The Popper Convention: Making Reproducible Systems Evaluation Practical

Ivo Jimenez[u], Michael Sevilla[u], Noah Watkins[u], Carlos Maltzahn[u], Jay Lofstead[s], Kathryn Mohror[l], Remzi Arpaci-Dusseau[w] and Andrea Arpaci-Dusseau[w]

[u]*UC Santa Cruz* [s]*Sandia National Labs* [l]*Lawrence Livermore National Labs* [w]*UW Madison*

Abstract

Independent validation of experimental results in the field of systems research is a challenging task, mainly due to differences in software and hardware in computational environments. Recreating an environment that resembles the original is difficult and time-consuming. In this paper we introduce *Popper*, a convention based on a set of modern open source software (OSS) development principles for generating reproducible scientific publications. Concretely, we make the case for treating an article as an OSS project following a DevOps approach and applying software engineering best-practices to manage its associated artifacts and maintain the reproducibility of its findings. Popper leverages existing cloud-computing infrastructure and DevOps tools to produce academic articles that are easy to validate, reproduce and extend. We present four use cases that illustrate the usefulness of this approach. We show how, by following the *Popper* convention, re-executing experiments on multiple platforms is more practical, allowing reviewers and researchers quickly get to the point of getting results without relying on the original author's intervention.

1. Introduction

A key component of the scientific method is the ability to revisit and replicate previous experiments. Managing information about an experiment allows scientists to interpret and understand results, as well as verify that the experiment was performed according to acceptable procedures. Additionally, reproducibility plays a major role in education since the amount of information that a student has to digest increases as the pace of scientific discovery accelerates. By having the ability to repeat experiments, a student learns by looking at provenance information about the experiment, re-evaluates the questions that the original experiment addressed and builds upon the results of the original study.

Independently validating experimental results in the field of computer systems research is a challenging task [1,2]. Recreating an environment that resembles the one where an experiment was originally executed is a time-consuming endeavour [3,4]. In this work, we revisit the idea of an executable paper [5–8], which proposes the integration of executables and data with scholarly articles to help facilitate its reproducibility. Our approach is to implement it in today's cloud-computing world by treating an article as an open source software (OSS) project. We introduce *Popper*, a convention for organizing an article's artifacts following a DevOps [9,10] approach that allows researchers to make all associated artifacts publicly available with the goal of maximizing automation in the re-execution of experiments and validation of results. There are two main goals for Popper:

1. It should be usable in as many research projects as possible, regardless of their domain.
2. It should abstract underlying technologies without requiring a strict set of tools, making it possible to apply it on multiple toolchains.

This paper describes our experiences with the Popper convention which we have successfully followed to aid in producing papers and classroom lessons that are easy to reproduce. This paper makes the following contributions:

- An analysis of how the DevOps practice can be repurposed to an academic article (Section 2 and Section 3);
- Popper: A methodology for writing academic articles and associated experiments following the DevOps model (Section 4);
- Popper-CLI: an experiment bootstrapping tool that makes Popper-compliant experiments readily available to researchers; and
- Four use cases detailing how to follow Popper in practice (Section 5).

In this work we demonstrate the benefits of following the Popper convention: it brings order to personal research workflows and makes it practical for others to re-execute experiments on multiple platforms with minimal effort, without having to speculate on what the original authors did to compile and configure the system; and shows how automated performance regression testing aids in maintaining the reproducibility integrity of experiments (Section 6 discusses more extensively these points).

2. Experimental Practices

In this section we examine common practices and identify desired features for a new experimental methodology.

2.1 Common Practice

2.1.1 Ad-hoc Personal Workflows

A typical practice is the use of custom bash scripts to automate some of the tasks of executing experiments and analyzing results. From the point of view of researchers, having an ad-hoc framework results in more efficient use of their time, or at least that's the belief. Since these are personalized scripts, they usually hard-code many of the parameters or paths to files in the local machine. Worst of all, a lot of the contextual information is in the mind of researchers. Without a list of guiding principles, going back to an experiment, *even for the original author on the same machine*, represents a time-consuming task.

2.1.2 Sharing Source Code

Version-control systems give authors, reviewers and readers access to the same code base [11] but the availability of source code does not guarantee reproducibility [3]; code may not compile, and even it does, results may differ due to differences from other components in the software stack. While sharing source code is beneficial, it leaves readers with the daunting task of recompiling, reconfiguring, deploying and re-executing an experiment. Things like compilation flags, experiment parameters and results are fundamental contextual information for re-executing an experiment.

2.1.3 Experiment Repositories

An alternative to sharing source code is experiment repositories [12–14]. These allow researchers to upload artifacts associated with a paper, such as input data sets. Similar to code repositories, one of the main problems is the lack of automation and structure for the artifacts. The availability of the artifacts does not guarantee the reproduction of results since a significant amount of manual work needs to be done after these have been downloaded. Additionally, large data dependencies cannot be uploaded since there is usually a limit on the artifact file size.

2.1.4 Virtual Machines

A Virtual Machine (VM) can be used to partially address the limitations of only sharing source code. However, in the case of systems research where the performance is the subject of study, the overheads in terms of performance (the hypervisor "tax") and management (creating, storing and transferring) can be high and, in some cases, they cannot be accounted for easily [15,16]. In scenarios where OS-level virtualization is a viable alternative, it can be used instead of hardware-level virtualization [17].

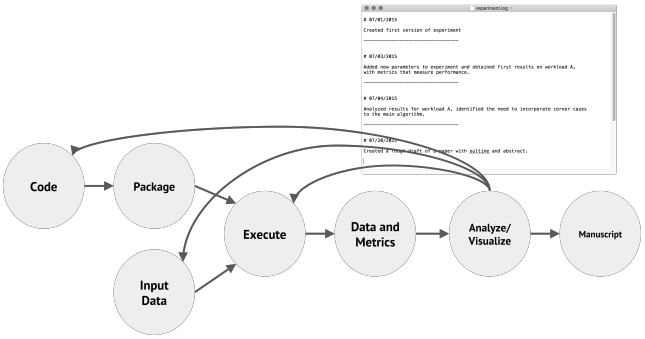

Figure 1: A generic experimentation workflow typically followed by researchers in projects with a computational component. Some of the reasons to iterate (backwards-going arrows) are: fixing a bug in the code of a system, changing a parameter of an experiment or running the same experiment on a new workload or compute platform. Although not usually done, in some cases researchers keep a chronological record on how experiments evolve over time (the analogy of the lab notebook in experimental sciences).

2.1.5 Experiment Packing

Experiment packing entails tracing an experiment at runtime to capture all its dependencies and generating a package that can be shared with others [18–21]. Experiment packing is an automated way of creating a virtual machine or environment and thus suffers from the same limitations: external dependencies such as large datasets cannot be packaged; the experiment is a black-box without contextual information (e.g. history of modifications) that is hard to introspect, making difficult to build upon existing work; and packaging does not explicitly capture validation criteria.

