# Sphinx with Markdown Documentation

*Release 0.1.0*

**Marijn van der Zee**

**Jun 12, 2018**

# Contents

**1 Getting Started** **3**
- 1.1 New pipeline . . . 3
- 1.2 Popper Run . . . 4
- 1.3 Adding Project to GitHub . . . 4
- 1.4 Adding Project to Travis . . . 4
- 1.5 Integrating with GitHub . . . 5
- 1.6 Learn More . . . 5

**2 Introduction to Popper Pipelines** **7**
- 2.1 Popper Pipelines . . . 7
- 2.2 Repository Structure . . . 9
- 2.3 Pipeline Folder Structure . . . 9

**3 The `popper.yml` configuration file** **11**
- 3.1 `pipelines` . . . 12
- 3.2 `metadata` . . . 13
- 3.3 `popperized` . . . 13

**4 Popper and CI systems** **15**
- 4.1 CI System Configuration . . . 15
- 4.2 CI Functionality . . . 16
- 4.3 Testing Locally . . . 17
- 4.4 Popper Badges . . . 17

**5 Popper vs. Other Software** **19**
- 5.1 Scientific Workflow Engines . . . 19
- 5.2 Virtualenv, Conda, Packrat, etc. . . . 20
- 5.3 CI systems . . . 20
- 5.4 Reprozip / Sciunit . . . 20

**6 Review Workflow** **21**

**7 Indices and tables** **23**

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
```

> **NOTE**: this exercise was written using 0.5

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
```

To run a pipeline named myexp:

```
popper run myexp
```

> **NOTE:** By default, `popper run` runs all commands directly on the host. We recommend running an isolated environment. In order to do this, one can create a pipeline using the `--env` flag of the `popper init` command. For example, `popper init <pipeline> --env=alpine-3.4` runs a command inside an `alpine-3.4` container.

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 and upload our commits.

## 1.4 Adding Project to Travis

For this, we need an account at Travis CI. Once we have one, we activate the project so it is continuously validated.

