Popper Documentation

Release 1.0.0

Ivo Jimenez

Nov 16, 2018

Contents

Popper is an experimentation protocol for organizing a academic article's artifacts following a DevOps approach (sometimes referred to as "SciOps"). This documentation describes the experimentation protocol and the Popper CLI tool; it gives examples from multiple domains showing how to follow the protocol; and also shows how to use a CI system to continuously validate Popperized experiments.

CHAPTER 1

Getting Started

Popper is a convention for organizing an academic article's artifacts following a DevOps approach, with the goal of making it easy for others (and yourself!) to repeat an experiment or analysis pipeline.

We first need to install the CLI tool by following these instructions. Show the available commands:

```
popper --help
```

Show which version you installed:

```
popper version
```

Create a project repository (if you are not familiar with git, look here):

```
mkdir mypaper
cd mypaper
git init
echo '# mypaper' > README.md
git add .
git commit -m 'first commit'
```

Initialize the popper repository and add the `.popper.yml` file to git:

```
popper init
git add .
git commit -m 'adds .popper.yml file'
```

1.1 New pipeline

Initialize pipeline using `init` (scaffolding):

```
popper init myexp
```

Show what this did:

```
ls -l pipelines/myexp
```

Commit the "empty" pipeline:

```
git add pipelines/myexp
git commit -m 'adding myexp scaffold'
```

1.2 Popper Run

Run popper run:

```
popper run
```

Once a pipeline is executed, one can show the logs:

```
ls -l pipelines/myexp/popper_logs
```

1.3 Adding Project to GitHub

Create a repository on github, register the remote repository to your local git and push all your commits:

```
git remote add origin git@github.com:<user>/<repo>
git push -u origin master
```

where `<user>` is your username and `<repo>` is the name of the repository you have created.

1.4 Adding Project to Travis

For this, we need to login to Travis CI using our Github credentials. Once this is done, we activate the project so it is continuously validated.

Generate `.travis.yml` file:

```
popper ci --service travis
```

And commit the file:

```
git add .travis.yml
git commit -m 'Adds TravisCI config file'
```

Trigger an execution by pushing to github:

```
git push
```

Go to TravisCI website to see your experiments being executed.

CHAPTER 2

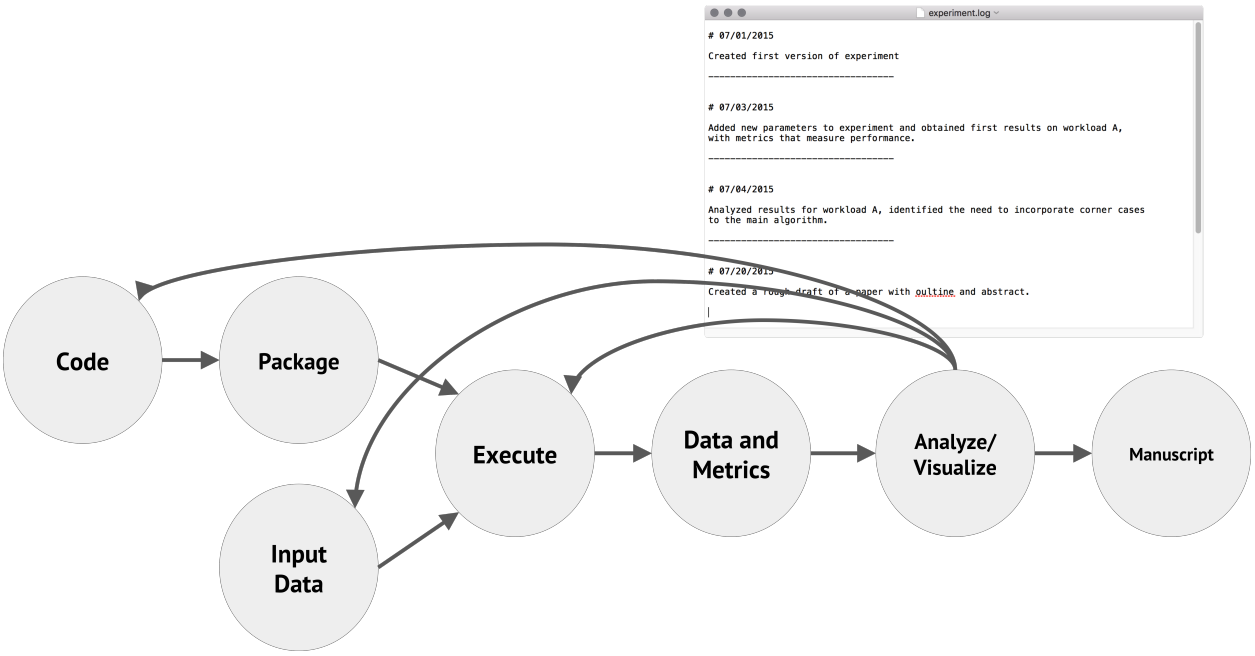
Concepts

2.1 Scientific Exploration 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 protocol 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 repeatable 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. There are two main goals for Popper:

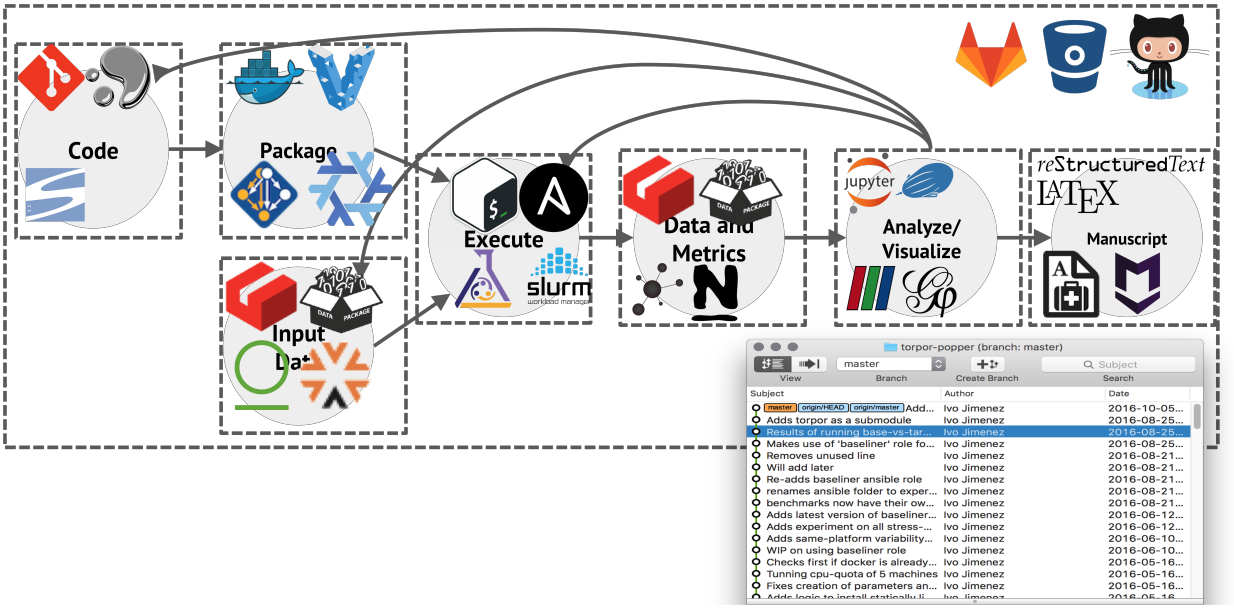
1. It should be usable in as many research projects as possible, regardless of their domain.
2. It should abstract underlying technologies without requiring a strict set of tools, making it possible to apply it on multiple toolchains.

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.).



2.2 Popper Pipelines

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 approach to the implementation of scientific pipelines (also referred to SciOps).



Popper is a convention (or protocol) that maps the implementation of a pipeline to software engineering (and DevOps/SciOps) best-practices followed in open-source software projects. 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 to re-execute and

validate.

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 "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).

2.2.1 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
```

2.2.2 Pipeline Folder Structure

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

```
$> tree -a paper-repo/pipelines/myexp
paper-repo/pipelines/myexp/
├── README.md
├── post-run.sh
├── run.sh
├── setup.sh
├── teardown.sh
└── validate.sh
```

Every pipeline has `setup.sh`, `run.sh`, `post-run.sh`, `validate.sh` and `teardown.sh` scripts that serve as the entrypoints 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).

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

The CLI tool includes a `pipeline init` subcommand that can be executed to scaffold a pipeline with the above structure. The syntax of this command is:

```
popper pipeline init <name>
```

Where `<name>` is the name of the pipeline to initialize. More details on how pipelines are executed is presented in the next section.

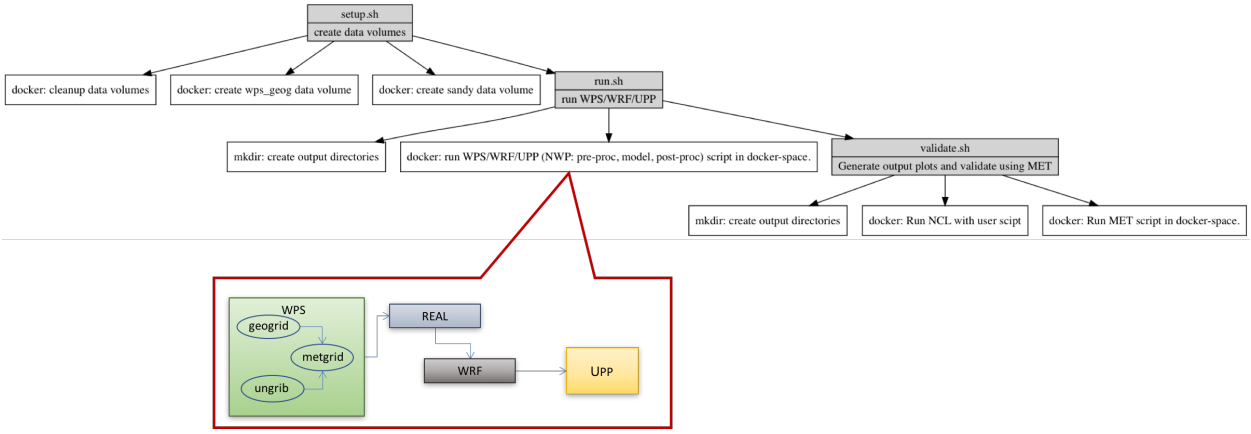
2.3 Popper vs. Other Software

With the goal of putting Popper in context, the following is a list of comparisons with other existing tools.

2.3.1 Scientific Workflow Engines

Scientific workflow engines are "a specialized form of a workflow management system designed specifically to compose and execute a series of computational or data manipulation steps, or workflow, in a scientific application." Taverna and Pegasus are examples of widely used scientific workflow engines. For a comprehensive list, see here.

A Popper pipeline can be seen as the highest-level workflow of a scientific exploration, the one which users or automation services interact with (which can be visualized by doing `popper workflow`). A stage in a popper pipeline can itself trigger the execution of a workflow on one of the aforementioned workflow engines. A way to visualize this is shown in the following image:



The above corresponds to a pipeline whose `run.sh` stage triggers the execution of a workflow for a numeric weather prediction setup (the code is available here). Ideally, the workflow specification files (e.g. in CWP format) would be stored in the repository and be passed as parameter in a bash script that is part of a popper pipeline. For an example of a popper pipeline using the Toil genomics workflow engine, see here.

2.3.2 Virtualenv, Conda, Packrat, etc.

Language runtime-specific tools for Python, R, and others, provide the ability of recreating and isolating environments with all the dependencies that are needed by an application that is written in one of these languages. For exam-

ple `virtualenv` can be used to create an isolated environment with all the dependencies of a python application, including the version of the python runtime itself. This is a lightweight way of creating portable pipelines.

Popper pipelines automate and create an explicit record of the steps that need to be followed in order to create these isolated environments. For an example of a pipeline of this kind, see here.

For pipelines that execute programs written in statically typed languages (e.g. C++), these types of tools are not a good fit and other "full system" virtualization solutions such as Docker or Vagrant might be a better alternative. For an example of such a pipeline, see here.

2.3.3 CI systems

Continuous Integration (CI) is a development practice where developers integrate and deploy code frequently with the purpose of catching errors as early as possible. The pipelines associated to an article can benefit from using CI services. If the output of a pipeline can be verified and validated by codifying any expectation, in the form of a unit test (a command returning a boolean value), this can be tested on every change to pipeline scripts.

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 and CodeShip. Other self-hosted solutions exist such as Jenkins. Each of these services require users to specify and automate tests using their own configuration files (or domain specific languages).

Popper can be seen as a service-agnostic way of automating the execution of a pipeline on CI services with minimal effort. The `popper ci` command generates configuration files that a CI service reads in order to execute a pipeline. Additionally, Popper can be used to test a pipeline locally. Lastly, since the concept of a pipeline and validations associated to them is a first-class citizen in Popper, we can not only check that a pipeline can execute correctly (`SUCCESS` or `FAIL` statuses) but we can also verify that the output is the one expected by the original implementers as explained here (`GOLD` status).

2.3.4 Reprozip / Sciunit

Reprozip "allows you to pack your research along with all necessary data files, libraries, environment variables and options", while Sciunit "are efficient, lightweight, self-contained packages of computational experiments that can be guaranteed to repeat or reproduce regardless of deployment issues". They accomplish this by making use of `ptrace` to track all dependencies of an application. Popper can help in automating the tasks required to install these tools; create and execute Reprozip packages and Sciunits; and re-execute experiments in order to verify that results are being reproduced.

CHAPTER 3

CLI feautures

3.1 New pipeline initialization

Create a Git repository:

```
mkdir mypaper
cd mypaper
git init
echo '# mypaper' > README.md
git add .
git commit -m 'first commit'
```

Initialize the popper repository and add the configuration file to git:

```
popper init
git add .
git commit -m 'adds .popper.yml file'
```

Initialize pipeline using `init` (scaffolding):

```
popper init myexp
```

Show what this did:

```
ls -l pipelines/myexp
```

Commit the "empty" pipeline:

```
git add pipelines/myexp
git commit -m 'adding myexp scaffold'
```

3.2 Executing a pipeline

To automatically run a pipeline:

```
popper run myexp
```

or to execute all the pipelines in a project:

```
popper run
```

Once a pipeline is run, one can show the logs:

```
ls -l pipelines/myexp/popper/host
```

For more on the execution logic, see here.

3.3 Specifying environment requirements

The `require` subcommand can be used to specify expectations on the environment, in particular, the availability of certain environment variables and binary commands. To specify that a variable is required, the following can be done:

```
popper require --env VARIABLE_NAME
```

and for commands:

```
popper require --binary command-name
```

In either case, the `popper run` command will check, prior to executing a pipeline, the existence of these and will proceed according to the value given to the `--requirement-level` flag of the `run` subcommand. By default, the execution fails if a dependency is missing.

3.4 Reusing existing pipelines

Many times, when starting an experiment, it is useful to be able to use existing pipelines as scaffolding for the operations we wish to make. The Popperized GitHub organization exists as a curated list of existing Popperized experiments and examples, for the purpose of both learning and scaffolding new projects. Additionally, the CLI includes capabilities easily sift through and import these pipelines.

3.4.1 Searching for existing pipelines

The Popper CLI is capable of searching for premade and template pipelines that you can modify for your own uses. You can use the `popper search` command to find pipelines using keywords. For example, to search for pipelines that use docker you can simply run:

```
$ popper search docker
[##################################] Searching in popperized | 100%

Search results:

> popperized/popper-readthedocs-examples/docker-data-science
```

(continues on next page)

(continued from previous page)

```
> popperized/swc-lesson-pipelines/docker-data-science
```

By default, this command will look inside the Popperized GitHub organization but you can configure it to search the GitHub organization or repository of your choice using the `popper search --add <org-or-repo-name>` command. If you've added more organizations, you may list them with `popper search --ls`, or remove one with `popper search --rm <org-or-repo-name>`

Additionally, when searching for a pipeline, you may choose to include the contents of the readme in your search if you wish by providing the additional `--include` flag to `popper search`.

3.4.2 Importing existing pipelines

Once you have found a pipeline you're interested in importing, you can use `popper add` plus the full pipeline name to add the pipeline to the popperized project:

```
$ popper add popperized/popper-readthedocs-examples/docker-data-science
Downloading pipeline docker-data-science as docker-data-science...
Updating popper configuration...
Pipeline docker-data-science has been added successfully.
```

This will download the contents of the repo to your project tree and register it in your `.popper.yml` configuration file. If you want to add the pipeline inside a different folder, you can also specify that in the `popper add` command:

```
$ popper add popperized/popper-readthedocs-examples/docker-data-science docker-
↪pipeline
Downloading pipeline docker-data-science as docker-pipeline...
Updating popper configuration...
Pipeline docker-pipeline has been added successfully.

$ tree
mypaper
└── pipelines
    └── docker-pipeline
        ├── README.md
        ├── analyze.sh
        ├── docker
        │   ├── Dockerfile
        │   ├── app.py
        │   ├── generate_figures.py
        │   └── requirements.txt
        ├── generate-figures.sh
        ├── results
        │   ├── naive_bayes.png
        │   ├── naive_bayes_results.csv
        │   ├── svm_estimator.png
        │   └── svm_estimator_results.csv
        └── setup.sh
```

You can also tell `popper add` to instead pull the pipeline from another git branch by optionally providing the `--branch <branch-name>` option to the command.

3.5 Continuously validating a pipeline

The `ci` subcommand generates configuration files for multiple CI systems. The syntax of this command is the following:

```
popper ci --service <name>
```

Where `<name>` is the name of CI system (see `popper ci --help` to get a list of supported systems). In the following, we show how to link github with some of the supported CI systems. In order to do so, we first need to create a repository on github and upload our commits:

```
# set the new remote
git remote add origin <your-github-repo-url>

# verify the remote URL
git remote -v

# push changes in your local repository up to github
git push -u origin master
```

3.5.1 TravisCI

For this, we need an account at Travis CI. Assuming our Popperized repository is already on GitHub, we can enable it on TravisCI so that it is continuously validated (see here for a guide). Once the project is registered on Travis, we proceed to generate a `.travis.yml` file:

```
cd my-popper-repo/
popper ci --service travis
```

And commit the file:

```
git add .travis.yml
git commit -m 'Adds TravisCI config file'
```

We then can trigger an execution by pushing to GitHub:

```
git push
```

After this, one go to the TravisCI website to see your pipelines being executed. Every new change committed to a public repository will trigger an execution of your pipelines. To avoid triggering an execution for a commit, include a line with `[skip ci]` as part of the commit message.

> **NOTE**: TravisCI has a limit of 2 hours, after which the test is terminated and failed.

3.5.2 CircleCI

For CircleCI, the procedure is similar to what we do for TravisCI (see above):

1. Sign in to CircleCI using your github account and enable your repository.
2. Generate config files and add them to the repo:

```
cd my-popper-repo/
popper ci --service circle
git add .circleci
git commit -m 'Adds CircleCI config files'
git push
```

3.5.3 GitLab-CI

For GitLab-CI, the procedure is similar to what we do for TravisCI and CircleCI (see above), i.e. generate config files and add them to the repo:

```
cd my-popper-repo/
popper ci --service gitlab
git add .gitlab-ci.yml
git commit -m 'Adds GitLab-CI config file'
git push
```

If CI is enabled on your instance of GitLab, the above should trigger an execution of the pipelines in your repository.

3.5.4 Jenkins

For Jenkins, generating a `Jenkinsfile` is done in a similar way:

```
cd my-popper-repo/
popper ci --service jenkins
git add Jenkinsfile
git commit -m 'Adds Jenkinsfile'
git push
```

Jenkins is a self-hosted service and needs to be properly configured in order to be able to read a github project with a `Jenkinsfile` in it. The easiest way to add a new project is to use the Blue Ocean UI. A step-by-step guide on how to create a new project using the Blue Ocean UI can be found here. In particular, the `New Pipeline from a Single Repository` has to be selected (as opposed to `Auto-discover Pipelines`).

As part of our efforts, we provide a ready-to-use Docker image for Jenkins with all the required dependencies. We also host an instance of this image at http://ci.falsifiable.us and allow anyone to make use of this Jenkins server.

For more on the CI concept, see here. For a detailed explanation on all the CI features, see here. And for In this case, as opposed

3.5.5 Testing Locally

The PopperCLI tool includes a `run` subcommand that can be executed to test locally. This subcommand is the same that is executed by the PopperCI service, so the output of its invocation should be, in most cases, the same as the one obtained when PopperCI executes it. This helps in cases where one is testing locally. To execute test locally:

```
cd my/paper/repo
popper run myexperiment

[##################################] None

status: SUCCESS
```

The status of the execution, as well as the `stdout` and `stderr` output for each stage is stored in the `popper/host` directory inside your pipeline. In addition to the `host` directory, a new directory will be created for every environment you set your pipeline to run on.

```
popper/host
├── popper_status
├── post-run.sh.err
├── post-run.sh.out
├── run.sh.err
├── run.sh.out
├── setup.sh.err
├── setup.sh.out
├── teardown.sh.err
├── teardown.sh.out
├── validate.sh.err
└── validate.sh.out
```

These files are added to the `.gitignore` file so they won't be committed to the git repository when doing `git add`. To quickly remove them, one can clean the working tree:

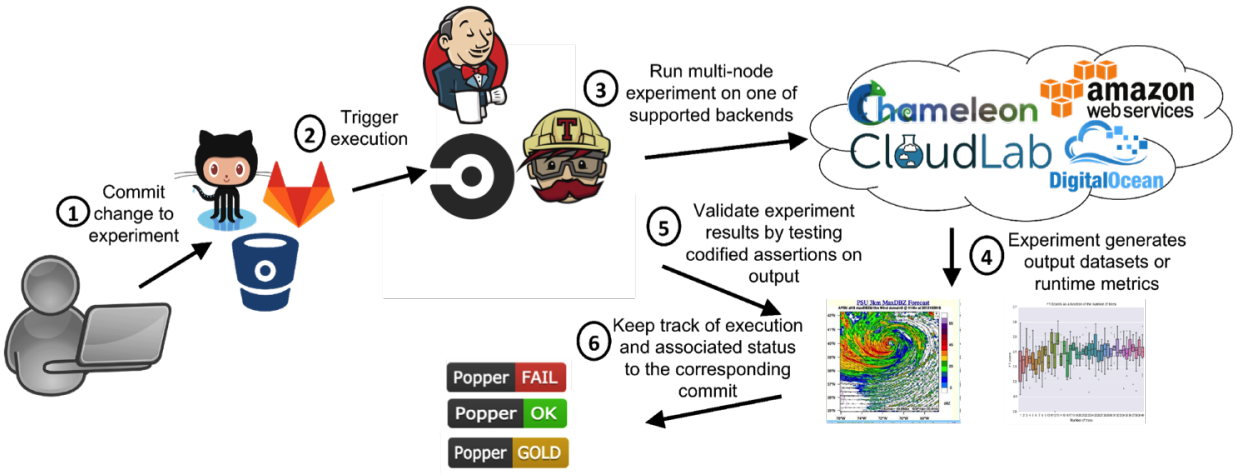
```
# get list of files that would be deleted
# include directories (-d)
# include ignored files (-x)
git clean -dx --dry-run

# remove --dry-run and add --force to actually delete files
git clean -dx --force
```

Execution timeout

By default, `popper run` will set a timeout on the execution of your pipelines. You may modify the timeout using the `--timeout` option, in the form of `popper run --timeout 600s`. You can also disable the timeout altogether by setting `--timeout` to 0.

3.6 Popper Badges

We maintain a badging service that can be used to keep track of the status of a pipeline.



Badges are commonly used to denote the status of a software project with respect to certain aspect, 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 the above figure. If badging is enabled, after the execution of a pipeline, the status of a pipeline is recorded in the badging server (hosted at http://badges.falsifiable.us), which keeps track of the status for every revision of a Popperized project. To retrieve the history for a Popper repo:

```
popper badge --history
```

A link to the badge can be included in the `README.md` page of a project, which is displayed on the web interface of the version control system (GitHub, GitLab, etc.). The CLI tool can generate the link automatically:

```
popper badge --service popper
```

Which prints to `stdout` the text that should be added to the `README.md` file of the project. If the `--inplace` flag is used, the link is added to the `README.md` file.

3.7 Visualizing a pipeline

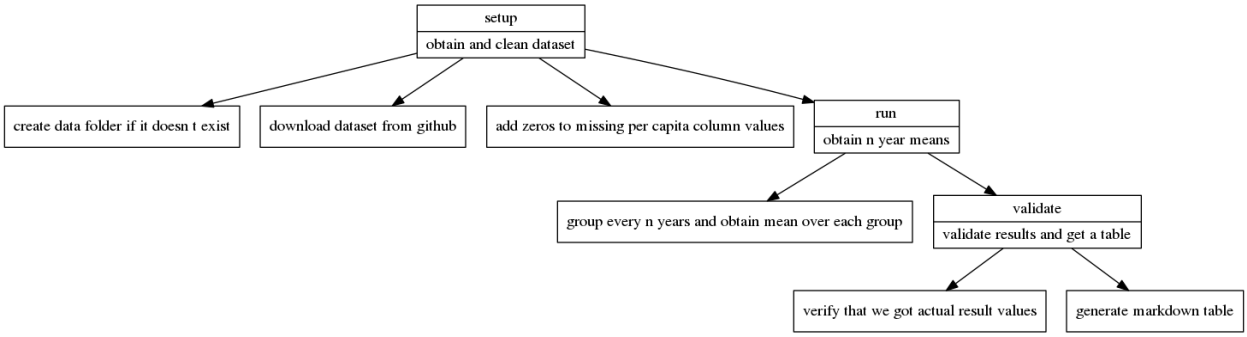
Popper gives a user the ability to visualize the workflow of a pipeline using the `popper workflow pipeline_name` command. The command generates a workflow diagram corresponding to a Popper pipeline, in the .dot format. The string defining the graph is printed to stdout so it can be piped into other tools. For example,to generate a png file, one can make use of the graphviz CLI tools:

```
popper workflow mypipe | dot -T png -o mypipe.png.
```

Suppose you want to visualize the co2-emissions pipeline. Assuming that this pipeline is added to your repository (as explained ini Searching and Importing pipelines, you need to type:

```
popper workflow co2-emissions | dot -T png -o co2_workflow.png
```

This will lead to the generation of the following dot graph:



3.8 Adding metadata to a project

Metadata to a project can be added using the `metadata` command, which adds a `key-value` pair to the repository (to the `.popper.yml` file). For example:

```
popper metadata --add author='Jane Doe'
```

The above adds the metadata item `author` to the project. To retrieve the list of keys:

```
popper metadata
```

And one removes a key by doing:

```
popper metadata --rm author
```

3.9 Archiving and DOI generation

Currently the Popper CLI tool integrates with archival services Zenodo and FigShare for uploading the contents of the repository. This is useful for archiving data that is not part of the Git repository (usually due to it being too big). In addition, these services provide the ability of obtaining a DOI for the archive associated to the project.

3.9.1 Zenodo

The first step is to create an account on Zenodo and generate an API token. Follow these steps (taken from here):

1. Register for a Zenodo account if you don't already have one.
2. Go to your Applications, to create a new token.
3. Select the OAuth scopes you need (you need at least `deposit:write` and `deposit:actions`).

Now add a set of minimal metadata (required by Zenodo, otherwise uploading will fail).

```
popper metadata --add title='<Your Title>'
popper metadata --add author1='<First Last, first.last@gmail.com, Affiliation>'
popper metadata --add abstract='<A short description of the your repo>'
popper metadata --add keywords='<comma, separated, keywords>'
```

Now use the `popper archive` command to perform the archiving.

```
popper archive --service zenodo
```

Enter the token obtained when prompted. Alternatively, this command checks the environment for a `POPPER_ZENODO_API_TOKEN` variable and, if available, uses it to authenticate with the service.

By default, the `archive` command will only upload the snapshot of the project but will not publish it. In order to publish and generate a DOI for the archive, pass the `--publish` flag to the `archive` command:

```
popper archive --service zenodo --publish
```

A URL containing the DOI will be printed to the terminal.

3.9.2 FigShare

Create a personal token using the following steps:

1. Go to https://figshare.com and create a new account.
2. Go to the Applications section of your profile and in the bottom click on `Create Personal Token`.
3. Keep the token safe for use in the next step.

Now add the list of minimal metadata entries (required by FigShare, otherwise uploading will fail).

```
popper metadata --add title='Popper test archive'
popper metadata --add author1='Test Author, testauthor@gmail.com, popper'
popper metadata --add abstract='A short description of the article'
popper metadata --add keywords='comma, separated, keywords'
popper metadata --add categories='1656'
```

After this, the `popper archive` command is used to perform the archiving.

```
popper archive --service figshare
```

Enter the token obtained when prompted. Alternatively, this command checks the environment for a `POPPER_FIGSHARE_API_TOKEN` variable and, if available, uses it to authenticate with the service.

By default, the `archive` command will only upload the snapshot of the project but will not publish it. In order to publish and generate a DOI for the archive, pass the `--publish` flag to the `archive` command:

```
popper archive --service figshare --publish
```

A URL containing the DOI will be printed to the terminal.

3.10 The `popper.yml` configuration file

The `popper` command reads the `.popper.yml` file in the root of a project to figure out how to execute pipelines. While this file can be manually created and modified, the `popper` command makes changes to this file depending on which commands are executed.

The project folder we will use as example looks like the following:

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

That is, it contains three pipelines named `paper`,`data-generation` and `analysis`. The `.popper.yml` for this project looks like:

```
metadata:
  access_right: open
  license: CC-BY-4.0
  publication_type: article
  upload_type: publication

pipelines:
  paper:
    envs:
    - host
    path: paper
    stages:
    - build
  data-generation:
    envs:
```

(continues on next page)

(continued from previous page)

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

popperized:
  - github/popperized
```

At the top-level of the YAML file there are entries named `pipelines`, `metadata` and `popperized`.

3.10.1 Pipelines

The `pipelines` YAML entry specifies the details for all the available pipelines. For each pipeline, there is information about:

- the environment(s) in which the pipeline is be executed.
- the path to that pipeline.
- the various stages that are present in it.

The special `paper` pipeline is generated by executing `popper init paper` and has by default a single stage named `build.sh`.

envs

The `envs` entry in `.popper.yml` specifies the environment in which a pipeline is executed as part of the `popper run` command. By default, a pipeline runs on the host, i.e. the same environment where the `popper` command runs. By leveraging Docker, a pipeline can run on an environment different to the host. The list of available environments can be shown by running:

```
popper env --ls
```

By default, the `host` is the registered environment when running `popper init`. The `--env` flag of the `init` subcommand can be used to specify another environment. For example:

```
popper init mypipe --env=alpine-3.4
```

The above specifies that the pipeline named `mypipe` will be executed inside a docker container using the `falsifiable/popper:alpine-3.4` image.

