PopperCI: Automated Reproducibility Validation

Ivo Jimenez, *UC Santa Cruz*, ivo@cs.ucsc.edu
Sina Hamedian, *UC Santa Cruz*, sina@ucsc.edu
Andrea Arpaci-Dusseau, *UW Madison*, dusseau@cs.wisc.edu
Remzi Arpaci-Dusseau, *UW Madison*, remzi@cs.wisc.edu
Jay Lofstead, *Sandia National Labs*, gflofst@sandia.gov
Carlos Maltzahn, *UC Santa Cruz*, carlosm@cs.ucsc.edu
Kathryn Mohror, *Lawrence Livermore National Laboratory*, kathryn@llnl.gov
Robert Ricci, *University of Utah*, ricci@cs.utah.edu

***Abstract*—This paper introduces PopperCI, a continous integration (CI) service hosted at UC Santa Cruz that allows researchers to automate the end-to-end execution and validation of experiments. PopperCI assumes that experiments follow Popper, a convention for implementing experiments and writing articles following a DevOps approach that has been proposed recently. PopperCI runs experiments on public, private or government-fundend cloud infrastructures in a fully automated way. We describe how PopperCI executes experiments and present a use case that illustrates the usefulness of the service.**

I. Introduction

Independently validating experimental results in the field of computer and networking 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]. Additionally, DOE's Office of Advanced Scientific Computing Research (ASCR) and the National Science Foundation (NSF) have been recently stressing the need of requiring grant proposals to include a section on reproducibility, detailing how the research byproducts of a computational project can be replicated [5].

Popper [6,7] is a convention for conducting experiments and writing academic article's following a DevOps [8] approach that allows researchers to generate work that is easy to reproduce. By following Popper, authors make experiment artifacts available to readers, requiring only very high-level instructions to re-execute experiments. While being Popper-compliant doesn't require projects to structure an experiment's artifacts in any particular way, organizing projects in the way it is described here allows experimenters to make use of PopperCI, a continuous integration (CI) service hosted at UC Santa Cruz that allows researchers to automate the end-to-end execution and validation of experiments. Popper is a convention (or methodology) for generating articles, while PopperCI is a service that ensures that the convention is followed.

In this paper we describe how PopperCI automates the execution and validation of experiment implementations without requiring manual intervention. Our contributions are: (1) an organizational structure for Popper-compliant experiments that serves as an interface to setup, execute and validate re-executions; (2) PopperCI, a service that automates the end-to-end execution of *Popperized* experiments; and (3) a use case that illustrates the usefulness of PopperCI.

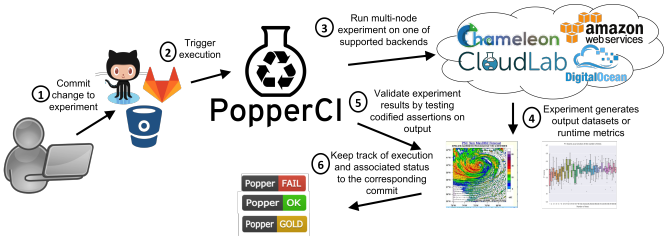

Figure 1: PopperCI automates the execution and validation of experiments. Status of experiments is reported with badges.

The article is organized as follows. We first give a brief description of the Popper convention (Section II-A) and how experiment validations are codified (Section II-B). We then describe PopperCI (Section III), followed by a use case (Section IV). We close with a brief discussion and outline for future work (Section VII).

II. Popper and Experiment Validations

In this section we give a brief introduction to Popper [6,7], in particular we look at how experiment validations are codified.

A. Popper

Popper revisits the idea of an executable paper [9,10], which proposes the integration of executables and data with scholarly articles to help facilitate its reproducibility. The goal of Popper is to implement executable papers in today's cloud-computing world by treating an article as an open source software (OSS) project. Popper is realized in the form of a convention for systematically implementing the different stages of the experimentation process following a DevOps [8] approach (see Fig. 2). The convention can be summarized in three high-level guidelines:

1. Pick a DevOps tool for each stage of the scientific experimentation workflow.
2. Put all associated scripts (experiment and manuscript) in version control, in order to provide a self-contained repository.
3. Document changes as an experiment evolves, in the form of version control commits.

