Standing on the Shoulders of Giants

By Managing Scientific Experiments like Software

Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn
Jay Lofstead Kathryn Mohror
Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau

Independently validating experimental results in the field of computer systems research is a challenging task. Recreating an environment that resembles the one where an experiment was originally executed is a time-consuming endeavour. In this article, we present Popper [1], a convention (or protocol) for conducting experiments following a DevOps [2] approach that allows researchers to make all associated artifacts publicly available with the goal of maximizing automation in the re-execution of an experiment and validation of its results.

A basic expectation in the practice of the scientific method is to document, archive, and share all data and the methodologies used so other scientists can reproduce and verify scientific results and students can learn how they were derived. However, in the scientific branches of computation and data exploration the lack of reproducibility has led to a credibility crisis. As more scientific disciplines are relying on computational methods and data-intensive exploration, it has become urgent to develop software tools that help document dependencies on data products, methodologies, and computational environments; that safely archive data products and documentation; and that reliably share data products and documentations so that scientists can rely on their availability.

Over the last decade software engineering and systems administration communities (also referred to as DevOps) have developed sophisticated techniques and strategies to ensure "software reproducibility", i.e. the reproducibility of software artifacts and their behavior using versioning, dependency management, containerization, orchestration, monitoring, testing and documentation. The key idea behind the Popper Convention is to manage every experiment in computation and data exploration as a software project, using tools and services that are readily available now and enjoy wide popularity. By doing so, scientific explorations become reproducible with the same convenience, efficiency, and scalability as software reproducibility while fully leveraging continuing improvements to these tools and services. Rather than mandating a particular set of tools, the Convention does not mandate a particular set of tools but requires that the tool set as a whole implements functionality necessary for software reproducibility. 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.

Common Experimental Practices

To understand the motivation behind Popper, we first take a look at common experimental practices that are currently in use. This allows us to identify the requirements for a new methodology

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.

Sharing Source Code: Version-control systems give authors, reviewers and readers access to the same code base 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.

Experiment Repositories: An alternative to sharing source code is experiment repositories [4]. 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.

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 [5].

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.

***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.

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 and requires 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 and scale to an arbitrary number of machines.
- 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; makes it easier to build upon existing work.

A DevOps Approach to Conducting Experiments

The core idea behind Popper is to borrow from the DevOps movement the idea of treating every component as an immutable piece of information [6] 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. Version control, package management, multi-node orchestration, bare-metal-as-a-service, dataset management, data analysis and visualization, performance monitoring, continuous integration; each of these categories of the DevOps toolkit has a corresponding open source software (OSS) project that is mature, well documented and that is easy to use (see Fig. 2 for examples). The goal for *Popper* is to give researchers a common framework to systematically reason about how to structure all the dependencies and generated artifacts associated with an experiment by making use of these DevOps tools. The 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 templates of curated experiments commonly used by a research community.

Self-containment

We say that an experiment is Popper-compliant (or that it has been "Popperized") if all of the following are 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 one or more experiments, optionally including a manuscript (article or tech report) that documents them.

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 and all input datasets are not part of the repository; instead, they reside in their own repositories and are stored in the `experiments/` folder of the paper repository as references.

With all these artifacts available the reader can easily deploy an experiment or rebuild the article's PDF. 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 presented later, 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 assertions made in the paper.

A paper is written in any desired markup language (LATeX in our), where a `build.sh` command generates the output (e.g. PDF file). 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 experiment output can be in any format (CSV, 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 (`setup.yml`), for example, the operating system version (if the experiment assumes one). Also, it is important to parametrize the experiment explicitly (`vars.yml`), so that readers can quickly get an idea of what is the parameter space and what can be modified 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).

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 specifying what tests to execute. These checks can verify that the paper is always in a state that can be built; 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. 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 [7] language that check the integrity of runtime performance metrics. 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%".

Toolchain Agnosticism

We designed Popper as a general convention, applicable to a wide variety of environments, from cloud to high-performance computing. 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.). 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 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.