2.1.6 Data Analysis Ad-hoc Approaches

A common approach to analyze data is to capture CSV files and manually paste their contents into Excel or Google Spreadsheets. This manual manipulation and plotting lacks the ability to record important steps in the process of analyzing results, such as the series of steps that took to go from a CSV to a figure. While available (if the spreadsheet is public), it is not immediately clear what a researcher did.

2.1.7 *Eyeball* Validation

Assuming the reader is able to recreate the environment of an experiment, validating the outcome requires domain-specific expertise in order to determine the differences between original and recreated environments that might be the root cause of any discrepancies in the results [1,22,23]. Additionally, reproducing experimental results when the underlying hardware environment changes is challenging mainly due to the inability to predict the effects of such changes in the outcome of an experiment [24,25]. In this case validation is typically done by "eyeballing" figures and the description of exper-

iments in a paper, a subjective task, based entirely on the intuition and expertise of domain-scientists.

2.2 Goals for a New Methodology

A diagram of a generic experimentation workflow is shown in Fig. 1. The problem with current practices is that each of them partially cover the workflow. For example, sharing source code only covers the first task (source code); experiment packing only covers the second one (packaging); and so on. Based on this, we see the need for a new methodology that:

- Is reproducible without incurring any extra work for the researcher. It should require the same or less effort than current practices with the difference of doing things in a systematic way.
- Improves the personal workflows of scientists by having a common methodology that works for as many projects as possible and that can be used as the basis of collaboration.
- Captures the end-to-end workflow in a modern way, including the history of changes that are made to an article throughout its lifecycle.
- Makes use of existing tools (don't reinvent the wheel!). The DevOps toolkit is already comprehensive and easy to use.
- Has the ability to handle large datasets.
- Captures validation criteria in an explicit manner so that subjective evaluation of results of a re-execution is minimized.
- Results in experiments that are amenable to improvement and allows easy collaboration, as well as making it easier to build upon existing work.

3. The DevOps Toolkit

We use the term DevOps [9,10] to refer to a set of common practices that have the goal of expediting the delivery of a software project, allowing to iterate as fast as possible on improvements and new features, without undermining the quality of the software product. In our work, we make the case for achieving the same outcome for academic articles, which can be seen as taking the idea of an executable paper and implementing it with a DevOps approach.

In this section we review and highlight salient features of the DevOps toolkit that makes it amenable to organize all artifacts associated with an academic article. To guide our discussion, we refer to the generic experimentation workflow viewed through a DevOps looking glass shown in Fig. 2. In Section 4 we analyze more closely the composability of these tools and describe general guidelines (the convention) on how to structure projects that make use of the DevOps toolkit

3.1 Version Control

Traditionally the content managed in a version-control system (VCS) is the project's source code; for an academic article the equivalent is the article's content: article text, experiments (code and data) and figures. The idea of keeping an article's source in a VCS is not new and in fact many people follow this practice [6,11]. However, this only considers automating the generation of the article in its final format (usually PDF). While this is useful, here we make the distinction between changing the prose of the paper, changing the parameters of the experiment (both its components and its configuration), as well as storing the experiment results.

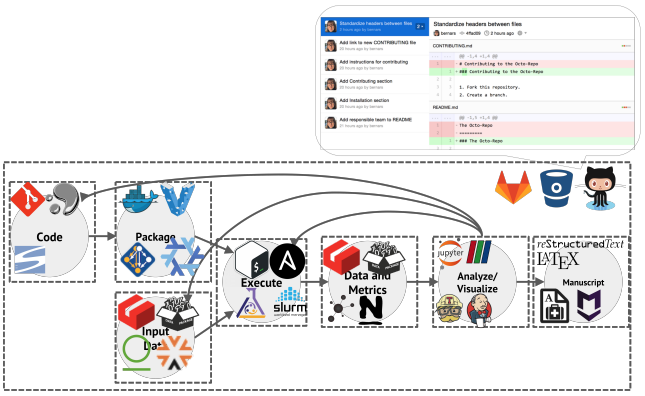

Figure 2: The same workflow as in Fig. 1 viewed through a DevOps looking glass. The logos correspond to commonly used tools from the "DevOps toolkit". From left-to-right, top-to-bottom: git, mercurial, subversion (code); docker, vagrant, spack, nix (packaging); git-lfs, datapackages, artifactory, archiva (input data); bash, ansible, puppet, slurm (execution); git-lfs, datapackages, icinga, nagios (output data and runtime metrics); jupyter, paraview, travis, jenkins (analysis, visualization and continuous integration); restructured text, latex, asciidoctor and markdown (manuscript); gitlab, bitbucket and github (experiment changes).}

Ideally, one would like the entire end-to-end pipeline for all the experiments contained in an article to be managed by a version control system. With the advent of cloud-computing, this is possible for most research articles[1]. One of the mantras of the DevOps movement [26] is to make "infrastructure as code". In a sense, having all the article's dependencies in the same repository is analogous to how large cloud companies maintain monolithic repositories to manage their internal infrastructure [27,28] but at a lower scale.

Tools and services: Git, Svn and Mercurial are popular VCS tools. GitHub and BitBucket are web-based Git repository hosting services. They offer all of the distributed revision control and source code management (SCM) functionality of Git as well as adding their own features. They give new users the ability to look at the entire history of the project and its artifacts.

[1] For large-scale experiments or those that run on specialized platforms, re-executing an experiment might be difficult. However, this does not exclude such research projects from being able to keep the article's associated assets under version control.

3.2 Package Management

Availability of code does not guarantee reproducibility of results [3]. The second main component in the experimentation pipeline is the packaging of applications so that users don't have to do it themselves. Software containers (e.g. Docker, OpenVZ or FreeBSD's jails) complement package managers by packaging all the dependencies of an application in an entire file system snapshot that can be deployed in systems "as is" without having to worry about problems such as package dependencies or specific OS versions. From the point of view of an academic article, these tools can be leveraged to package the dependencies of an experiment. Software containers like Docker have the great potential for being of great use in computational sciences [29].

Tools and services: Docker [30] automates the deployment of applications inside software containers by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux. Alternatives to Docker are modern package managers such as Nix [31] or Spack [32], or even virtual machines.

3.3 Multi-node Orchestration

Experiments that require a set of machines to be orchestrated can make use of a tool that automatically manages binaries, updates packages across machines and drives the end-to-end execution of the experiment. Traditionally, this has been done with an ad-hoc bash script but for experiments that are continually tested there needs to be an automated solution.

Tools and services: Ansible is a configuration management utility for configuring and managing computers, as well as deploying and orchestrating multi-node applications. Similar tools include Puppet, Chef, Salt, among others.

3.4 Bare-metal-as-a-Service

For experiments that are sensitive to the inherent variability associated to executing on consolidated infrastructures (e.g. Amazon's EC2), bare-metal as a service is an alternative.