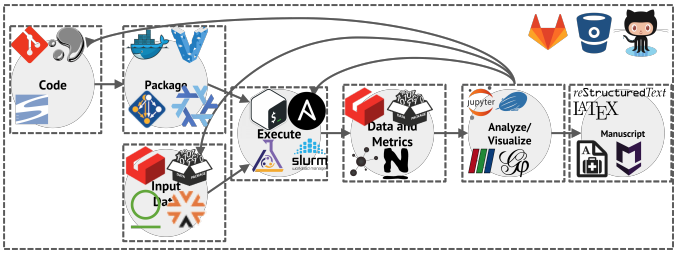

Figure 2: A generic experimentation workflow typically followed by researchers in projects with a computational component viewed through a DevOps looking glass. The logos correspond to commonly used tools from the "DevOps toolkit". From left-to-right, top-to-bottom: Git, Mercurial, Subversion (code); 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, Zeppelin, Paraview, Gephi (analysis and visualization); RestructuredText, LATeX, Asciidoc and Markdown (manuscript); GitLab, Bitbucket and GitHub (experiment changes and labnotebook functionality).

By following these guidelines researchers can make all associated artifacts publicly available with the goal of minimizing the effort for others to re-execute and validate experiments.

B. Experiment Validations

One optional but important component in Popper is the validation of experiments by explicitly codifying expectations. These domain-specific tests ensure that the claims made about the results of an experiment are valid after every re-execution. An example of this is performance regression testing done in software projects (e.g. ScalaMeter). In general, this can be part of the analysis/visualization phase of the experimentation workflow. To illustrate this stage further, consider an experiment that measures the scalability of a system as the number of nodes increases. An assertion to check this might look like the on in Lst. 1.

Listing 1 Example validation in the Aver language.

```
WHEN
  NOT network_saturated AND num_nodes=*
EXPECT
  system_throughput >= (baseline_throughput * 0.9)
```

The above is written in the Aver[1] [11] language and expresses linear scalability with respect to the underlying raw performance, i.e. "regardless of the number of nodes in the system, its throughput is always at least 90% of the raw performance". The boolean value for `network_saturated` comes from network metrics that are captured at runtime. For example, some switches implement the SNMP protocol that allows to identify if the network is getting saturated. In general, for experiments in the computer and networking systems research domain, most of the data that is used at this stage comes from capturing runtime metrics about the underlying resources. Monitoring tools such as Nagios and `collectd` can be used for this purpose. Other examples of this 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 5% at the 95 percent level of confidence".

[1]Aver is a language and tool that can be used to check the integrity of runtime performance metrics that claims make reference to. The tool evaluates simple if-then statements in SQL-like syntax against metrics captured in tabular format files (e.g. CSV files).

III. PopperCI

Following the Popper convention results in producing self-contained experiments and articles, and reduces significantly the amount of work that a reviewer or reader has to undergo in order to re-execute experiments. However, it still requires manual effort in order to re-execute an experiment. For experiments that can run locally where the Popper repository is checked out (e.g. not sensitive to variability of underlying hardware), this is not an issue since usually an experiment is executed by typing a couple of commands to re-execute and validate an experiment. In the case of experiments that need to be executed remotely (e.g. dedicated hardware), this is not as straight-forward since there is a significant amount of effort involved in requesting and configuring infrastructure.

The idea behind PopperCI is simple: by structuring a project in a commonly agreed way, experiment execution and validation can be automated without the need for manual intervention. In addition to this, the status of an experiment (integrity over time) can be tracked by the service hosted at ci.falsifiable.us. In this section we describe the workflow that one follows in order to make an experiment suitable for automation on the PopperCI service. In the next section, we show a use case that illustrates the usage with a concrete example.