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 that provides a list of experiment templates and helps to bootstrap a paper repository that follows the Popper convention (available at https://systemslab.github.io/popper).

Use Case

We now illustrate how to follow the Popper convention when conducting an experiment.

Initializing a Popper Repository: Our Popper-CLI tool assumes a git repository exists (Lst. 2). Given a git repository, we invoke the Popper-CLI tool and initialize Popper 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: Assume the code of the system under study has already been packaged. In order to add an experiment that evaluates a particular aspect of the system, we first start by stating, in either a language such as Aver [7] or in plain text, the result validation criteria. We then proceed with the implementation of the logic of the experiment, mainly orchestration code: configuration, deployment, analysis and visualization of performance metrics, and validation of results. All these files are placed in the paper repository in order to make the experiment Popper-compliant (self-contained).

Obtaining an Existing Experiment: As mentioned before, we maintain a list of experiment templates that have been "Popperized". For this example, assume we select the `gassyfs` template from the list. GassyFS [8] is a new prototype in-memory file system system that stores data in distributed remote memory. 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. Having all this information in a git repository simplifies the setup since one doesn't need to speculate on which things where done by the original authors; all the information is available. In Fig. 4 we show results of an experiment that validates the scalability of GassyFS. We note that while the obtained performance is relevant, it is not our main focus. Instead, we put more emphasis on the goals of the experiment, how we can reproduce results on multiple environments with minimal effort, and how we can validate them. 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 hostnames), everything else, including validation, is fully automated. The Aver [7] assertion in Lst. 3 is used to check the integrity of this result and 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.

Documenting the Experiment: After we are done with an experiment, we might want to document it by creating a report or article. The Popper-CLI also provides with manuscript templates. We can use the generic `article` template or other more domain-specific ones. To display the available templates we do `popper paper list`. In our example we use the template for USENIX articles by issuing a `popper paper add usenix`, which creates a `paper/` folder in the project's root folder, with a sample LATeX file. We then can make reference to figures that have been generated as part of an experiment and reference them from the LATeX file. We then generate the article (all paper templates have a `build.sh` command inside the `paper` folder) and see the new images added to the resulting PDF file.

The Case for Popper

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.

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.

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 [3,9] 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.

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 one), making use of these tools improves the prospects of future employment. 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.

Conclusion

We named the convention Popper as a reference to Karl Popper, the philosopher of science that famously argued that *Falsifiability* should be used as the demarcation criterion when determining if a theory is scientific or pseudo-scientific. 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 case 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.