To add more environment(s):

```
popper env mypipe --add ubuntu-xenial,centos-7.2
```

To deregister an environment:

```
popper env mypipe --rm centos-7.2
```

Arbitrary images can be specified. The only requirement from the point of view of Popper is that they must have `popper` installed in the image. For example:

```
popper env mypipe --add my-docker-repo/image-with-popper-inside
```

stages

The `stages` YAML entry specifies the sequence of stages that are executed by the `popper run` command. By default, the `popper init` command generates scaffold scripts for `setup.sh`, `run.sh`, `post-run.sh`, `validate.sh`, `teardown.sh`. If any of those are not present when the pipeline is executed using `popper run`, they are just skipped (without throwing an error). At least one stage needs to be executed, otherwise `popper run` throws an error.

If arbitrary names are desired for a pipeline, the `--stages` flag of the `popper init` command can be used. For example:

```
popper init arbitrary_stages \
  --stages 'preparation,execution,validation'
```

The above line generates the configuration for the `arbitrary_stages` pipeline showed in the example.

3.10.2 Metadata

The `metadata` YAML entry specifies a set of key-value pairs that describes and gives us information about a project.

By default, a project's metadata will be initialized with the following key-value pairs:

```
$> popper metadata

access_right: open
license: CC-BY-4.0
publication_type: article
upload_type: publication
```

A custom key-value pair can be added using the popper metadata `--add KEY=VALUE` command. For example:

```
popper metadata --add year=2018
```

This adds a metadata entry 'year' to the metadata. The metadata will now look like:

```
access_right: open
license: CC-BY-4.0
publication_type: article
upload_type: publication
year: '2019'
```

To remove the entry 'year' from the `metadata`, the `popper metadata --rm KEY` command can be used as show below:

```
popper metadata --rm year
```

3.10.3 Popperized Repositories and Organizations

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.

CHAPTER 4

CI features

Popper can be used to apply a CI service-agnostic approach to automating the execution of pipelines. The `popper ci` command generates configuration files that a CI service reads in order to execute a pipeline. This section describes this functionality. For more information on how to link your project to a CI service and how to test locally, check here.

4.1 Execution logic

When `popper run` is invoked, each pipeline is executed in alpha-numerical order. When a pipeline runs, each of the stages is executed in the order specified by the `--stages` flag of the `init` command; the `stages` command; or by manually editing the `.popper.yml` file.

The following is the list of high-level tasks that are executed when a pipeline is executed:

1. If specified, environmental requirements are checked. See here for more.
2. For every pipeline, sequentially invoke all the scripts for all the defined stages of the pipeline.
3. After the pipeline finishes, if a `validate.sh` script is defined, its output gets parsed. The `validate.sh` 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>`, for example:

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

4. Keep track of every pipeline and report their status.

There are three possible values for the status of a pipeline: `FAIL`, `SUCCESS` and `GOLD`. When a pipeline does not run to completion (i.e. one of the stages failed), the status of the pipeline is `FAIL`. When the pipeline status' is `GOLD`, the status of all validations is `true`. When all the stages of a pipeline run successfully but one or more validations fail (the status of one or more validations is `false`), the status of a pipeline is `SUCCESS`.

When multiple pipelines are executed, the lowest status among all the pipelines is reported by the `popper run` command, with `FAIL` being the lowest and `GOLD` the highest.

4.1.1 Skipping stages

The `popper run` command has a `--skip` argument that can be used to execute a pipeline in multiple steps. So for example, assuming we have a pipeline with the following scripts: `setup`, `run`, `post-run` and `validate`, we can run:

```
popper run --skip post-run,validate
```

Which runs the first part (setup and execution). Then, later we can either manually check whether the `run` stage is done or automate this task in the `post-run` script. In either way, we would then run:

```
popper run --skip setup,run
```

and the above will just execute the second half of the pipeline.

4.1.2 Specifying which pipelines to run

By default, the `run` subcommand will try to run all the pipelines in a project, unless the current working directory is the folder containing a pipeline. Alternatively, the `run` subcommand takes as argument the name of a pipeline. For example:

```
cd my-popper-repo
popper run
```

The above runs all the pipelines in a repository. While the following runs only the pipeline named `my-pipe`.

```
cd my-popper-repo
popper run my-pipe
```

or alternatively:

```
cd my-popper-repo
cd pipelines/my-pipe
popper run
```

4.1.3 Specifying which pipelines to run via commit messages

The previous subsection applies when `popper run` is invoked directly on a shell. However, when a CI service executes a pipeline, it does so by invoking `popper run` on the CI server, without passing any arguments or flags, and thus we cannot specify which pipelines to execute or skip. To make this more flexible, the `ci` command provides the ability to control which pipelines are executed by looking for special keywords in commit messages.

The `popper:whitelist[<list>]` keyword can be used in a commit message to specify which pipelines to execute. For example:

```
An example commit message

This is a sample commit message that shows how we can request the
execution of a particular pipeline.

popper:whitelist[my-pipe]
```

The above commit message specifies that the pipeline `my-pipe` is to be executed and any other pipeline will be skipped. A comma-separated list of pipeline names can be given in order to request the execution of more then one pipeline. A skip list is also supported with the `popper:skip[<list>]` keyword.

4.2 Execution environments

By default, a pipeline runs on the same environment where the `popper` command is being executed. In certain cases, it is useful to run a pipeline on a different environment. Popper leverages Docker to accomplish this. For more on how to define and remove environments, see here. For each environment, an output log folder is created. For example, the following pipeline 2-stage pipeline:

```
pipelines:
  my-pipe:
    path: pipelines/my-pipe
    envs:
    - host
    - debian-9
    stages:
    - one
    - two
```

Results in the logs folder in `pipelines/my-pipe/popper` to have the following structure:

```
$ tree pipelines/my-pipe/popper
pipelines/my-pipe/popper
├── debian-9
│   ├── popper_status
│   ├── one.sh.err
│   ├── one.sh.out
│   ├── two.sh.err
│   └── two.sh.out
└── host
    ├── popper_status
    ├── one.sh.err
    ├── one.sh.err
    ├── two.sh.err
    └── two.sh.out
```

That is, there is one folder for each distinct environment. The status of the pipeline reported by `popper run` is the lowest status from all the executions, with `FAIL` being the lowest and `GOLD` the highest.