Tools and services: Cloudlab [33], Chameleon and PRObE [34] are NSF-sponsored infrastructures for research on cloud computing that allows users to easily provision bare-metal machines to execute multi-node experiments. Some cloud service providers such as Amazon allow users to deploy applications on bare-metal instances.

3.5 Dataset Management

Some experiments involve the processing of large input, intermediary or output datasets. While possible, traditional version control tools such as Git were not designed to store large binary files. A proper artifact repository client or dataset management tool can take care of handling data dependencies.

Tools and services: Examples are Apache Archiva [35], Git-LFS [36], Datapackages [37] or Artifactory [38].

3.6 Data Analysis and Visualization

Once an experiment runs, the next task is to analyze and visualize results.

Tools and services: Jupyter notebooks run on a web-based application. It facilitates the sharing of documents containing live code (in Julia, Python or R), equations, visualizations and explanatory text. Other domain-specific visualization tools can also fit into this category. Binder is an online service that allows one to turn a GitHub repository into a collection of interactive Jupyter notebooks so that readers don't need to deploy web servers themselves. Alternatives to Jupyter are Gnuplot, Zeppelin and Beaker. Other scientific visualization such as Paraview tools can also fit in this category.

3.7 Performance Monitoring

Prior to and during the execution of an experiment, capturing performance metrics can be beneficial. In the case of systems research articles, where performance is the main subject of study, capturing performance metrics is fundamental. Instead of creating ad-hoc tools to achieve this we can benefit by adopting and extending existing tools. At the end of the execution, the captured data can be analyzed and many of the graphs included in the article can come directly from running analysis scripts on top of this data.

Tools and services: Many mature monitoring tools exist such as Nagios, Ganglia, StatD, CollectD, among many others. For measuring single-machine baseline performance, tools like Conceptual (network), stress-ng (CPU, memory, file system) and many others exist.

3.8 Continuous Integration

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository frequently with the purpose of catching errors as early as possible. The experiments associated with an article can also benefit from CI. If an experiment's findings can be codified in the form of a unit test, this can be verified on every change to the article's repository.

Tools and services: Travis CI is an open-source, hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. Alternatives to Travis CI are CircleCI, CodeShip. Other on-premises solutions exist such as Jenkins.

3.9 Automated Performance Regression Testing

Open source projects such as the Linux kernel go through rigorous performance testing [39] to ensure that newer versions don't introduce any problems. Performance regression testing is usually an ad-hoc activity but can be automated using high-level languages [40] or statistical techniques [41].

Tools and services: Aver [40] is an example of a language and validation tool that allows authors to express and corroborate statements about the runtime metrics gathered of an experiment.

Listing 1 Sample contents of a Popper repository.

```
paper-repo
| README.md
| .travis.yml
| experiments
|    |-- myexp
|    |    |-- datasets/
|    |    |       |-- input-data.csv
|    |    |-- figure.png
|    |    |-- process-result.py
|    |    |-- setup.yml
|    |    |-- results.csv
|    |    |-- run.sh
|    |    |-- validations.aver
|    |     -- vars.yml
| paper
|    |-- build.sh
|    |-- figures/
|    |-- paper.tex
|     -- references.bib
```

4. The Popper Convention: A DevOps Approach to Producing Academic Papers

The goal for *Popper* is to give researchers a common framework to reason, in a systematic way, about how to structure all the dependencies and generated artifacts associated with an experiment. This convention provides the following unique features:

1. Provides a methodology (or experiment protocol) for generating self-contained experiments.
2. Makes it easier for researchers to explicitly specify validation criteria.
3. Abstracts domain-specific experimentation workflows and toolchains.
4. Provides reusable experiment templates that provide curated experiments commonly used by a research community.

4.1 Self-containment

We say that an experiment is Popper-compliant (or that it has been "Popperized") if all of the following is available, either directly or by reference, in one single source-code repository: experiment code, experiment orchestration code, reference to data dependencies, parametrization of experiment, validation criteria and results. In other words, a Popper repository contains all the dependencies for an article, including its manuscript. The structure of a Popper repo is simple, there are `paper/` and `experiments/` folders, and every experiment has a `datasets/` folder in it.

An example paper project is shown in Lst. 1. A paper repository is composed primarily of the article text and experiment orchestration logic. The actual code that gets executed by an experiment is not part of the repository. This, as well as any large datasets that are used as input to an experiment, reside in their own repositories and are stored in the experiment folder of paper repository as references.

With all these artifacts available, the reader can easily deploy an experiment or rebuild the article's PDF that might include new results. Fig. 3 shows our vision for the reader/reviewer workflow when reading a Popper for a Popperized article. The diagram uses tools we use in the use-case in Section 5.2, like Ansible and Docker, but as mentioned earlier, these can be swapped by equivalent tools. Using this workflow, the writer is completely transparent and the article consumer is free to explore results, re-run experiments, and contradict assetions in the paper.

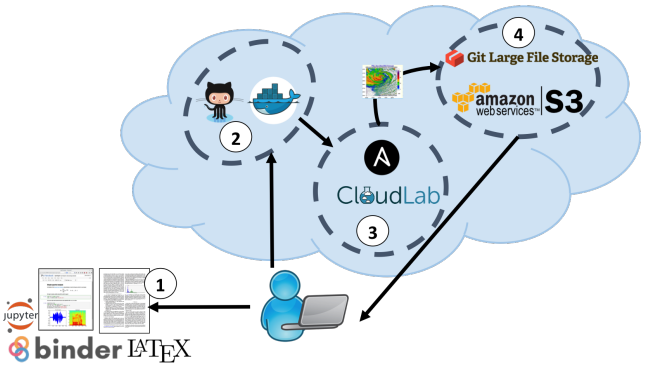

Figure 3: A sample workflow a paper reviewer or reader would use to read a Popperized article. (1) The PDF, Jupyter or Binder are used to visualize and interact with the results post-mortem on the reader's local machine. (2) If needed the reader has the option of looking at the code and clone it locally (GitHub); for single-node experiments, they can be deployed locally too (Docker). (3) For multi-node experiments, Ansible can then be used to deploy the experiment on a public or private cloud (NSF's CloudLab in this case). (4) Lastly, experiments producing large data sets can make use of cloud storage. Popper is tool agnostic, so GitHub can be replaced with GitLab, Ansible with Puppet, Docker with VMs, etc.

A paper is written in any desired markup language. In the above listing we use LATeX as an example (`paper.tex` file). There is a `build.sh` command that generates the output format (e.g. `PDF`). For the experiment execution logic, each experiment folder contains the necessary information such as setup, output post-processing (data analysis) and scripts for generating an image from the results. The execution of the experiment will produce output that is either consumed by a post-processing script, or directly by the scripts that generate an image.