Figures

Bibliography

[1] 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*

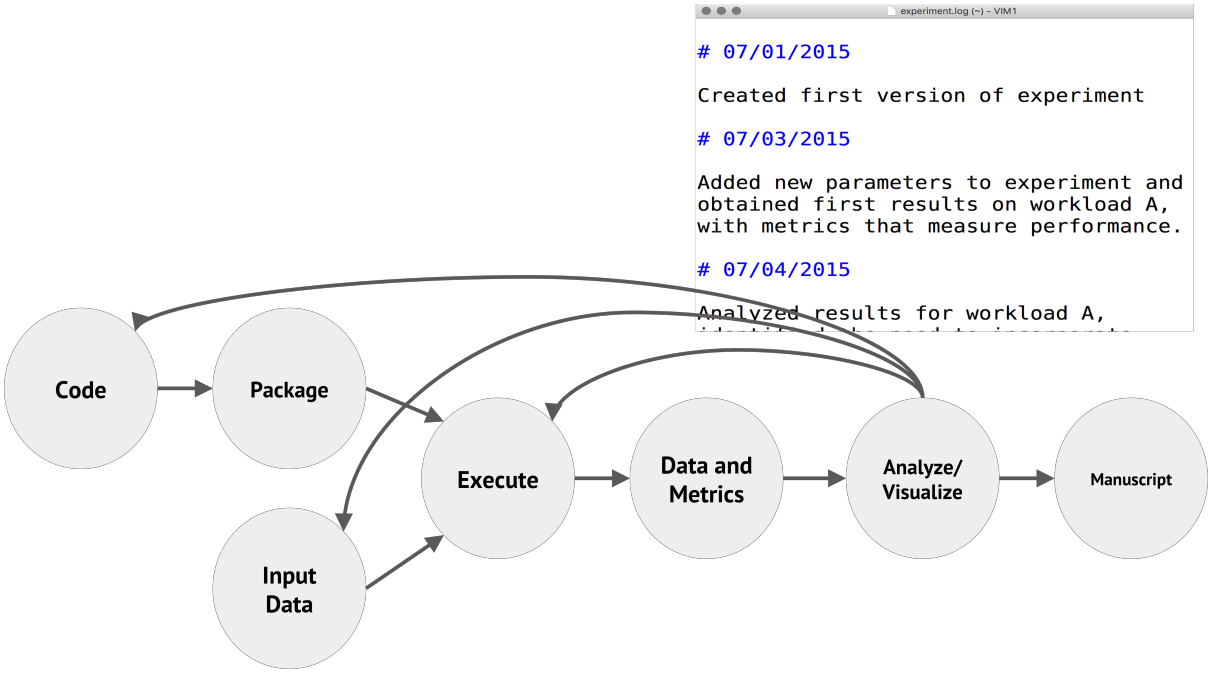

Figure 1: A generic experimentation workflow typically followed by researchers in projects with a computational component. Some of the reasons to iterate (backwards-going arrows) are: fixing a bug in the code of a system, changing a parameter of an experiment or running the same experiment on a new workload 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).

Practical, UC Santa Cruz, 2016.

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

[3] C. Collberg and T.A. Proebsting, "Repeatability in Computer Systems Research," *Communications of the ACM*, vol. 59, Feb. 2016.

[4] 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.

[5] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J.N. Matthews, *Xen and the Art of Repeated Research*, 2004.

[6] A. Wiggins, "The Twelve-Factor App" Available at: http://12factor.net. Available at: http://12factor.net.

[7] 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.

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

[9] 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.