Generate `.travis.yml` file:

```
popper ci 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.

## 1.5 Integrating with GitHub

Some of the popper sub-commands (e.g. :- popper search) make use of the GitHub API. Since GitHub only allows up to 60 unauthenticated requests per hour on its API, some of these sub-commands will fail to give appropriate results on heavy usage.

To resolve this, we need to :

- Create a GitHub personal access token, as shown here.
- Copy the token and set it as an environment variable with the name `POPPER_GITHUB_API_TOKEN` in our computer.

This will allow the popper command to use our access token to make authenticated requests.

## 1.6 Learn More

A more detailed description of Popper is explained in the next section.

A step-by-step guide describes how to "Popperize" a repository. Additionally, the following is a list of examples on how to bootstrap a Popper project (repository) in specific domains:

- Data Science
- High Performance Computing (HPC)
- Mathematical Sciences

A list of articles describing the Popper protocol, as well as other Popperized papers that have been submitted for publication can be found here.

CHAPTER 2

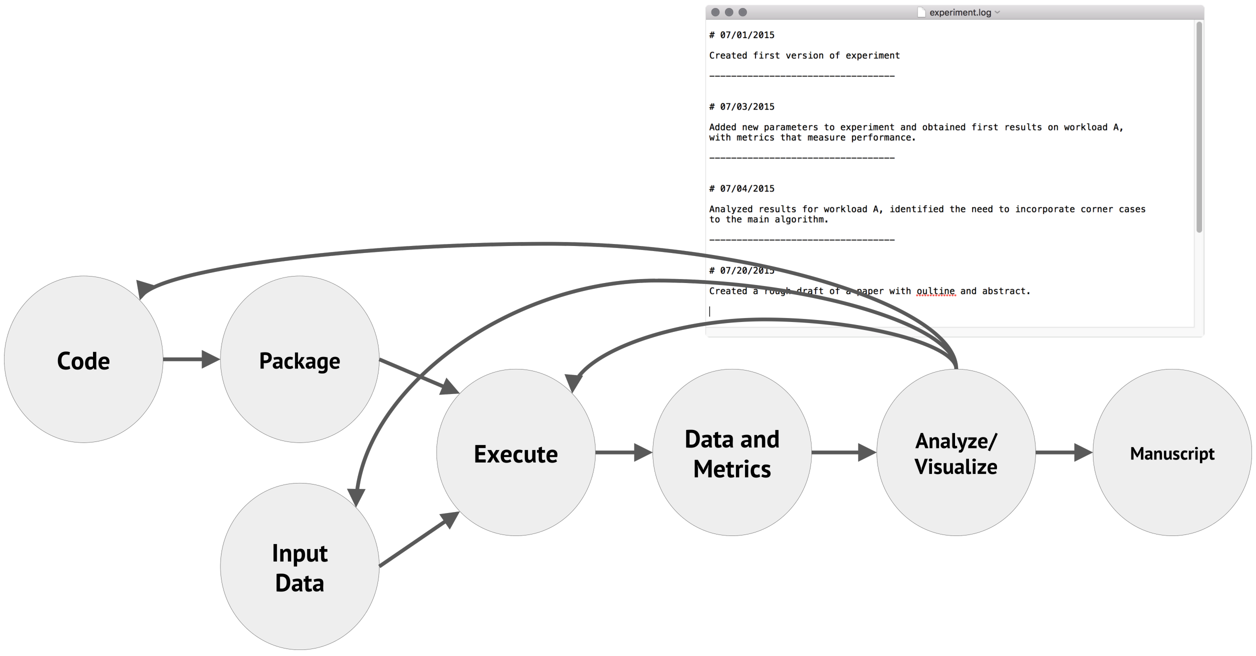
# Introduction to Popper Pipelines

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.

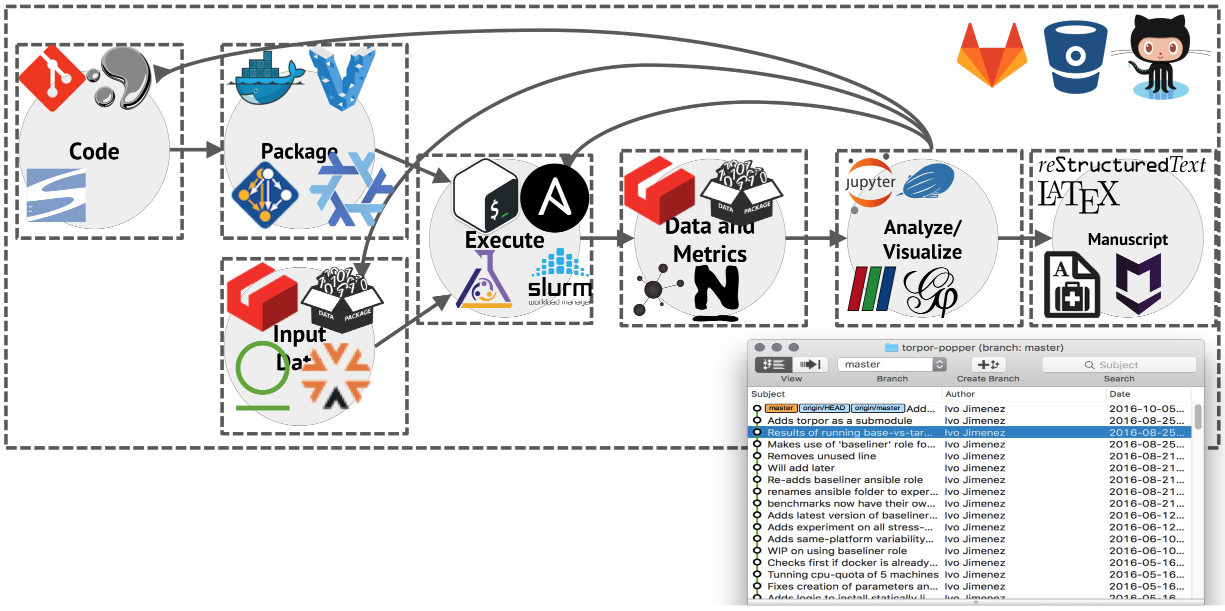
## 2.1 Popper Pipelines

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.).