The output can be in any format (CSVs, HDF, NetCDF, etc.), as long as it is versioned and referenced. An important component of the experiment logic is that it should assert the original assumptions made about the environment (a `setup.yml` file in the example), for example, the operating system version (if the experiment assumes one). Also, it is important to parametrize the experiment explicitly (e.g. `vars.yml`), so that readers can quickly get an idea of what is the parameter space of the experiment and what they can modify in order to obtain different results. One common

practice we follow is to place in the caption of every figure a `[source]` link that points to the URL of the corresponding post-processing script in the version control web interface (e.g. GitHub[2]).

4.2 Automated Validation

Validation of experiments can be classified in two categories. In the first one, the integrity of the experimentation logic is checked using existing continuous-integration (CI) services such as TravisCI, which expects a `.travis.yml` file in the root folder. This file contains a specification that consists of a list of tests that get executed every time a new commit is added to the repository. These types of checks can verify that the paper is always in a state that can be built (generate the PDF correctly); that the syntax of orchestration files is correct so that if changes occur, e.g., addition of a new variable, it can be executed without any issues; or that the post-processing routines can be executed without problems.

The second category of validations is related to the integrity of the experimental results. These domain-specific tests ensure that the claims made in the paper are valid for every re-execution of the experiment, analogous to performance regression tests done in software projects that can be implemented using the same class of tools. Alternatively, claims can also be corroborated as part of the analysis code. When experiments are not sensitive to the effects of virtualized platforms, these assertions can be executed on public/free CI platforms (e.g. TravisCI runs tests in VMs). However, when results are sensitive to the underlying hardware, it is preferable to leave this out of the CI pipeline and make them part of the post-processing routines of the experiment. In the example above, an `assertions.aver` file contains validations in the Aver [40] language that check the integrity of runtime performance metrics that claims make reference to. Examples of these type of assertions are: "the runtime of our algorithm is 10x better than the baseline when the level of parallelism exceeds 4 concurrent threads"; or "for dataset A, our model predicts the outcome with an error of 95%". More concrete examples are given in Section 5.

When validating assertions that depend on the underlying hardware, i.e. that come from capturing runtime performance metrics, an important step is to corroborate that the baseline performance of the experiment for a new environment can be reproduced. While this is a similar test that can be codified using performance regression testing as mentioned in the above paragraph, we make the distinction since this step can be executed before any experiment runs. If the baseline performance cannot be reproduced, there is no point in executing the experiment. For example, the results of an experiment that originally ran on an environment consisting of HDDs, with a particular ratio of storage to network bandwidth where the bottleneck resides in storage, results will likely differ from those executed in another environment where the bottleneck is in the network (e.g. because storage is faster). Many of the commonly used orchestration tools incorporate functionality for obtaining "facts" about the environment, information that is useful to have when corroborating assumptions; other monitoring tools such as Nagios [42] can capture raw system-level performance; and existing frameworks such as baseliner [43] are designed to obtain baseline profiles that are associated to experimental results. All these different sources of baseline performance characteristics can serve as a "fingerprint" of the underlying platform and can be given to tools such as Aver so that assertions about the environment are executed before an experiment runs, as a way of sanitizing the execution.

[2] GitHub has the ability to render Jupyter notebooks on its web interface. This is a static view of the notebook (as produced by the original author). In order to have a live version of the notebook, one has to instantiate a Binder or run a local notebook server.

4.3 Toolchain Agnosticism

We designed Popper as a general convention, applicable to a wide variety of environments, from cloud to high-performance computing (HPC). In general, Popper can be applied in any scenario where a component (data, code, infrastructure, hardware, etc.) can be referenced by an identifier, and where there is an underlying tool that consumes these identifiers so that they can be acted upon (install, run, store, visualize, etc.). The core idea behind Popper is to borrow from the DevOps movement [26] the idea of treating every component as an immutable piece of information and provide references to scripts and components for the creation, execution and validation of experiments (in a systematic way) rather than leaving to the reader the daunting task of inferring how binaries and experiments were generated or configured.

We say that a tool is Popper-compliant if it has the following two properties:

1. Assets can be be associated to unique identifiers. Code, packages, configurations, data, results, etc. can all be referenced and uniquely identified.
2. The tool is scriptable (e.g. can be invoked from the command line) and can act upon given asset IDs.

In Section 3 we provided a list of tools for every category of the generic experimentation workflow (Fig. 1) that comply with the two properties given above. In order to illustrate better why these are important, we give examples of non-compliant tools:

- source-code management: Visual SourceSafe (VSS) or StarTeam
- packaging: visual package installers.
- manuscript: Word.
- experiment orchestration: GUI-based workflow systems.
- visualization/analysis: Google spreadsheets

In general, tools that are hard to script e.g. because they don't provide a command-line interface (can only interact via GUI) or they only have a programmatic API for a non-interpreted language, are beyond the scope of Popper.

Listing 2 Initialization of a Popper repo.

```
$ cd mypaper-repo
$ popper init
-- Initialized Popper repo

$ popper experiment list
-- available templates ---------------
ceph-rados        proteustm  mpi-comm-variability
cloverleaf        gassyfs    zlog
spark-standalone  torpor     malacology

$ popper add torpor myexp
```

4.4 Experiment Templates

Researchers that decide to follow Popper are faced with a steep learning curve, especially if they have only used a couple of tools from the DevOps toolkit. To lower the entry barrier, we have developed a command line interface (CLI) tool[3] to help bootstrap a paper repository that follows the Popper convention.

As part of our efforts, we maintain a list of experiment templates that have been "popperized"[4]. These are end-to-end experiments that use a particular toolchain and for which execution, production of results and generation of figures has been implemented (see Section 5 for examples; each use case is available as an experiment in the templates repository). The CLI tool can list and show information about available experiments. Assuming a git repository has been initialized, the tool allows to add experiments to the repository (Lst. 2).

5. Use Cases

We now show use cases for different toolchains that illustrate how Popper is tool-agnostic. Due to space constraints we have to reduce the number of experiments we show for each use case. We refer the reader to this paper's Popper repository https://github.com/systemslab/popper-paper/tree/asplos17/ for more detailed information about the experimental setup as well as more comprehensive results.

5.1 Torpor: Quantifying Cross-platform Performance Variability

Reproducing systems experiments is sometimes challenging due to their sensitivity to the underlying hardware and low-level software stack (firmware and OS). In this use case we exemplify how an experiment that is potentially sensitive to a customized version of the OS can be "Popperized", e.g. because it needs particular features of a custom kernel or specific drivers. Torpor [44] is a workload- and architecture-independent technique for characterizing the performance of a computing platform. Given that the authors of the Torpor paper followed the Popper convention, thus we just have taken the experiment for this use case "as is" and include it in the Popper repository of this article.