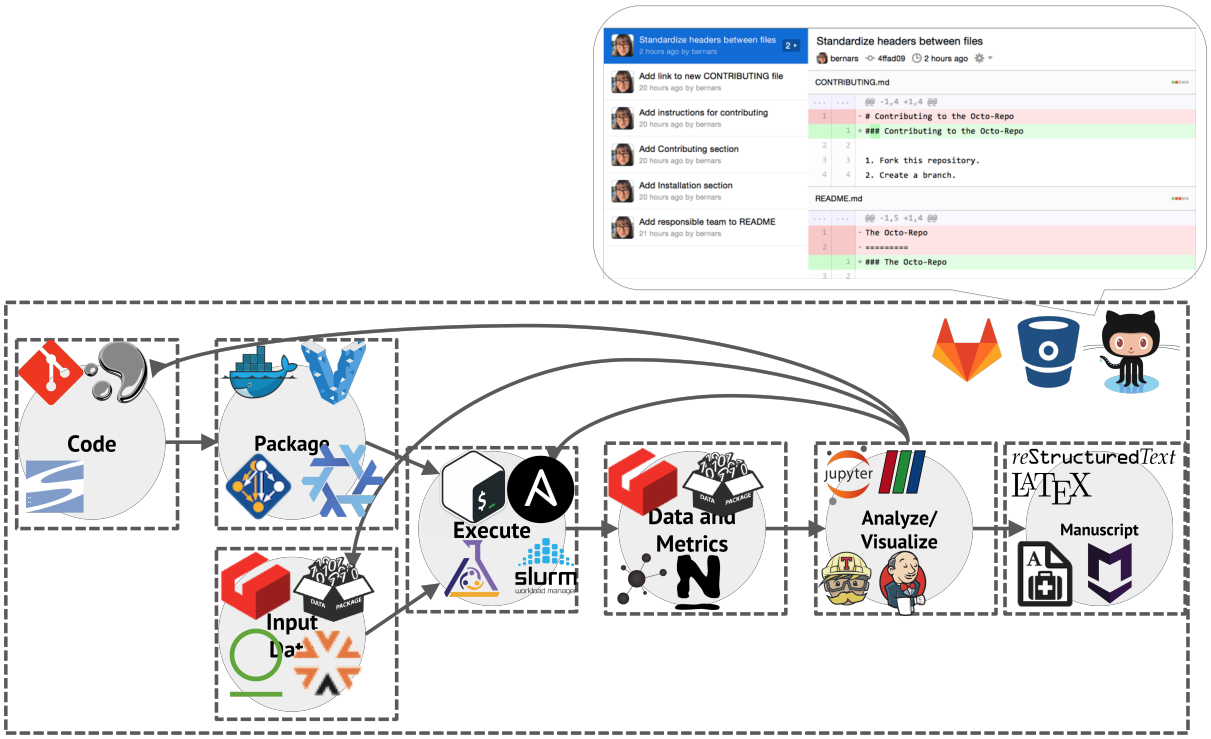

Figure 2: The same workflow as in Fig. 1 viewed through a DevOps looking glass. The logos correspond to commonly used tools from the "DevOps toolkit". From left-to-right, top-to-bottom: git, mercurial, subversion (code); docker, vagrant, spack, nix (packaging); git-lfs, datapackages, artifactory, archiva (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).}

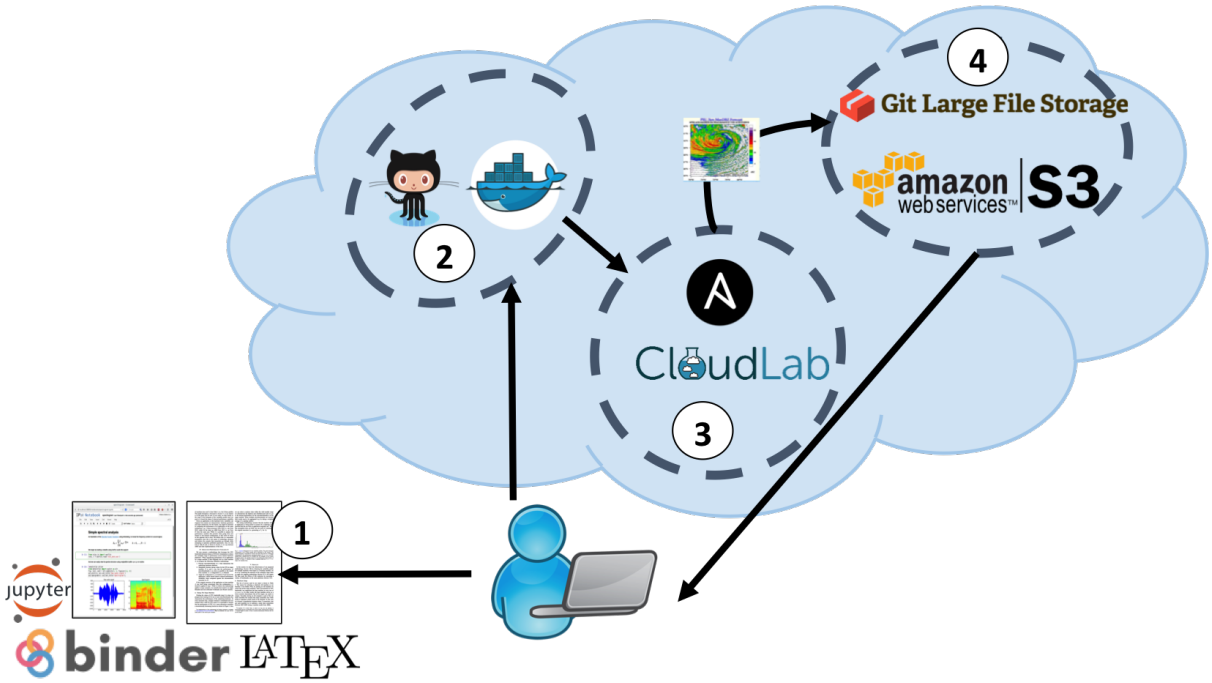

Figure 3: A sample workflow a paper reviewer or reader would use to read a Popperized article. (1) The PDF, Jupyter or Binder are used to visualize and interact with the results post-mortem on the reader's local machine. (2) If needed the reader has the option of looking at the code and clone it locally (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.