```
# 07/01/2015

Created first version of experiment

-----------------------------------


# 07/03/2015

Added new parameters to experiment and obtained first results on workload A,
with metrics that measure performance.

-----------------------------------


# 07/04/2015

Analyzed results for workload A, identified the need to incorporate corner cases
to the main algorithm.

-----------------------------------


# 07/20/2015

Created a rough draft of a paper with oultine and abstract.
```

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).



| Subject | Author | Date |
|---|---|---|
| master origin/HEAD origin/master Add... | Ivo Jimenez | 2016-10-05... |
| Adds torpor as a submodule | Ivo Jimenez | 2016-08-25... |
| Results of running base-vs-tar... | Ivo Jimenez | 2016-08-25... |
| Makes use of 'baseliner' role fo... | Ivo Jimenez | 2016-08-25... |
| Removes unused line | Ivo Jimenez | 2016-08-21... |
| Will add later | Ivo Jimenez | 2016-08-21... |
| Re-adds baseliner ansible role | Ivo Jimenez | 2016-08-21... |
| renames ansible folder to exper... | Ivo Jimenez | 2016-08-21... |
| benchmarks now have their ow... | Ivo Jimenez | 2016-08-21... |
| Adds latest version of baseliner... | Ivo Jimenez | 2016-06-12... |
| Adds experiment on all stress-... | Ivo Jimenez | 2016-06-12... |
| Adds same-platform variability... | Ivo Jimenez | 2016-06-10... |
| WIP on using baseliner role | Ivo Jimenez | 2016-06-10... |
| Checks first if docker is already... | Ivo Jimenez | 2016-05-16... |
| Tunning cpu-quota of 5 machines | Ivo Jimenez | 2016-05-16... |
| Fixes creation of parameters an... | Ivo Jimenez | 2016-05-16... |
| Adds logic to install statically li... | Ivo Jimenez | 2016-05-16 |

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 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
```

## 2.3 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.

CHAPTER 3

# 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 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
```

(continues on next page)

(continued from previous page)

```
    - 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`.

## 3.1 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`.

### 3.1.1 envs

The `envs` entry in `.popper.yml` specifies the environment that a pipeline is used when the pipeline is executed as part of the `popper run` command. The available environments are:

- `host`. The experiment is executed directly on the host.
- `alpine-3.4`, `ubuntu-16.04` and `centos-7.2`. For each of these, `popper check` is executed within a docker container whose base image is the given Linux distribution name. The container has `docker` available inside it so other containers can be executed from within the `popper check` container.

The `popper init` command can be used to initialize a pipeline. By default, the `host` is registered when using `popper init`. The `--env` flag of `popper init` 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 `alpine-3.4` popper check image.

To add more environment(s):

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

To remove enviroment(s):

```
popper env myexp --rm centos-7.2
```

### 3.1.2 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.

## 3.2 metadata

The `metadata` YAML entry specifies the set of data that gives information about the user's project. It can be added using the `popper metadata --add command` For example :

```
popper metadata --add authors='Dennis Ritchie'
```

This adds a metadata entry 'authors' to the the project metadata.

To view the `metadata` of a repository type:

```
popper metadata
```

To remove the entry 'authors' from the `metadata`:

```
popper metadata --rm authors
```

## 3.3 popperized

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

# Popper and CI systems

By following a convention for structuring the files of a project, an experimentation pipeline execution and validation can be automated without the need for manual intervention. In addition to this, the status of a pipeline (integrity over time) can be tracked by a continuous integration (CI) service. In this section we describe how Popper integrates with some existing CI services.

## 4.1 CI System Configuration

The PopperCLI tool includes a `ci` subcommand that can be executed to generate configuration files for multiple CI systems. The syntax of this command is the following:

```
popper ci <system-name>
```

Where `<system-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.

### 4.1.1 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 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.

### 4.1.2 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 circleci
git add .circleci
git commit -m 'Adds CircleCI config file'
git push
```

### 4.1.3 Jenkins

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

```
cd my-popper-repo/
popper ci 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. See here for an example of how to use it. We also host an instance of this image at http://ci.falsifiable.us and can provide accounts for users to make use of this Jenkins server (for an account, send an email to ivo@cs.ucsc.edu).