[3] https://github.com/systemslab/popper/tree/master/popper

[4] https://github.com/systemslab/popper-templates

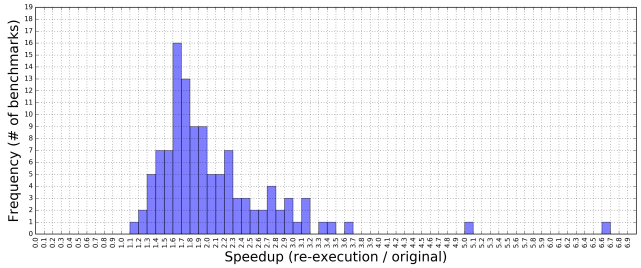

Figure 4: [source] Variability profile of a set of CPU-bound benchmarks. Each data point in the histogram corresponds to the speedup of a stress-ng microbenchmark that a node in CloudLab has with respect to one of our machines in our lab, a 10 year old Xeon. For example, the architectural improvements of the newer machine cause 7 stressors to have a speedup within the `(2.2, 2.3]` range over the base machine.

In short, Torpor works by executing a battery of micro-benchmarks and using these as the performance profile of a system. Given two profiles of two distinct platforms A and B, Torpor generates a variability range of B with respect to A. These variability profiles can then be used to predict the variability of any application running on B, that originally ran A. The goal is to predict as well as recreate performance of applications that run on newer (and faster) platforms using OS-level virtualization.

We take the experiment from the original paper that quantifies the variability of a list of machines with respect to a 10 year old system [44] (Figure 2 of the original article). Due to space constraints, we only show the variability profile for one machine but results for all the other machines are available in this paper's repository. Since we are interested in "pinning" a particular kernel version[5], a natural option is to use a virtual machine to package the experiment. Vagrant [45] is a higher-level wrapper around virtualization software that provides the framework and configuration format (Ruby language scripts) to create and manage complete portable development environments.

The experiment folder contains all necessary files to easily invoke it and generate figures. A bash `run.sh` script installs Vagrant if it is not present; downloads the VM or builds it if links are broken; and lastly executes the experiment. Once the VM runs, the results are placed in the folder where the experiment was invoked (CSV files). The analysis is done in Gnuplot, which in this case runs on the host but could be packaged in a Docker container or another VM. The result of executing the Gnuplot script generates Fig. 4. As mentioned before, the goal of Popper is to provide self-

[5] Strictly speaking, this Torpor experiment doesn't necessarily depend on a particular Linux version but we assume it does to illustrate the need of running a specific version of the kernel.

contained experiments with minimal 3rd party and effort requirements.

5.2 GassyFS: Scalability of an In-memory File System

This use case illustrates how multi-node experiments can be easily ported between multiple platforms. It also exemplifies how the validation criteria of an experiment can be made explicit and be automatically checked with currently available tools.

GassyFS [46] is a new prototype file system system that stores files in distributed remote memory and provides support for checkpointing file system state. The core of the file system is a user-space library that implements a POSIX file interface. File system metadata is managed locally in memory, and file data is distributed across a pool of network-attached RAM managed by worker nodes and accessible over RDMA or Ethernet. Applications access GassyFS through a standard FUSE mount, or may link directly to the library to avoid any overhead that FUSE may introduce. By default all data in GassyFS is non-persistent. That is, all metadata and file data is kept in memory, and any node failure will result in data loss. In this mode GassyFS can be thought of as a high-volume tmpfs that can be instantiated and destroyed as needed, or kept mounted and used by applications with multiple stages of execution. The differences between GassyFS and tmpfs become apparent when we consider how users deal with durability concerns.

Although GassyFS is simple in design, it is relatively complex to setup. The combinatorial space of possible ways in which the system can be compiled, packaged and configured is large. For example, current version of GCC (4.9) has approximately 10^8 possible ways of compiling a binary. For the version of GASNet that we use (2.6), there are 64 flags for additional packages and 138 flags for additional features[6]. To mount GassyFS, we use FUSE, which can be given more than 30 different options, many of them taking multiple values.

In Fig. 5 we show one of multiple experiments that evaluate the performance of GassyFS. We note that while the performance numbers obtained are relevant, they are not our main focus. Instead, we put more emphasis on the goals of the experiments, how we can reproduce results on multiple environments with minimal effort, and how we can ensure the validity of the results. Re-executing this experiment on a new platform only requires to have host nodes to run Docker and to modify the list of hosts given to Ansible (a list of IP addresses), everything else, including the validation of results, is fully automated.

In this experiment we aim to evaluate the scalability of GassyFS, i.e. how it performs when we increase the number of nodes in the underlying GASNet-backed FUSE mount. Fig. 5 shows the results of compiling Git on GassyFS. We observe that once the cluster gets to 2 nodes, performance

[6] These are the flags that are documented. There are many more that can be configured but not shown in the official documentation.

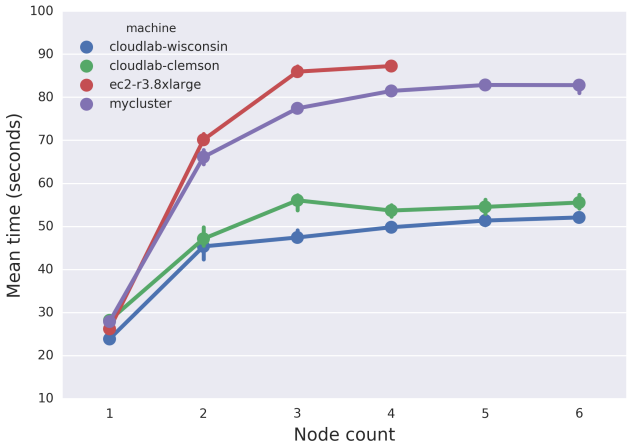

Figure 5: [source] Scalability of GassyFS as the number of nodes in the GASNet cluster increases. The workload in question compiles Git.

degrades sublinearly with the number of nodes. This is expected for workloads such as the one in question. The Aver assertion in Lst. 3 is used to check the integrity of this result.

Listing 3 Assertion to check scalability behavior.

```
when
  workload=* and machine=*
expect
  sublinear(nodes,time)
```

The above expresses our expectation of GassyFS performing sublinearly with respect to the number of nodes. After the experiment runs, Aver is invoked to test the above statement against the experiment results obtained.

5.3 MPI Noisy Neighborhood Characterization

In this use case we exemplify the scenario where OS- and hardware-level virtualization is prohibited such as HPC. In this case, one still can capture most of the software environment by making use of modern package managers such as Spack or Nix. To restore the software stack, one provides the package manager with a list of packages with specific versions and, when necessary, compiler and compilation flags. The package manager then is in charge of restoring the environment, re-compiling binaries if needed. This use case also illustrates the effort required to Popperize an experiment that didn't follow Popper from its inception.