A. Experiment Folder Structure

A minimal experiment folder structure for an experiment is shown below:

Listing 2 Basic structure of a Popper repository.

```
$> tree -a paper-repo/experiments/myexp
paper-repo/experiments/myexp/
|-- README.md
|-- .popper.yml
|-- run.sh
|-- setup.sh
|-- validate.sh
```

Every experiment has `setup.sh`, `run.sh` and `validate.sh` scripts that serve as the interface to the experiment. All these return non-zero exit codes if there's a failure. In the case of `validate.sh`, this script should print to standard output one line per validation, denoting whether a validation passed or not. In general, the form for validation results is `[true|false] <statement>`. Examples are shown in Lst. 3.

Listing 3 Example output of validations.

```
[true]  algorithm A outperforms B
[false] network throughput is 2x the IO bandwidth
```

B. Special Subfolders

Folders named after a tool (e.g. `docker` or `terraform`) have special meaning. For each of these, tests are executed that check the integrity of the associated files. For example, if we have an experiment that is orchestrated with Ansible, the associated files are stored in an `ansible` folder. When checking the integrity of this experiment, the `ansible` folder is inspected and associated files are checked to see if they are healthy. The following is a list of currently supported folder names and their CI semantics (support for others is in the making):

- `docker`. An image is created for every `Dockerfile`.
- `ansible`. YAML syntax is checked.
- `datapackages`. Availability of every dataset is checked.
- `vagrant`. Definition of the VM is verified.
- `terraform`. Infrastructure configuration files are checked by running `terraform validate`.
- `geni`. Test using the `omni validate` command.

By default, when a check invokes the corresponding tool, PopperCI uses the latest stable version. If another version is required, users can add a `.popper.yml` file to specify this (Lst. 2).

C. CI Functionality

Assuming users have created an account at the PopperCI website and installed a git hook in their local repository, after a new commit is pushed to the repository that stores the experiments, the service goes over the following steps:

1. Ensure that every versioned dependency is healthy. For example, ensure that external repos can be cloned correctly.
2. Check the integrity of every special subfolder (see previous subsection).
3. For every experiment, trigger an execution (invokes `run.sh`), possibly launching the experiment on remote infrastructure (see next section).
4. After the experiment finishes, execute validations on the output (invoke `validate.sh` command).
5. Keep track of every experiment and report their status.

Once an experiment has been successfully validated by PopperCI, it becomes push-button repeatable. If an experiment has been made public, other users can re-execute it instantly, assuming they have an account at the PopperCI website with the appropriate credentials on the platform where the experiment originally executed (e.g. authentication certificates for CloudLab).

D. Experiment Execution

Experiments that run on remote infrastructure specify any preparation tasks in the `setup.sh` script. For example, an experiment can leverage Terraform to initialize the resources required to execute. In this case, an special `terraform/` folder contains one or more Terraform configuration files (JSON-compatible, declarative format) that specify the infrastructure that needs to be instantiated in order for the experiment to execute. The `run.sh` script assumes that there is a `terraform.tfstate` folder that contains the output of the `terraform apply` command. For example, this folder contains information about whether all the nodes in an experiment have initialized correctly.

Terraform is a generic tool that initializes infrastructure in a platform-agnostic way by interposing an abstraction layer that is implemented using platform-specific tools. When a plugin for a particular infrastructure is not available, one can resort to using platform-specific tools directly. For example CloudLab [12] and Grid500K [13] have a set of CLI tools that can be used to manage the request of infrastructure. In general, any tool that fits in this category that has a command line interface (CLI) tool available can be used to automate this process.

ivotron / my-paper-repository

experiment	commit	message	time	date	status
spark-ml	d4baa54	Using parameter...	23 mins 10 secs	2016-12-20T11:30:32Z	GOLD
spark-ml	fa3135ac	Fixed bug in funct...	22 mins 58 secs	2016-12-20T09:13:47Z	SUCCESS
spark-ml	b92560d	Finished experim...	2 mins 46 secs	2016-12-19T23:56:01Z	FAIL

Figure 3: PopperCI dashboard showing the status of every experiment for every commit.