## 4.2 CI Functionality

The following is the list of steps that are verified when validating an pipeline:

1. For every pipeline, trigger an execution by sequentially invoking all the scripts for all the defined stages of the pipeline.
2. After the pipeline finishes, if a `validate.sh` script is defined, parse its output.
3. Keep track of every pipeline and report their status.

There are three possible statuses for every pipeline: `FAIL`, `PASS` and `GOLD`. There are two possible values for the status of a validation, `FAIL` or `PASS`. When the pipeline status is `FAIL`, this list of validations is empty since the pipeline execution has failed and validations are not able to execute at all. When the pipeline status' is `GOLD`, the status of all validations is `PASS`. When the pipeline runs correctly but one or more validations fail (pipeline's status is `PASS`), the status of one or more validations is `FAIL`.

## 4.3 Testing Locally

The PopperCLI tool includes a `check` 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 check myexperiment

Popper check started
Running stage setup.sh ....
Running stage run.sh ................
Running stage validate.sh .
Running stage teardown.sh ..
Popper check finished: SUCCESS
```

The status of the execution is stored in the `popper_status` file, while `stdout` and `stderr` output for each stage is written to the `popper_logs` folder.

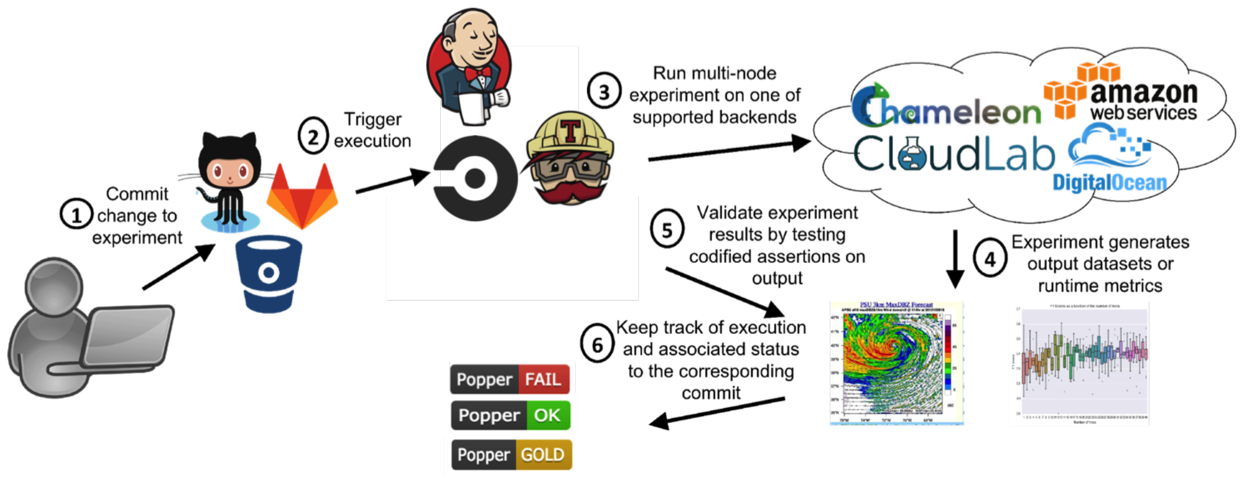
```
tree popper_logs
popper_logs/
├── run.sh.out
├── run.sh.err
├── setup.sh.out
├── setup.sh.err
├── teardown.sh.out
├── teardown.sh.err
├── validate.sh.out
└── validate.sh.err
```

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
```

## 4.4 Popper Badges

We maintain a badging service that can be used to keep track of the status of a pipeline. In order to enable this, the `--enable-badging` flag has to be passed to the `popper ci` subcommand.


1 Commit change to experiment
2 Trigger execution
3 Run multi-node experiment on one of supported backends
Chameleon
amazon web services
CloudLab
DigitalOcean
4 Experiment generates output datasets or runtime metrics
5 Validate experiment results by testing codified assertions on output
6 Keep track of execution and associated status to the corresponding commit
Popper FAIL
Popper OK
Popper GOLD

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, which keeps track of statuses for every revision of ever pipeline.