4.2.1 Specifying arguments for `docker run`

A Docker environment image is instantiated with the following command:

```
docker run --rm \
  --volume /path/to/project:/path/to/project \
  --workdir /path/to/project/path/to/pipeline \
  <popper-docker-image> \
    popper run <flags> <arg>
```

Where `<popper-docker-image>` is an image with Popper available inside of it. The project folder is shared with the container and the pipeline folder is the working directory. To specify other flags to the `docker run` command, the `--argument` flag of the `env` command can be used. For usage, type `popper env --help`.

4.3 Parametrizing pipelines

A pipeline can be parametrized so that it can be executed multiple times, taking a distinct set of parameters each time. Parameters are specified with the `parameters` subcommand and are given in the form of environment variables. For example, if a pipeline takes parameters `par1` and `par2`, the following can specify these:

```
popper parameters my-pipe --add par1=val1 --add par2=val2
```

This will cause the `.popper.yml` to look like the following:

```
pipelines:
  my-pipe:
    envs:
    - host
    stages:
    - one
    - two
    parameters:
    - { par1: val1, par2: val2 }
```

Each new set of parameters results in a dictionary of key-value pairs, where each item in the dictionary is an environment variable. A subsequent set of parameters can be added:

```
popper parameters my-pipe --add par1=val3 --add par2=val4
```

Which will result in the following:

```
pipelines:
  my-pipe:
    envs:
    - host
    stages:
    - one
    - two
    parameters:
    - par1: val1
      par2: val2
    - par1: val3
      par2: val4
```

The above results in executing this same pipeline two times, defining environment variables `par1` and `par2`, with these 2 sets of values, every time it runs. We refer to each execution in a parametrized pipeline as a *Job*. In this example we will have 2 jobs every time the pipeline runs.

When a parametrized pipeline is executed, a subfolder for each job is created. For example, the 2-job pipeline specified above results in the following folder structure:

```
$ tree pipelines/your-popper-pipeline/popper
pipelines/your-popper-pipeline/popper
└── host
    ├── 0
    │   ├── popper_status
    │   ├── one.sh.err
    │   ├── one.sh.out
    │   ├── two.sh.err
    │   └── two.sh.out
    └── 1
```

(continues on next page)

(continued from previous page)

```
        ├── popper_status
        ├── one.sh.err
        ├── one.sh.out
        ├── two.sh.err
        └── two.sh.out
```

Subfolders are numbered (starting from 0), where each ID corresponds to the position of the set of parameters in the `parameters` list. So the run 0 above corresponds to `par1=val1,par2=val2` and 1 to `par1=val3, par2=val4`.

4.4 Matrix Executions

Combining *Execution environments* with *Parameters* results in a bi-dimensional matrix of jobs. For example, the following pipeline:

```
pipelines:
  my-pipe:
    envs:
    - host
    - debian-9
    stages:
    - one
    - two
    parameters:
    - par1: val1
      par2: val2
    - par1: val3
      par2: val4
```

Results in the following folder structure for the `popper/` logs folder:

```
$ tree pipelines/your-popper-pipeline/popper
pipelines/your-popper-pipeline/popper
├── debian-9
│   ├── 0
│   │   ├── popper_status
│   │   ├── one.sh.err
│   │   ├── one.sh.out
│   │   ├── two.sh.err
│   │   └── two.sh.out
│   └── 1
│       ├── popper_status
│       ├── one.sh.err
│       ├── one.sh.out
│       ├── two.sh.err
│       └── two.sh.out
└── host
    ├── 0
    │   ├── popper_status
    │   ├── one.sh.err
    │   ├── one.sh.out
    │   ├── two.sh.err
    │   └── two.sh.out
    └── 1
```

(continues on next page)

(continued from previous page)

```
        ├── popper_status
        ├── one.sh.err
        ├── one.sh.out
        ├── two.sh.err
        └── two.sh.out
```

4.5 Popper Badges

See here.

CHAPTER 5

Examples

In this section we present a examples of Popper pipelines. All these are available on github and can be added to a local repo by doing:

```
popper add popperized/<repo>/<pipeline>
```

Where `<repo>` is the name of the repository where a pipeline is contained, and `<pipeline>` is the name of the pipeline.

5.1 Pipeline portability

- Using Virtualenv (Python) (and also here).
- Using Packrat (R).
- Using Spack.
- Using Docker.
- Using Vagrant.

5.2 Dataset Management

- Using Datapackages.
- Using data.world.

5.3 Infrastructure automation

- On CloudLab using geni-lib.

- On Chameleon using enos.
- On Google Compute Platform using terraform.

5.4 Domain-specific pipelines

- Atmospheric science.
- Applied math.
- Machine learning.
- Linux kernel development.
- Relational databases.
- Genomics.
- High Performance Computing.
- Computational Neuroscience.

Coming soon:

- Distributed file systems
- High energy physics

5.5 Results Validation

- Statistical validations.
- Bitwise image comparison.

5.6 Pipeline Parametrization

- Using environment variables
- Using baseliner.

5.7 Provenance and automatic dependency resolution

- Using Sumatra
- Sci-Unit

Coming soon:

- ReproZip

CHAPTER 6

Other Resources

6.1 Automated Artifact Evaluation

A growing number of Computer Science conferences and journals incorporate an artifact evaluation process in which authors of articles submit artifact descriptions that are tested by a committee, in order to verify that experiments presented in a paper can be re-executed by others. In short, an artifact description is a 2-3 page narrative on how to replicate results, including steps that detail how to install software and how to re-execute experiments and analysis contained in a paper.

An alternative to the manual creation and verification of an Artifact Description (AD) is to use a continuous integration (CI) service such as GitLab-CI or Jenkins. Authors can make use of a CI service to automate the experimentation pipelines associated to a paper. By doing this, the URL pointing to the project on the CI server that holds execution logs, as well as the repository containing all the automation scripts, can serve as the AD. In other words, the repository containing the code for experimentation pipelines, and the associated CI project, serve both as a "self-verifiable AD". Thus, instead of requiring manually created ADs, conferences and journals can request that authors submit a link to a code repository (Github, Gitlab, etc.) where automation scripts reside, along with a link to the CI server that executes the pipelines.