E. PopperCI Dashboard

The PopperCI website, once users have logged in, shows the status of the experiments for their projects. For each project, there is a table that shows the status of every experiment, for every commit (Fig. 3).

There are three possible statuses for every experiment: `FAIL`, `PASS` and `GOLD`. Clicking an entry on the above table shows a `validations` sub-table with two columns, `validation` and `status`, that shows the status for every validation. There are two possible values for the status of a validation, `FAIL` or `PASS`. When the experiment status is `FAIL`, this list is empty since the experiment execution has failed and validations are not able to execute at all. When the experiment status is `GOLD`, the status of all validations is `PASS`. When the experiment runs correctly but one or more validations fail (experiment's status is `PASS`), the status of one or more validations is `FAIL`.

PopperCI has a badge service that projects can include in the `README` page of a project on the web interface of the version control system (e.g. GitHub). Badges are commonly used to denote the status of a software project, e.g. whether the latest version can be built without errors, or the percentage of code that unit tests cover (code coverage). Badges available for Popper are shown in Fig. 1 (step 6).

F. Popper CLI

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 CLI tool to help bootstrap a paper repository that follows the Popper convention and that makes use of PopperCI. The CLI tool can list and show information about available experiments (Lst. 4).

Listing 4 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 experiment add ceph-rados
```

By default, the tool takes examples from the official Popper repository but other repositories containing Popperized experiments can be queried by passing the `--popper-templates-repo` flag to the `popper experiment` command. This is useful in cases where the associated repository is not public.

The Popper CLI can also be used to trigger an end-to-end execution (locally in the user's machine) of an experiment via the `popper check` command. Additionally, the `popper` binary also contains all the dependencies to launch a self-hosted PopperCI instance via the `popper service` command.

IV. Use Case

We show an experiment being seamlessly executed on multiple platforms. The experiment measures the overhead of Redis, an in-memory key-value store, with respect to the raw memory bandwidth available in a machine. Due to space limitations we leave out the details of the setup but refer the reader to the Popper repository for this paper at github.com/systemslab/popperci-paper.

This experiment makes use of Docker for packaging the software stack, Ansible to orchestrate the logic of the experiment, Aver to verify the output of the experiment and Terraform/geni-lib to specify the resources needed to execute an experiment. When a new commit is pushed to the main branch of this paper repository, a git hook registered at GitHub triggers the execution of the experiment at PopperCI. As mentioned earlier, before the experiment executes, basic checks on the experiment artifacts are executed: Docker images are built, Ansible scripts' syntax is checked and Terraform/GENI configuration files are sanitized.

Terraform's DSL allows to succinctly specify the resources that an experiment needs. An example of how this is specified is shown below (Lst. 5). We have as many of these resource specifications as nodes in an experiment. The resource request for CloudLab uses the Python `geni-lib` library (Lst. 6).

In Fig. 4 we show results of executing the Redis benchmark (the `SET` test in particular) on three different platforms: an on-premises cluster (`UC` nodes), DigitalOcean (`DO` nodes) and CloudLab (`CL` nodes). The y axis shows the slowdown of Redis

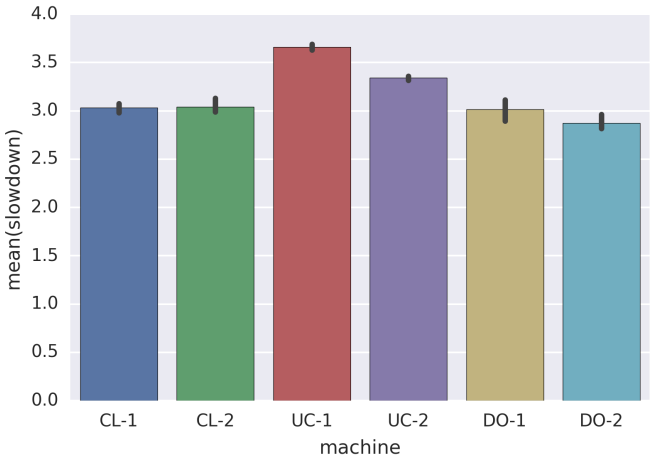

Figure 4: [source] Redis benchmark (`SET` test) results.

w.r.t. STREAM. A value of 1 for machine X means that, in machine X, while STREAM can process 1000 MB/s, Redis processes data at a rate of 1 MB/s. Thus, this graph shows Redis' overhead being within a conservative range of [2.5, 4]. Black lines denote standard deviation ($n = 3$).

Listing 5 Terraform configuartion for requesting a Droplet.

