Popperize It! Improving The Sustainability of Scholarly Articles By Systematically Following OSS and DevOps Practices

Ivo Jimenez, Michael Sevilla, Carlos Maltzahn (UC Santa Cruz)

Currently, approaches to scientific research require activities that take up much time but do not actually advance our scientific understanding. For example, researchers and their students spend countless hours reformatting data and writing code to attempt to reproduce previously published research. What if the scientific community could find a better way to create and publish our workflows, data, and models to minimize the amount of the time spent "reinventing the wheel"? Popper [1,2] is a protocol and CLI tool for implementing scientific exploration pipelines following a DevOps approach that allows researchers to generate scholarly work that is easy to reproduce. In this whitepaper we introduce Popper and show how can be used to follow OSS and DevOps practices with the goal of producing articles (and associated code) that are accessible and easily re-runnable.

1 Popper Pipelines: A DevOps Approach to Implementing Experimentation Pipelines

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 computational and data science 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 only expects components of an experiment to be scripted.

A common generic analysis/experimentation workflow involving a computational component is the one shown below. We refer to this as a pipeline in order to abstract from experiments, simulations, analysis and other types of scientific explorations. Although there are some projects that don't fit this description, we focus on this model since it covers a large portion of pipelines out there. Typically, the implementation and documentation of a scientific exploration is commonly done in an ad-hoc way (custom bash scripts, storing in local archives, etc.).

The idea behind Popper is simple: make an article self-contained by including in a code repository the manuscript along with every experiment's scripts, inputs, parametrization, results and validation. To this end we propose leveraging state-of-the-art technologies and applying a DevOps[1] approach to the

[1]According to Wikipedia, DevOps, a clipped compound of "development" and "operations", is a software engineering culture and practice that aims at unifying software development (Dev) and software operation (Ops). The main characteristic of the DevOps movement is to strongly advocate automation and monitoring at all steps of software construction, from integration, testing, releasing to deployment and infrastructure management. DevOps aims at shorter development cycles, increased deployment frequency, more dependable releases, in close alignment with business objectives. In practice, following DevOps practices means creating versioned, portable experimentation pipelines by

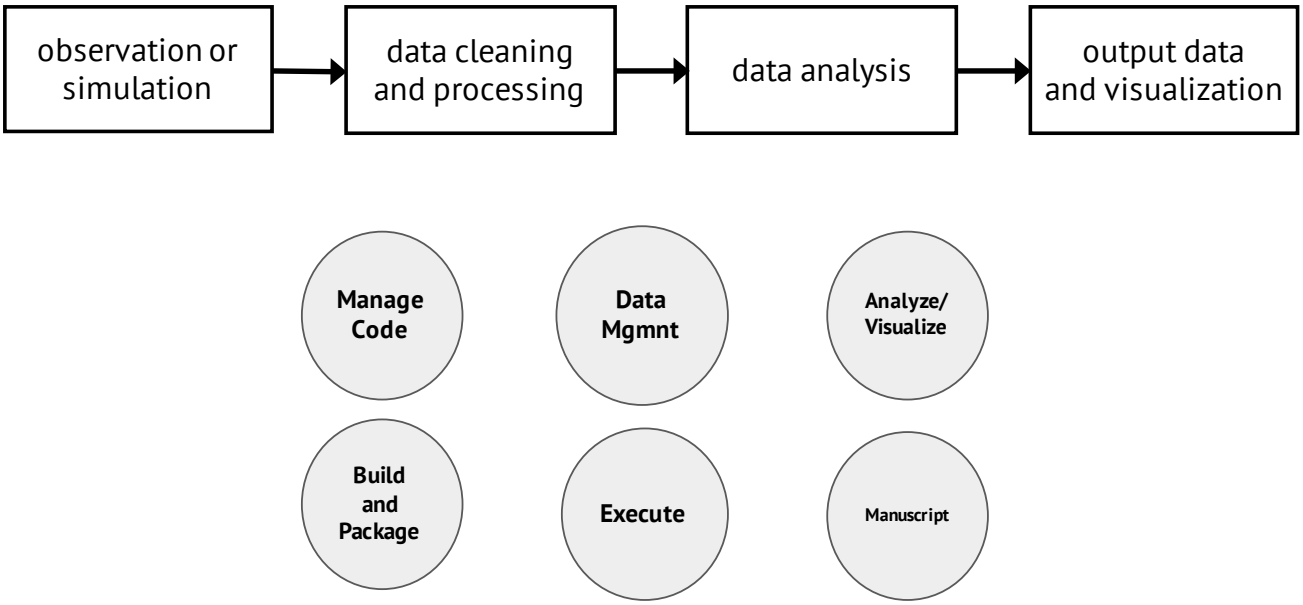

Figure 1: A generic experimentation workflow (modified from the one presented in [3].

implementation of scientific pipelines (also referred to SciOps).