While automating the execution of a pipeline can be done in many ways, in order for this approach to serve as an alternative to ADs, there are five high-level tasks that pipelines must carry out in every execution:

- Code and data dependencies. Code must reside on a version control system (e.g. github, gitlab, etc.). If datasets are used, then they should reside in a dataset management system (datapackage, gitlfs, dataverse, etc.). The experimentation pipelines must obtain the code/data from these services on every execution.
- Setup. The pipeline should build and deploy the code under test. For example, if a pipeline is using containers or VMs to package their code, the pipeline should build the container/VM images prior to executing them. The goal of this is to verify that all the code and 3rd party dependencies are available at the time a pipeline runs, as well as the instructions on how to build the software.
- Resource allocation. If a pipeline requires a cluster or custom hardware to reproduce results, resource allocation must be done as part of the execution of the pipeline. This allocation can be static or dynamic. For example, if an experiment runs on custom hardware, the pipeline can statically allocate (i.e. hardcode IP/hostnames) the machines where the code under study runs (e.g. GPU/FPGA nodes). Alternatively, a pipeline can dynamically

allocate nodes (using infrastructure automation tools) on CloudLab, Chameleon, Grid5k, SLURM, Terraform (EC2, GCE, etc.), etc.

- Environment capture. Capture information about the runtime environment. For example, hardware description, OS, system packages (i.e. software installed by system administrators), remote services (e.g. a scheduler). Many open-source tools can aid in aggregating this information such as SOSReport or facter.
- Validation. Scripts must verify that the output corroborates the claims made on the article. For example, the pipeline might check that the throughput of a system is within an expected confidence interval (e.g. defined with respect to a baseline obtained at runtime).

A list of example Popper pipelines meeting the above criteria:

- BLIS paper. We took an appendix and turned it into executable pipeline.
- HPC Proxy App. Runs LULESH linked against MPIp to capture runtime MPI perf metrics.
- Linux kernel development. Uses a VM to compile, test and deploy Linux.
- Relational database performance. Runs pgbench to compare two versions of postgres.

More examples are listed here.

> **NOTE**: A pipeline can be implemented by any means and does **not** need to be implemented using the Popper CLI. While the examples we link above are of Popper pipelines, this subsection has the intention to apply, in general, to any type of automated approach. Our intention is to define, in written form, the criteria for Automated Artifact Evaluation.

6.1.1 CI Infrastructure for Automated Artifact Evaluation

We have an instance of Jenkins running at http://ci.falsifiable.us, maintained by members of the Systems Research Lab (SRL) at UC Santa Cruz. Detailed instructions on how to create an account on this service and how to use it is available here (also includes instructions on how to self-host it). This service allows researchers and students to automate the execution and validation of experimentation pipelines without happing to deploy infrastructure of their own.

6.2 Self-paced Tutorial

A hands-on, self-paced tutorial is available here.

CHAPTER 7

FAQ

7.1 How can we deal with large datasets? For example I have to work on large data of hundreds GB, how would this be integrated into Popper?

For datasets that are large enough that they cannot be managed by Git, solutions such as a PFS, GitLFS, Datapackages, ckan, among others exist. These tools and services allow users to manage large datasets and version-control them. From the point of view of Popper, this is just another tool that will get invoked as part of the execution of a pipeline. As part of our documentation, we have examples on how to use datapackages, and another on how to use data.world.

7.2 How can Popper capture more complex workflows? For example, automatically restarting failed tasks?

A Popper pipeline is a simple sequence of bash scripts. Popper is not a replacement for scientific workflow engines, instead, its goal is to capture the highest-most workflow: the human interaction with a terminal. For more on this, please take a look at the Popper vs. other software section of our documentation.

7.3 Can I follow Popper in computational science research, as opposed to computer science?

Yes, the goal for Popper is to make it a domain-agnostic experimentation protocol. Examples of how to follow Popper on distinct domains: atmospheric science, computational neuroscience, genomics and applied math.

7.4 How to apply the Popper protocol for applications that take large quantities of computer time?

The `popper run` command has a `--skip` argument that can be used to execute a pipeline in multiple steps. See here for more information.

Another practice we have been following is to have a specific set of parameters for the pipeline with the goal of running a smaller scale simulation/analysis. The idea is to use this when running on a CI service such as Travis in order to test the entire pipeline in a relatively short amount of time (Travis times out jobs after 3 hours). So this ends up looking something like this, i.e. a conditional in a stage that, depending on the environment (in this case a `CI` variable defined), the parametrization and setup is different, but the rest of the pipeline runs in the same fashion. While this approach doesn't really executes the actual original simulation, at least it lets us test the integrity of the scripts.

CHAPTER 8

Contributing

8.1 Code of Conduct

Anyone is welcome to contribute to Popper! To get started, take a look at our contributing guidelines, then dive in with our list of good first issues and open projects.

Popper adheres to the code of conduct posted in this repository. By participating or contributing to Popper, you're expected to uphold this code. If you encounter unacceptable behavior, please immediately email us.

8.2 Contributing CLI features

To contribute new CLI features:

1. Add a new issue describing the feature.
2. Fork the official repo and implement the issue on a new branch.
3. Add tests for the new feature. We test the `popper` CLI command using Popper itself. The Popper pipeline for testing the `popper` command is available here.
4. Open a pull request against the `master` branch.

8.3 Contributing example pipelines

We invite anyone to implement (and document) Popper pipelines demonstrating the use of a DevOps tool, or how to apply Popper in a particular domain. Implementing a new example is done in two parts.

8.3.1 Implement the pipeline

A popper pipeline is implemented by following the convention. See the Concepts and Examples section for more.

Once a pipeline has been implemented, it needs to be uploaded to github, gitlab or any other repo publicly available. We use the organization https://github.com/popperized to host examples developed by the Popper team and collaborators. Pipelines on this organization are available by default to the `popper search` command, so users can add it easily to their repos (using `popper add`). To add a repository containing one or more pipelines to this organization, please first create the repository on GitHub under an organization you own and then do one of the following:

- Transfer ownership of the repo to the `popperized` organization.
- Open an issue requesting the repository to be forked or mirrored. **NOTE**: forks and mirrors need to be updated manually in order to reflect changes done on the base/upstream repository.

8.3.2 Document the pipeline

We encourage contributors to document pipelines by adding them to our list of examples. To add new documentation:

1. Fork the official repo.
2. Add a new item on the `docs/sections/examples.md` file.
3. Open pull request against the `master` branch.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search