```
resource "digitalocean_droplet" "web" {
    image = "docker-ubuntu-16-04-x64"
    name = "node1"
    region = "sf2"
    size = "16gb"
}
```

The output dataset for this experiment is in tabular format with two columns `machine` and `slowdown`. Thus, the condition that Aver will check on subsequent executions is the following: `WHEN machine=* EXPECT slowdown BETWEEN(2.5, 4)`.

We note that, for the purposes of this paper, while the performance numbers obtained are relevant, they are not our main focus. Instead, we put more emphasis on how we can reproduce results on multiple environments with minimal effort, and how we can verify the outcome of re-executions.

Listing 6 Python script for requesting a node on CloudLab.

```
import geni.portal as portal
import geni.rspec.pg as rspec
request = portal.context.makeRequestRSpec()
node1 = request.RawPC("node1")
node1.disk_image = "urn:publicid:IDN+image//UBUNTU16-64-STD"
portal.context.printRequestRSpec()
```

V. Discussion

A. A Shift in Experimentation Paradigms

Traditional experimentation practices are deeply rooted in the muscle memory of researchers, typing commands in "live"

systems and getting results as they go. Popper (and more generally DevOps) puts an emphasis on versioning every dependency, from infrastructure to any asset required by an execution, with the goal of executing pipelines by providing a list of these versioned artifacts to DevOps tools. In the DevOps world, this is referred to as having "infrastructre-as-code" and basically "anything-as-code" [14]. In practice, this means typing commands in a script file, instead of directly on the CLI, and letting automation tools execute them. By using the PopperCI service (or the `popper check` command of the CLI tool) researchers can force themselves to create the habit of generating Popper-compliant experiments. Additionally, PopperCI incentivizes researchers to follow Popper by providing badges that denote a "reproducibility stamp" that they can refer to as proof of producing reproducible work.

B. Making the Cloud Research-friendly

Shared infrastructures "in the cloud" are becoming the norm and enable new kinds of sharing, such as experiments, that were not practical before. NSF- and DOE-funded infrastructures are a great asset for researchers to use, 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. PopperCI repurposes the DevOps practice for hypothesis-driven, research-oriented projects, with the goal of having the same simplicity and level of maturity as existing DevOps tools and services.

C. Popperized Experiments as Experiment Packages

Our vision is that, over time, as more experiments become "Popperized" and aggregated in the form of Popper template repositories, these can become analogous to software packages that are currently used in the open source software community. With such a list of experiments for a particular community, these experiments then can be indexed so that when a student or researcher looks for preliminary work, they can get to existing, reproducible experiments that they can use as the basis of their work.

D. Repeatability and Replicability

The ACM policy on Artifact Review and Badging[2] introduces the definition of *Repeatability* and *Replicability* for academic articles. PopperCI enables the automatic verification of these designations. Since the `PASS` badge denotes that an experiment was re-executed without errors, this means that an experiment is repeatable. If the results of an experiment are valid (PopperCI `GOLD` badge), and the experiment was executed "by a person or team other than the authors", then results have been replicated.

E. Statistical Studies vs. Controlled Environments

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 and PopperCI make 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.

F. Limitations

One of the main limitations for PopperCI is that it requires users to know at least one tool of the DevOps toolkit for each stage of the generic experimentation workflow. While this learning curve 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. Similarly, the implementation of these processes requires a cultural change for organizations that want to embrace these new approaches.

VI. Related Work