Users can include a link to the badge in the `README` page of a pipeline, which can be displayed on the web interface of the version control system (GitHub in this case). The CLI tool can generate links for pipelines:

```
popper badge <exp>
```

Which prints to `stdout` the text that should be added to the `README` file of the pipeline.

CHAPTER 5

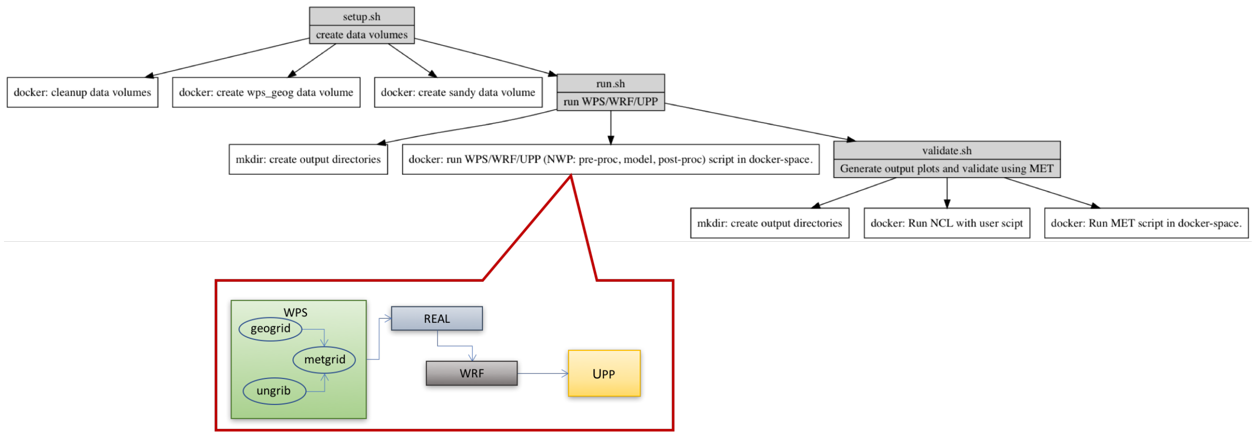
# Popper vs. Other Software

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

## 5.1 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:


setup.sh
create data volumes
docker: cleanup data volumes
docker: create wps_geog data volume
docker: create sandy data volume
run.sh
run WPS/WRF/UPP
mkdir: create output directories
docker: run WPS/WRF/UPP (NWP: pre-proc, model, post-proc) script in docker-space.
validate.sh
Generate output plots and validate using MET
mkdir: create output directories
docker: Run NCL with user scipt
docker: Run MET script in docker-space.
WPS
geogrid
metgrid
ungrib
REAL
WRF
UPP

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.

## 5.2 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 example `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.

## 5.3 CI systems

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository frequently with the purpose of catching errors as early as possible. The pipelines associated with an article can also benefit from CI. 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 verified on every change to a pipeline repository.

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 tests that can run on multiple CI services with minimal effort. The `popper ci` command generates configuration files that existing CI systems read in order to execute a popper pipeline. Additionally, with most of existing tools and services, users don't have a way of easily checking the integrity of a pipeline locally, whereas Popper can be used easily 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 FAILURE) but we can also verify that the output is the one expected by the original implementers.

## 5.4 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 Reprozip/Sciunit, as well as to create and execute Reprozip packages and Sciunits. However, a Popper pipeline is already self-contained and can be made portable by explicitly using language (e.g. virtualenv), OS-level (e.g. Singularity) or hardware (e.g. Virtualbox) virtualization tools. In these cases, using Reprozip or Sciunit would be redundant, since they make use of Docker or Vagrant "underneath the covers" in order to provide portable experiment packages/units.

# CHAPTER 6

# Review Workflow

The following provides a list of steps with the goal of reviewing a popper pipeline that someone else has created:

1. The `popper workflow` command generates a graph that gives a high-level view of what the pipeline does.
2. Inspect the content of each of the scripts from the pipeline.
3. Test that the pipeline works on your machine by running `popper run`.
4. Check that the git repo was archived (snapshotted) to zenodo or figshare by running `popper zenodo` or `popper figshare`.

# CHAPTER 7

# Indices and tables

- genindex
- modindex
- search