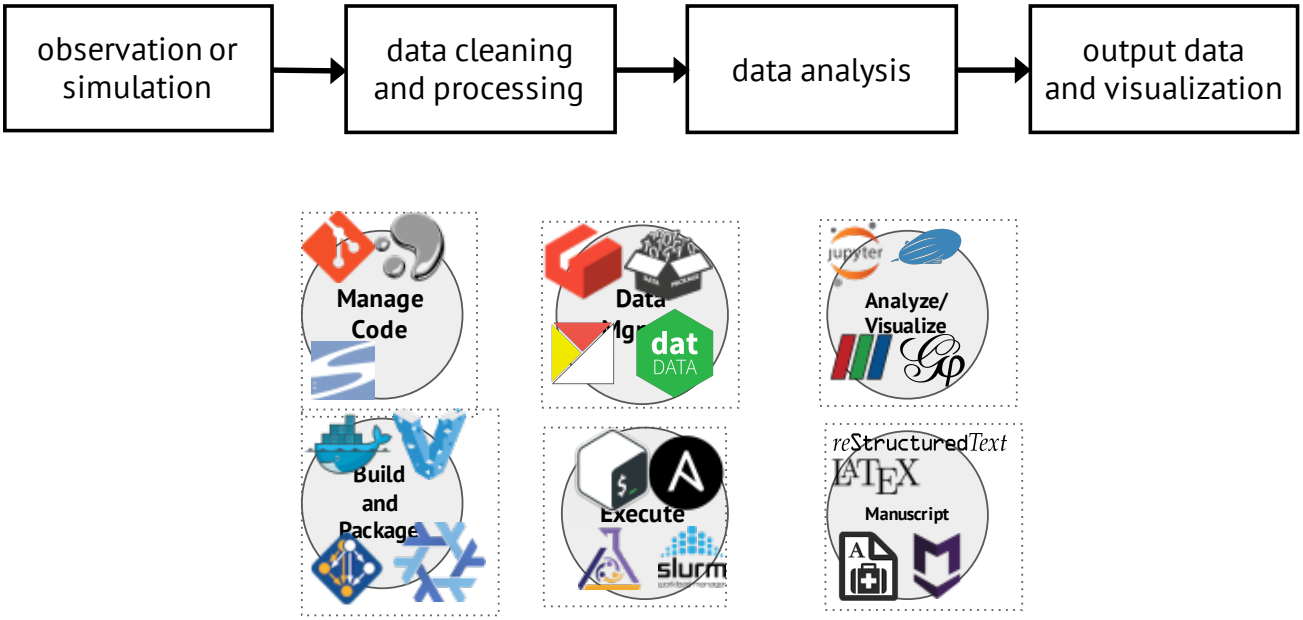

Figure 2: DevOps approach to implementing scientific exploration pipelines, also referred to as SciOps.

Popper is a convention (or protocol) that maps the implementation of a pipeline to software engineering (and DevOps/SciOps) best practices followed in software development communities. If a pipeline is implemented by following the Popper convention, we call it a popper-compliant pipeline or popper pipeline for short. A popper pipeline is implemented using DevOps tools (e.g., version-control systems, lightweight OS-level virtualization, automated multi-node orchestration, continuous integration and web-based data visualization), which makes it easier for others to re-execute and validate their findings.

We say that an article (or a repository) is Popper-compliant if its scripts, dependencies, parameterization, results and validations are all in the same respository (i.e., the pipeline is self-contained). If resources are available, one should be able to easily re-execute a popper pipeline in its entirety. Additionally, the commit log becomes the lab notebook, which makes the history of changes made to it available to readers, an invaluable tool to learn from others and "stand on the shoulder of giants". A

scripting all the steps using automation (DevOps) tools.

"popperized" pipeline also makes it easier to advance the state-of-the-art, since it becomes easier to extend existing work by applying the same model of development in OSS (fork, make changes, publish new findings).

1.1 Repository Structure

The general repository structure is simple: a `paper` and `pipelines` folders on the root of the project with one subfolder per pipeline

```
$> tree mypaper/
  pipelines
      exp1
          README.md
          output
              exp1.csv
              post.sh
              view.ipynb
          run.sh
          setup.sh
          teardown.sh
          validate.sh
      analysis1
          README.md
          ...
      analysis2
           README.md
           ...
  paper
      build.sh
      figures/
      paper.tex
      refs.bib
```

1.2 Pipeline Folder Structure

A minimal pipeline folder structure for an experiment or analysis is shown below:

Listing 1 Basic structure of a Popper repository.

```
$> tree -a paper-repo/pipelines/myexp
paper-repo/pipelines/myexp/
  README.md
  run.sh
  setup.sh
  validate.sh
```

Every pipeline folder contains a set of bash scripts associated to them (`setup.sh`, `run.sh`, and `validate.sh` in the example) that serve as the entry points to each of the stages of a pipeline. 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>` (see examples below).