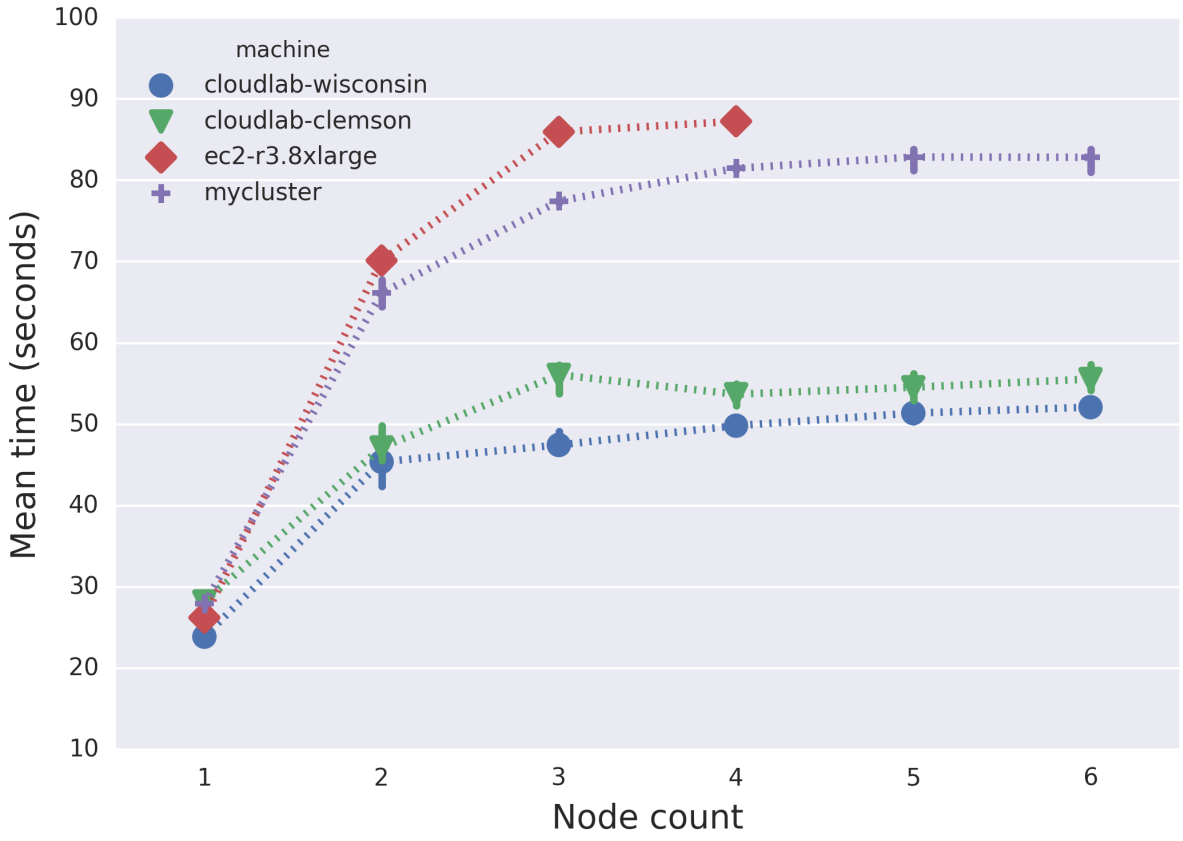

Figure 4: [source] Scalability of GassyFS as the number of nodes in the GASNet cluster increases. The workload in question compiles Git.

Listing 1 Sample contents of a Popper repository.

```
paper-repo
| README.md
| .git/
| .popper.yml
| .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
```

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 gassyfs myexp
```

Listing 3 Assertion to check scalability behavior.

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