We took an experiment in which an MPI application runs multiple times and its communication performance is measured with mpiP[7] [47]. The goal in this experiment is

[7] **Note to reviewers**. Due to time constraints, we weren't able to prepare the experiment in time so that we could allocate hours in an HPC site to re-run it. The final version of the paper will include a figure with the result of re-executing this experiment.

to identify root causes of variability across executions. The goal of Popperizing this experiment is to fully automate its execution and quantify the effort. The original authors kindly shared the workflow they shared, which consisted of manually installing the software stack, custom Excel spreadsheets with analysis (CSV files) as well as visualization scripts in the Paraview tool. In the end, it took a PhD student approximately one week to Popperize the experiment, with the advice of one of the original authors. Once completed, the end-to-end execution takes care of sanitizing the environment (check kernel and OS versions) and installing packages; execute the LULESH code and capture MPI communication metrics with mpiP. The experiment produces 1) output of simulation (output.hdf5) and 2) generate MPI communication performance metrics. The analysis and visualization scripts run on top of these results to generate the figures.

5.4 Numerical Weather Prediction: A Data-centric Use Case

In this use case we show how to bootstrap a data science paper that follows the Popper convention using the Popper-CLI tool. Popper in this scenario is followed so that datasets are properly referenced and analysis scripts used to process data are versioned and associated to an article.

Initializing a Popper Repository: Our Popper-CLI tool assumes a git repository exists. Given a git repo, we invoke the Popper-CLI tool and initialize the Popper files by issuing a `popper init` command in the root of the git repository. This creates a `.popper.yml` file that contains configuration options for the CLI tool. This file is committed to the paper (git) repository.

Adding a New Experiment: As mentioned before, we maintain a list of experiment templates that have been "Popperized" (Lst. 2). In this example we want to analyze data from an experiment in the area of meteorological sciences (a template created as part of the Big Weather Web project). For this experiment, sensor data has been generated elsewhere and we are interested in properly referencing the dataset, i.e. dataset creation is not part of the experiment. Additionally, the analysis runs on a single machine. Other types of data science projects might involve generating their input datasets and/or process data in a cluster of machines. Popper still can be followed in these scenarios, as shown in previous sections.

This data analysis experiment consists of one dataset and a Jupyter notebook. Relatively large datasets are not managed well by Git, so they should be managed by other tools. We use the datapackage manager tool in this case (third line in Lst. 4). Once the datasets are downloaded, one can open the notebook to visualize and interact with the data analysis of this experiment. The last line above opens a browser and points it to the notebook.

Documenting the Experiment: After we are done with our experiment, we might want to document it and add a paper. The Popper-CLI also provides with manuscript templates. We

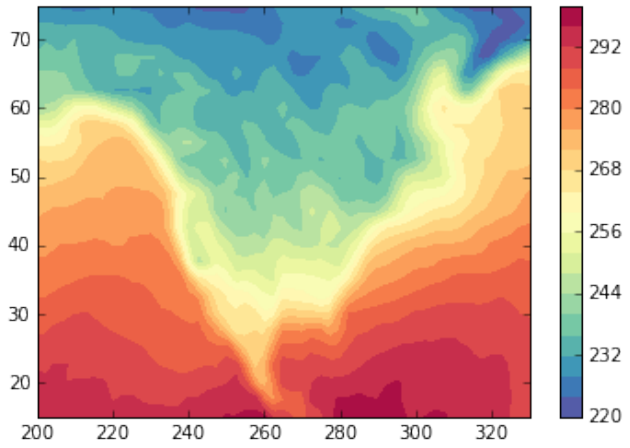

Figure 6: [source] The output of analysis of weather prediction data. The output comes from data processed with the `xarray` Python library. The data corresponds to the NCEP/NCAR Reanalysis 1.

Listing 4 A Data Analysis Experiment.

```
$ popper add jupyter-bww airtemp-analysis
$ cd experiments/airtemp-analysis
$ dpm install datapackages/air-temperature
$ ./visualize.sh
```

can use the generic `article` latex template or other more domain-specific ones. To display the available templates we do `popper paper list`. In this example we will use the latex template for articles that appear in the Bulletin of the American meteorological Society (BAMS).

Let us assume we have a new section in the LATeX file where we describe our experiment. We make use of the figure that we have generated and reference it from the LATeX file. We then regenerate the article (with a `build.sh` command inside the `paper` folder) and see the new image like the one shown in Fig. 6.

5.5 Benefits and Limitations

The use cases in this section illustrate how easier it is to pull an already Popperized experiment (Section 5.1, 5.2, 5.4) than "Popperizing" one (Section 5.3). While it might seem like a burden, at the beginning of an experimental exploration, following Popper quickly pays-off. Consider the common situation of going back to an experiment after a short amount of time and the struggle the represents having to remember what was done, or why things were done in a particular way. However, Popper is not perfect. Obvious issues such as the lack of resources, either because of the use of special hardware or due to the large-scale nature of an experiment, have to be resolved before a Popperized experiment is re-executed. Nevertheless, having access to the original experiment and all associated artifacts is extremely

valuable. Additionally, in some cases, the choice one selects to package an experiment might affect its reproducibility such as in cases where VMs introduce ineligible overheads.

6. The Case for Popper

6.1 We did well for 50 years. Why fix it?

Shared infrastructures "in the cloud" are becoming the norm and enable new kinds of sharing, such as experiments, that were not practical before. Thus, the opportunity of these services goes beyond just economies of scale: by using conventions and tools to enable reproducibility, we can dramatically increase the value of scientific experiments for education and for research. The Popper Convention makes not only the result of a systems experiment available but the entire experiment and allows researchers to study and reuse all aspects of it, making it practical to "stand on the shoulders of giants" by building upon the work of the community to improve the state-of-the-art without having to start from scratch every time.

6.2 The power of "crystallization points."

Docker images, Ansible playbooks, CI unit tests, Git repositories, and Jupyter notebooks are all examples of artifacts around which broad-based efforts can be organized. Crystallization points are pieces of technology, and are intended to be easily shareable, have the ability to grow and improve over time, and ensure buy-in from researchers and students. Examples of very successful crystallization points are the Linux kernel, Wikipedia, and the Apache Project. Crystallization points encode community knowledge and are therefore useful for leveraging past research efforts for ongoing research as well as education and training. They help people to form abstractions and common understanding that enables them to more effectively communicate reproducible science. By having popular tools such as Docker/Ansible as a lingua franca for researchers, and Popper to guide them in how to structure their paper repositories, we can expedite collaboration and at the same time benefit from all the new advances done in the DevOps world.

6.3 Perfect is the enemy of good

No matter how hard we try, there will always be something that goes wrong. The context of systems experiments is often very complex and that complexity is likely to increase in the future. Perfect repeatability will be very difficult to achieve. Recent empirical studies in computer systems [3,4] have brought attention to the main issues that permeate the current practice of our research communities, where scenarios like the lack of information on how a particular package was compiled, or which statistical functions were used make it difficult to reproduce or even interpret results. We don't aim at perfect repeatability but to minimize issues that we currently face and to have a common language that can be used while collaborating to fix all these type of reproducibility issues.