TravisCI is a hosted CI service that allows open source projects on GitHub to connect their work. We strive to have the same simplicity of Travis by choosing "convention over configuration". Jenkins is an open source automation server most commonly used for CI. Additionally, other domain-specific frameworks exist [15]. PopperCI strikes a middle ground: it can be seen as an specialization of generic CI tools and services; at the same time, by having experiments and validations as "first-class citizens" in the CI cycle, PopperCI targets research communities in a domain-agnostic way.

Current experimental practices include the usage of hosted version-control systems to share the source code associated to an experiment. However, availability of source code does not guarantee reproducibility [3]. An alternative to sharing source code is experiment repositories [16] which, due to the lack of common organizational structures for artifacts (no common experimentation workflow), do not solve the issues of validating reproducibility. Another approach is to pack experiments by tracing, at runtime, dependencies and generating a package that can be shared with others [17]. The Popper Convention and PopperCI can be seen as a superset of all these approaches since it embodies all the different stages of the experimentation process.

VII. Conclusion and Future Work

By making use of PopperCI, researchers can ensure that their work is reproducible, making it easier to share and collaborate with others. We are currently working with researchers from other domains such as numeric weather prediction [18] and

[2]https://www.acm.org/publications/policies/artifact-review-badging

mathematical sciences [19] to automate experiments that follow the Popper convention so that they can make use of PopperCI.

While Popper and PopperCI facilitate the re-execution of experiments, they cannot serve for identifying causes of irreproducibility. An open problem is to automate the identification of root causes of irreproducibility, either from changes made to an experiment or from changes in the environment. An even more challenging problem is how to automatically "fix" an experiment, once the cause for differences has been found.

Acknowledgments: This work was partially funded by the Center for Research in Open Source Software[3], Sandia National Laboratories and NSF Awards #1450488 #1338155 #1419199.

VIII. 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*, ACM, 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 and T.A. Proebsting, "Repeatability in Computer Systems Research," *Communications of the ACM*, vol. 59, Feb. 2016.

[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*, ACM, 2015.

[5] H. Johansen, D.E. Bernholdt, B. Collins, M.H. SNL, P. Jones, J.D. Moulton, L. McInnes, J. Carver, and R. Hournung, *Extreme*, 2014.

[6] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, *Popper: Making Reproducible Systems Performance Evaluation Practical, UCSC-SOE-16-10*, UC Santa Cruz, 2016.

[7] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Standing on the Shoulders of Giants by Managing Scientific Experiments Like Software," *USENIX; login*, vol. 41, Nov. 2016.

[8] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook*, O'Reilly Media, 2016.

[9] 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.

[10] 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.

[11] I. Jimenez, C. Maltzahn, J. Lofstead, 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.

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

[13] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *Int. J. High Perform. Comput. Appl.*, vol. 20, Nov. 2006.

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

[15] A.B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister, "DataMill: Rigorous Performance Evaluation Made Easy," *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ACM, 2013.

[16] 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.

[17] 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.

[18] J.P. Hacker, J. Exby, D. Gill, I. Jimenez, C. Maltzahn, T. See, G. Mullendore, and K. Fossell, "A containerized mesoscale model and analysis toolkit to accelerate classroom learning, collaborative research, and uncertainty quantification," *Bulletin of the American Meteorological Society*, Oct. 2016.

[19] I. Jimenez, "Following Popper for Papers in the Mathematical Sciences," *Popper Wiki*, Dec. 2016. Available at: https://github.com/systemslab/popper/wiki/Popper-Math-Science.

[3] http://cross.ucsc.edu