be needed to make consistent comparisons.

IRUPs can be used to compare differences between current and past versions of an application. In order to do so, we apply a simple algorithm. Given two profiles A and B, look at first feature in the ranking (highest in the chart). Then, compare the relative importance value for the feature and importance values for A and B. If relative importance does not have the same value, the importance is considered not equivalent and the algorithm stops. If values are similar, we move to the next, less important factor and the compare again. This is repeated for as many features are present in the dataset.

Since comparisons across profiles will only be statistical, there are multiple ways one can make profile comparisons. One way is to design assertions on statistical properties and invariants that can be reasoned over distributions. For instance, one may assert that any consistent execution of the RuBBOS benchmark must not show any performance deviations due to transient bottlenecks. One may then check if repeated executions of the program satisfy these statistical properties and invariants and that the program is "reproducible". Once suitable assertion primitives are formalized, we must develop system artifacts (or operationalize them) for verification — costly instrumentation enablement for computer systems. Another way is to make comparisons based on features. For instance, given two profiles, compare the first feature in the ranking followed by comparing the relative importance value for the feature and the importance values of the two profiles. If relative importance does not have the same value, the importance is considered not equivalent. Otherwise, if values are similar, we move to the next, less important factor and then compare again. This is repeated as many features are present in the dataset.

D. Simulating with Profiles

Finally, we envision simulators that will consume both performance and application profiles and allow simulation of the application behavior across arbitrary topologies, latency and bandwidth characteristics. The objective will not be to simulate the execution of an application on a number of processors different from the number of processors the trace was gathered on, but to simulate the execution of an application on a machine (real or hypothetical) other than that where the trace was gathered.

IV. Conclusion

Containers enable the conduct of reproducible science. However, to use containers for cloud computing experiments, further improvements are needed. In this paper, we highlighted three of them: a cloud infrastructure that supports container orchestration, standard ways of including performance data within containers without increasing their size, and a methodical way of making statistical comparisons. We believe with these improvements the conduct of reproducible science will be transformational.

References

[1] R. Ahmad, et. al., Reproducible Notebook Containers using Application Virtualization. *IEEE eScience, 2022.*

[2] CloudLab project web page [https://www.cloudlab.us/]

[3] Sobel, Will, et al. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0., CCA, 2008.

[4] Chameleon Cloud project web portal, https://rb.gy/mhut1

[5] Docker containers [https://www.docker.com/]

[6] D. G. Feitelson, From repeatability to reproducibility and corroboration, *ACM Operating Systems Review, 2015.*

[7] Kleinrock, L. Queueing systems: Theory. Wiley, 1975.

[8] T. Malik, et. al., SOLE: Towards Descriptive and Interactive Publications. *Implementing Reproducible Research*, CRC, 2014.

[9] Beth Plale et. al., Reproducibility Practice in High-Performance Computing: Community Survey Results. *CiSE*, 23, 55-60, 2021.

[10] T. Malik, et. al., Expanding the Scope of AE at HPC Conferences: Experience of SC21. In *ACM P-RECS* , 2022.

[11] H. Meng, et. al. An Invariant Framework for Conducting Reproducible Computational Science. *Journal of Computational Science*, Elsevier, 2015.

[12] NAS Report, Reproducibility and replicability in science, *National Academies Press*, 2019.

[13] C. Niddodi, et. al., MiDas: Containerizing Data-Intensive Applications with I/O Specialization. *In P-RECS*, 2020.

[14] Jimenez, Ivo, et. al. The POPPER convention: Making reproducible systems evaluation practical. In IPDPS Workshops, 2017.

[15] A. V. Papadopoulos et al., Methodological Principles for Reproducible Performance Evaluation in Cloud Computing, in IEEE Transactions on Software Engineering, 2021.

[16] Q. Pham, et. al. Using Provenance for Repeatability. *USENIX TaPP*, 2013.

[17] Calton Pu, et. al., The Millibottleneck Theory of Performance Bugs and Its Experimental Verification, In ICDCS, 2017.

[18] Rubis. https://github.com/uillianluiz/RUBiS

[19] RUBBoS benchmark. [http://jmob.ow2.org/rubbos.html]

[20] Supercomputing Reproducibility Initiative, 2023, https://rb.gy/8cv9e

[21] stress-ng Suite, https://rb.gy/j5ysg

[22] Victoria Stodden et. al., Implementing reproducible research., *CRC Press*, 2016.

[23] Dai-Hai Ton That, et. al., Sciunits: Reusable research objects, *IEEE e-Science*, 2017.

[24] Cooper, Brian F., et al. Benchmarking cloud serving systems with YCSB. ACM SoCC, 2010.