6.4 Drawing the line between packaging and deployment

Figuring out where something should be in the deploy framework (e.g., Ansible) or in the package framework (e.g., Docker) must be standardized by the users of the project. One could implement Popper entirely with Ansible but this introduces complicated playbooks and permantently installs packages on the host. Alternatively, one could use Docker to orchestrate services but this requires "chaining" images together. This process is hard to develop since containers must be recompiled and shared around the cluster. We expect that communities of practice will find the right balance between these technologies, e.g. by improving on the co-design of Ansible playbooks and Docker images within their communities.

6.5 Usability is the key to make this work

The technologies underlying the DevOps development model are not new. However, the open-source software community has significantly increased the usability of the tools involved. Usability is the key to make reproducibility work: it is already hard enough to publish scientific papers, so in order to make reproducibility even practical, the tools have to be extremely easy to use. The Popper Convention enables systems researchers to leverage the usability of DevOps tools.

However, with all great advances in usability, scientists still have to get used to new concepts these tools introduce. In our experience, experimental setups that do not ensure any reproducibility are still a lot easier to create than the ones that do. Not everyone knows git and people are irritated by the number of files and submodules in the paper repository. They also usually misunderstand how OS-level virtualization works and do not realize that there is no performance hit, no network port remapping, and no layers of indirection. Lastly, first encounters with Docker require users to understand that Docker containers do not represent baremetal hardware but immutable infrastructure, i.e. one cannot ssh into them to start services, one need to have a service per image, and one cannot install software inside of them and expect those installations to persist after relaunching a container.

6.6 Numerical vs. Performance Reproducibility

In many areas of computer systems research, the main subject of study is performance, a property of a system that is highly dependant on changes and differences in software and hardware in computational environments. Performance reproducibility can be contrasted with numerical reproducibility. Numerical reproducibility deals with obtaining the same numerical values from every run, with the same code and input, on distinct platforms. For example, the result of the same simulation on two distinct CPU architectures should yield the same numerical values. Performance reproducibility deals with the issue of obtaining the same performance (run time, throughput, latency, etc.) across executions. We set

up an experiment on a particular machine and compare two algorithms or systems.

We can compare two systems with either controlled or statistical methods. In controlled experiments, the computational environment is controlled in such a way that the executions are deterministic, and all the factors that influence performance can be quantified. The statistical approach starts by first executing both systems on a number of distinct environments (distinct computers, OS, networks, etc.). Then, after taking a significant number of samples, the claims of the behavior of each system are formed in statistical terms, e.g. with 95% confidence one system is 10x better than the other. The statistical reproducibility method is gaining popularity, e.g. [4].

Current practices in the systems research community don't include either controlled or statistical reproducibility experiments. Instead, people run several executions (usually 10) on the same machine and report averages. Our convention can be used to either of these two approaches.

6.7 Controlled Experiments become Practical

Almost all publications about systems experiments underreport the context of an experiment, making it very difficult for someone trying to reproduce the experiment to control for differences between the context of the reported experiment and the reproduced one. Due to traditional intractability of controlling for all aspects of the setup of an experiment systems researchers typically strive for making results "understandable" by applying sound statistical analysis to the experimental design and analysis of results [4]. The Popper Convention makes controlled experiments practical by managing all aspects of the setup of an experiment and leveraging shared infrastructure. By providing performance profiles alongside experimental results, this allows to preserve the performance characteristics of the underlying hardware that an experiment executed on and facilitates the interpretation of results in the future.

6.8 DevOps Skills Are Highly Valued by Industry

While the learning curve for the DevOps toolkit is steep, having these as part of the skillset of students or researchers-in-training can only improve their curriculum. Since industry and many industrial/national laboratories have embraced a DevOps approach (or are in the process of embracing), making use of these tools improves their prospects of future employment. In other words, these are skills that will hardly represent wasted time investments, on the contrary, this might be motivation enough for students to learn at least one tool from each of the stages of the DevOps pipeline.

6.9 Popper Complements Existing Efforts

There have been efforts to address the issues in subdomains of the systems research community. We believe Popper complements many of these since it encourages a practice (i.e. to follow a protocol) that applies on top of tools that researchers already know rather than requiring scientists to learn a whole new suite of tools. Some examples of community efforts and projects that Popper complements well are the following.

- Ctuning Foundation's Extended Artifact Description Guide [48] is a set of high-level guidelines for authors on how to prepare an "Artifact Evaluation" appendix for academic articles. Conferences such as Supercomputing, TRUST@PLDI, CGO/PPoP and others are currently making use of it for their reproducibility initiatives. Popper implements a similar pipeline as the one described in the Artifact Description Guide. A Popper repository could even be used instead of an "Artifact Evaluation" appendix.
- Elsevier's 2011 Executable Paper Challenge [49] gave the first prize to the Collage Authoring Environment [50]. Popper is an alternative that makes use of the DevOps toolkit, allowing researchers to keep using their tools but to structure their explorations in a systematic way.
- Proxy applications (Mini-apps) in HPC [51] can be accompanied with a Popper repository to make it easier to validate performance results and facilitate the execution of these on different platforms.
- The Open Encyclopedia of Parallel Algorithmic Features [52]. We envision having a Popper repository for the encyclopedia to make it easier for readers to reuse the algorithms and their insights. Since MediaWiki is already versioned, the wiki and the experiments could reside on the same repository, with the `README` of the experiment being the wiki article, linking to figures that are obtained directly from the algorithm execution output.
- The Journal of Information Systems has recently adopted a new publication model that incentivizes reproducibility by inviting original authors to collaborate with independent reviewers and publish a subsequent paper on whether they could reproduce the original work [53]. By following Popper authors can potentially reduce the amount of work that these subsequent publications entail.

7. Conclusion

In the words of Karl Popper: "*the criterion of the scientific status of a theory is its falsifiability, or refutability, or testability*". The OSS development model and the DevOps practice have proven to be an extraordinary way for people around the world to collaborate in software projects. As the use cases presented here showed, by writing articles following the *Popper* convention, authors can improve their personal workflows, while at the same time generate research that is easier to validate and replicate.

8. Bibliography

[1] J. Freire, P. Bonnet, and D. Shasha, "Computational Reproducibility: State-of-the-art, Challenges, and Database Research Opportunities," *Proceedings of the 2012 ACM*

SIGMOD International Conference on Management of Data, 2012.

[2] G. Fursin, "Collective Mind: Cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning," *arXiv:1308.2410 [cs, stat]*, Aug. 2013. Available at: http://arxiv.org/abs/1308.2410.

[3] C. Collberg, T. Proebsting, and A.M. Warren, "Repeatability and Benefaction in Computer Systems Research," 2015.

[4] T. Hoefler and R. Belli, "Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[5] R. Strijkers, R. Cushing, D. Vasyunin, C. de Laat, A.S.Z. Belloum, and R. Meijer, "Toward Executable Scientific Publications," *Procedia Computer Science*, vol. 4, 2011.