Listing 2 Example output of validations.

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

2 Popper CLI Tool

The Popper CLI Tool allows users to systematically structure a Git repository following the convention specified earlier. The `popper` command finds information about the structure and location of pipelines by reading the `.popper.yml` file in the root of a project. While this file can be manually created and modified, the `popper` command makes changes to this file depending on which commands are executed.

For example, a project folder might look like the following:

```
$ tree -a -L 2 my-paper
my-paper/
   .git
   .popper.yml
   paper
   pipelines
       analysis
       data-generation
```

This example contains two pipelines named `data-generation` and `analysis`. The `.popper.yml` for this project looks like:

```
pipelines:
  paper:
    envs:
    - host
    path: paper
    stages:
    - build
  data-generation:
    envs:
    - host
    path: pipelines/data-generation
    stages:
    - first
    - second
    - post-run
    - validate
    - teardown
  analysis:
    envs:
    - host
    path: pipelines/analysis
    stages:
    - run
    - post-run
    - validate
    - teardown

metadata:
  author: My Name
```

```
  name: The name of my study

popperized:
  - github/popperized
  - github/ivotron/quiho-popper
```

At the top-level of the YAML file there are entries named `pipelines`, `metadata` and `popperized`. The `pipelines` YAML entry specifies the details for all the available pipelines. The `metadata` YAML entry specifies the set of data that gives information about the user's project. The `popperized` YAML entry specifies the list of Github organizations and repositories that contain popperized pipelines. By default, it points to the `github/popperized` organization. This list is used to look for pipelines as part of the `popper search` command.

Please consult the official documentation for more information about the contents of the `.popper.yml` file and how the CLI tool makes use of it in order to make it.

3 Use Case

We will now show a use case that exemplifies how a Popper pipeline looks in practice. We will implement a pipeline that obtains a dataset of global CO2 emissions from fossil fuels since 1751, and obtain the mean of per capita emissions.

```
popper init --stages=setup,run,validate co2-emissions
$ tree .
.
   pipelines
       co2-emissions
           README.md
           run.sh
           setup.sh
           validate.sh
```

The pipeline consists of 3 stages:

- Setup. Fetch the CSV file and pre-process it so it is ready for our purposes.
- Run. Run a simple python program that groups per capita emissions every 5 years and calculates the mean.
- Validate. We check that the output is as expected and generate a table that can be used in an article.

We will now go over implementing the actual logic on each of these stages.

3.1 Setup

In order to download a public dataset, we can make use of `wget` or `curl`:

```
curl -LO https://github.com/datasets/co2-fossil-global/raw/master/global.csv
```

This dataset has a `Per Capita` column that only contains values from 1950. In order to make it easier to be processed, we'll add zeros to all the missing values.

```
#!/usr/bin/env python
import csv
import sys
```

```
fname = sys.argv[1]
fout = fname.replace('.csv', '') + '_clean.csv'

with open(fname, 'r') as fi, open(fout, 'w') as fo:
    r = csv.reader(fi)
    w = csv.writer(fo)

    # get 0-based index of last column in CSV file
    last = len(next(r)) - 1

    # go back to first line
    fi.seek(0)

    for row in r:
        if not row[last]:
            row[last] = 0
        w.writerow(row)
```

We store the above in a `pipelines/co2-emissions/scripts/add_zeros.py` python file and make it executable (0755 permissions). Putting these together, the `setup.sh` stage looks like the following:

```
#!/bin/bash
# [wf] obtain and clean dataset
set -ex

# [wf] create data folder if it doesn't exist
mkdir -p data/

# [wf] download dataset from github
curl -L \
  -o data/global.csv \
  https://github.com/datasets/co2-fossil-global/raw/master/global.csv

# [wf] add zeros to missing per capita column values
scripts/add_zeros.py data/global.csv
```