[6] M. Dolfi, J. Gukelberger, A. Hehn, J. Imriska, K. Pakrouski, T.F. Rønnow, M. Troyer, I. Zintchenko, F.S. Chirigati, J. Freire, and D. Shasha, "A Model Project for Reproducible Papers: Critical Temperature for the Ising Model on a Square Lattice," *arXiv:1401.2000 [cond-mat, physics:physics]*, vol. abs/1401.2000, Jan. 2014. Available at: http://arxiv.org/abs/1401.2000.

[7] F. Leisch, M. Eugster, and T. Hothorn, "Executable papers for the R community: The R2 platform for reproducible research," *Procedia Computer Science*, vol. 4, 2011.

[8] T. Kauppinen and G.M. de Espindola, "Linked open science-communicating, sharing and evaluating data, methods and results for executable papers," *Procedia Computer Science*, vol. 4, 2011.

[9] M. Httermann, *DevOps for Developers*, 2012.

[10] M. Loukides, "What is DevOps?" 2012.

[11] C.T. Brown, "How we make our papers replicable," 2014.

[12] V. Stodden, S. Miguez, and J. Seiler, "ResearchCompendia.org: Cyberinfrastructure for Reproducibility and Collaboration in Computational Science," *Computing in Science & Engineering*, vol. 17, Jan. 2015.

[13] V. Stodden, C. Hurlin, and C. Pérignon, "RunMyCode.org: A novel dissemination and collaboration platform for executing published computational results," *2012 IEEE 8th International Conference on E-Science (e-Science)*, 2012.

[14] D.D. Roure, C. Goble, and R. Stevens, "Designing the myExperiment Virtual Research Environment for the Social Sharing of Workflows," *IEEE International Conference on e-Science and Grid Computing*, 2007.

[15] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J.N. Matthews, "Xen and the Art of Repeated Research," *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2004.

[16] G. Klimeck, M. McLennan, S.P. Brophy, G.B.A. Iii, and M.S. Lundstrom, "nanoHUB.org: Advancing Education and Research in Nanotechnology," *Computing in Science & Engineering*, vol. 10, Sep. 2008.

[17] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R.H. Arpaci-Dusseau, and A. Arpaci-Dusseau, "The Role of Container Technology in Reproducible Computer Systems Research," *2015 IEEE International Conference on Cloud Engineering (IC2E)*, 2015.

[18] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational Reproducibility with Ease," *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[19] P.J. Guo and M.I. Seltzer, "BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure," 2012.

[20] Q. Pham, S. Thaler, T. Malik, I. Foster, and B. Glavic, "Sharing and Reproducing Database Applications," *Proc. VLDB Endow.*, vol. 8, Aug. 2015.

[21] A.P. Davison, M. Mattioni, D. Samarkanov, and B. Telenczuk, "Sumatra: A Toolkit for Reproducible Research," *Implementing Reproducible Research*, 2014.

[22] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "Tackling the Reproducibility Problem in Storage Systems Research with Declarative Experiment Specifications," *Proceedings of the 10th Parallel Data Storage Workshop*, 2015.

[23] D.L. Donoho, A. Maleki, I.U. Rahman, M. Shahram, and V. Stodden, "Reproducible Research in Computational Harmonic Analysis," *Computing in Science & Engineering*, vol. 11, Jan. 2009.

[24] R.H. Saavedra-Barrera, "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking," PhD thesis, EECS Department, University of California, Berkeley, 1992.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22Nd An-*

nual International Symposium on Computer Architecture, 1995.

[26] A. Wiggins, "The Twelve-Factor App," *The Twelve-Factor App*, 2011.

[27] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[28] C. Metz, "Google Is 2 Billion Lines of Code's All in One Place," *WIRED*, Sep. 2015.

[29] C. Boettiger, "An introduction to Docker for reproducible research, with examples from the R environment," *arXiv:1410.0846 [cs]*, Oct. 2014. Available at: http://arxiv.org/abs/1410.0846.

[30] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, Mar. 2014.

[31] E. Dolstra and A. Löh, "NixOS: A Purely Functional Linux Distribution," *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[32] T. Gamblin, M. LeGendre, M.R. Collette, G.L. Lee, A. Moody, B.R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[33] R. Ricci and E. Eide, "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications,";*login:* vol. 39, 2014/December.

[34] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "Probe: A thousand-node experimental cluster for computer systems research," *USENIX; login*, vol. 38, 2013.

[35] Apache Archiva Contributors, "Archiva," 2016.

[36] GitHub, "Git Large File Storage," *Git Large File Storage*, 2016.

[37] Open Knowledge International, "Data Packages - Frictionless Data," 2016.

[38] JFrog, "Artifactory," *Managing your binaries*, 2016.

[39] Intel, "Linux Kernel Performance," 2016.

[40] I. Jimenez, C. Maltzahn, A. Moody, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "I Aver: Providing Declarative Experiment Specifications Facilitates the Evaluation of Computer Systems Research," *TinyToCS*, vol. 4, 2016.

[41] T.H. Nguyen, B. Adams, Z.M. Jiang, A.E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Statistical Process Control Techniques," *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 2012.

[42] NagiosEnterprises, "Nagios," *GitHub*, 2016.

[43] I. Jimenez, "Baseliner," *GitHub*, 2016.

[44] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, "Characterizing and Reducing Cross-Platform Performance Variability Using OS-level Virtualization," *The First IEEE International Workshop on Variability in Parallel and Distributed Systems*, 2016.

[45] HashiCorp, "Vagrant," 2016.

[46] N. Watkins, M. Sevilla, and C. Maltzahn, *GassyFS: An In-Memory File System That Embraces Volatility*, UC Santa Cruz, 2016.

[47] A. Bhatele, K. Mohror, S.H. Langer, and K.E. Isaacs, "There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs," *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[48] Ctuning Foundation, "Extended Submission Guide," May. 2016.

[49] Elsevier, "Executable Paper Grand Challenge - Knowledge Enhancement in the Computational Sciences," 2011.

[50] P. Nowakowski, E. Ciepiela, D. Harężlak, J. Kocot, M. Kasztelnik, T. Bartyński, J. Meizner, G. Dyk, and M. Malawski, "The collage authoring environment," *Procedia Computer Science*, vol. 4, 2011.

[51] S. Dosanjh, R. Barrett, M. Heroux, and A. Rodrigues, "Achieving Exascale Computing through Hardware/Software Co-design," *Recent Advances in the Message Passing Interface*, 2011.

[52] V. Voevodin, A. Antonov, and J. Dongarra, "AlgoWiki: An open encyclopedia of parallel algorithmic features," *Supercomputing frontiers and innovations*, vol. 2, 2015.

[53] F. Chirigati, R. Capone, R. Rampin, J. Freire, and D. Shasha, "A collaborative approach to computational reproducibility," *Information Systems*, Mar. 2016.