3.2 Run

We would like to group the data every 5 years and obtain the main. We can do this with the following Python script:

```
#!/usr/bin/env python
import csv
import sys

from itertools import izip_longest

fname = sys.argv[1]
group_size = int(sys.argv[2])
fout = fname.replace('_clean.csv', '') + '_per_capita_mean.csv'


def grouper(iterable, n, fillvalue=None):
```

```
    args = [iter(iterable)] * n
    return izip_longest(*args, fillvalue=fillvalue)


with open(fname, 'r') as fi, open(fout, 'w') as fo:
    r = csv.reader(fi)
    w = csv.writer(fo)

    # get 0-based index of last column in CSV file
    last = len(next(r)) - 1

    for g in grouper(r, group_size):
        group_sum = 0
        year = 0

        for row in g:
            group_sum += float(row[last])
            year = row[0]

        w.writerow([year, group_sum / 5.0])
```

we store the above script in a `pipelines/co2-emissions/scripts/get_mean.py` (and make it executable). And the `run.sh` script is simple and looks like the following:

```
#!/bin/bash
# [wf] obtain n-year means
set -ex

# [wf] group every n years and obtain mean over each group
scripts/get_mean.py data/global_clean.csv 5
```

3.3 Validate

In this phase, we generate a table in Markdown format from the CSV output we obtained in the previous step. We'll create an executable `pipelines/co2-emissions/scripts/get_mdown_table.py` script containing the following:

```
#!/usr/bin/env python
import csv
import sys

fname = sys.argv[1]
fout = fname.replace('.csv', '') + '.md'

with open(fname, 'r') as fi, open(fout, 'w') as fo:
    r = csv.reader(fi)

    fo.write('| Year | Mean |\n')
    fo.write('| ---- | ---- |\n')

    for row in r:
        fo.write('| {} |\n'.format(' | '.join(row)))
```

We would also like to ensure that we generated data as we expected it on the previous step:

```python
#!/usr/bin/env python
import csv
import sys

fname = sys.argv[1]


with open(fname, 'r') as fi:
    r = csv.reader(fi)

    # get 0-based index of last column in CSV file
    last = len(next(r)) - 1

    for row in r:
        # for years greater than 1950, we should have non-zero mean
        if int(row[0]) < 1950:
            assert float(row[last]) == 0.0
        else:
            assert float(row[last]) != 0
```

We store the above script in a `pipelines/co2-emissions/scripts/validate_output.py` python script and make it executable. Putting all together we have the `validate.sh` stage:

```
#!/bin/bash
# [wf] validate results and get a table
set -ex

# [wf] verify that we got actual result values
scripts/validate_output.py data/global_per_capita_mean.csv

# [wf] generate markdown table
scripts/get_mdown_table.py data/global_per_capita_mean.csv
```

3.4 Test and Commit

After we write all the commands above, we can test them by running `popper check`:

```
cd pipelines/co2-emissions
popper check
```

Once we verify that the pipeline runs OK, we can then commit to the repository:

```
cd ../../
git add .
git commit -m "adding co2-emissions pipeline"
```

3.5 Visualizing a Pipeline

As mentioned in previously, a useful way of abstracting scientific explorations is by thinking in terms of a generic pipeline. While bash scripts from popper pipelines are simple to read, it is useful to have a way of quickly visualizing what a pipeline does. Popper provides the option of generating a call graph for pipelines.

As you may have noticed, some of the comments of the bash scripts we created for all the three stages

start with the `[wf]` prefix. The `popper` command can generate a diagram of the call graph for the pipeline in order to visualize what it does. The `[wf]` comments generate nodes in the call graph, following the convention specified here.

To generate a graph for this pipeline, execute the following:

```
popper workflow co2-emissions
```

The above generates a graph in `.dot` format. To visualize it, you can install the `graphviz` package and execute:

```
popper workflow co2-emissions | dot -T png -o wf.png | open wf.png
```

Alternatively you can use the http://www.webgraphviz.com/ website to generate a graph by copy-pasting the output of the `popper workflow` command.

4 Conclusion

In this whitepaper we have introduced the Popper protocol and CLI tool. Following Popper allows researchers to bring best DevOps practices into their scientific endeavor, which in practice translates in researchers generating scientific exploration pipelines that are versioned, automated and portable. This in turn allows others not only to easily reproduce experiments but also to build upon the work of others.

Acknowledgements: This work was partially funded by the Center for Research in Open Source Software[2], Better Scientific Software Fellowship[3], Sandia National Laboratories, NSF Award #1450488 and DOE Award #DE-SC0016074.

References

[1] 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. Available at: https://www.usenix.org/publications/login/winter-2016-vol-41-no-4/jimenez.

[2] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The Popper Convention: Making Reproducible Systems Evaluation Practical," *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.

[3] J. Kitzes, D. Turek, and F. Deniz, *The practice of reproducible research: Case studies and lessons from the data-intensive sciences*, 2017.

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

[3] http://bssw.io