Popper Documentation

Release 2.x

Ivo Jimenez

Jun 14, 2020

Contents

Popper is a container-native workflow engine for seamlessly running workflows locally and on CI services.

CHAPTER 1

Getting Started

Popper is a container-native workflow execution engine. A container-native workflow is one where all steps contained in it are executed in containers. Before going through this guide, you need to have the Docker engine installed on your machine (see installations instructions here), as well as a Python 3.6+ installation capable of adding packages via Pip or Virtualenv.

1.1 Installation

We provide a `pip` package for Popper. To install simply run:

```
pip install popper
```

Depending on your Python distribution or specific environment configuration, using Pip might not be possible (e.g. you need administrator privileges) or using `pip` directly might incorrectly install Popper. We **highly recommend** to install Popper in a Python virtual environment using virtualenv. The following installation instructions assume that `virtualenv` is installed in your environment (see here for more). Once `virtualenv` is available in your machine, we proceed to create a folder where we will place the Popper virtual environment:

```
# create a folder for storing virtual environments
mkdir $HOME/virtualenvs
```

We then create a `virtualenv` for Popper. This will depend on the method with which `virtualenv` was installed:

```
# 1) if virtualenv was installed via package, e.g.:
# - apt install virtualenv (debian/ubuntu)
# - yum install virtualenv (centos/redhat)
# - conda install virtualenv (conda)
# - pip install virtualenv (pip)
virtualenv $HOME/virtualenvs/popper

# OR
#
```

(continues on next page)

(continued from previous page)

```
# 2) if virtualenv installed via Python 3.6+ module
python -m venv $HOME/virtualenvs/popper
```

> **NOTE**: in the case of `conda`, we recommend the creation of a new environment before `virtualenv` is installed in order to avoid issues with packages that might have been installed previously.

We then load the environment we just created above:

```
source $HOME/virtualenvs/popper/bin/activate
```

Finally, we install Popper in this environment using `pip`:

```
pip install popper
```

To test all is working as it should, we can show the version we installed:

```
popper version
```

And to get a list of available commands:

```
popper --help
```

> **NOTE**: given that we are using `virtualenv`, once the shell session ends (when we close the terminal window or tab), the environment gets unloaded and newer sessions (new window or tab) will not have the `popper` command available in the `PATH` variable. In order to have the environment loaded again we need to execute the `source` command (see above). In the case of `conda` we need to load the Conda environment (`conda activate` command).

1.2 Create a Git repository

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

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

> **NOTE**: if you run on MacOS, make sure the `myproject/` folder is in a folder that is shared with the Docker engine. By default, Docker For Mac shares the `/Users` folder, so putting the `myproject/` folder in any subfolder of `/Users/<USERNAME>/` should suffice. Otherwise, if you want to put it on an folder other than `/Users`, you will need to modify the Docker For Mac settings so that this other folder is also shared with the underlying Linux VM.

1.3 Create a workflow

We create a small, pre-defined workflow by running:

```
popper scaffold
```

The above generates an example workflow that you can use as the starting point of your project. This minimal example illustrates two distinct ways in which a `Dockerfile` image can be used in a workflow (by pulling an image from a registry, or by referencing one stored in a public repository). To show the content of the workflow:

```
cat wf.yml
```

For each step in the workflow, an image is created (or pulled) and a container is instantiated. For a more detailed description of how Popper processes a workflow, take a look at the "Workflow Language and Runtime" section. To learn more on how to modify this workflow in order to fit your needs, take a look at this tutorial or take a look at some examples.

Before we go ahead and test this workflow, we first commit the files to the Git repository:

```
git add .
git commit -m 'Adding example workflow.'
```

1.4 Run your workflow

To execute the workflow you just created:

```
popper run -f wf.yml
```

You should see the output printed to the terminal that informs of the three main tasks that Popper executes for each step in a workflow: build (or pull) a container image, instantiate a container, and execute the step by invoking the specified command within the container.

> **TIP:** Type `popper run --help` to learn more about other options available and a high-level description of what this command does.

Since this workflow consists of two steps, there were two corresponding containers that were executed by the underlying container engine, which is Docker in this case. We can verify this by asking Docker to show the list of existing containers:

```
docker ps -a
```

You should see the two containers from the example workflow being listed.

To obtain more detailed information of what this command does, you can pass the `--help` flag to it:

```
popper run --help
```

> **NOTE**: All Popper subcommands allow you to pass `--help` flag to it to get more information about what the command does.

1.5 Link to GitHub repository

Create a repository on Github. Once your Github repository has been created, register it as a remote repository on your local repository:

```
git remote add origin git@github.com:<user>/<repo>
```

where `<user>` is your username and `<repo>` is the name of the repository you have created. Then, push your local commits:

```
git push -u origin master
```

1.6 Continuously Run Your Workflow on 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 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 the TravisCI website to see your experiments being executed.

1.7 Next Steps

For a detailed description of how Popper processes workflows, take a look at the "Workflow Language and Runtime" section. To learn more on how to modify workflows to fit your needs, take a look at this tutorial or at some examples.

CHAPTER 2

CLI feautures

2.1 New workflow 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 a workflow

```
popper scaffold
```

Show what this did (a `wf.yml` should have been created):

```
ls -l
```

Commit the "empty" pipeline:

```
git add .
git commit -m 'adding my first workflow'
```

2.2 Executing a workflow

To run the workflow:

```
popper run -f wf.yml
```

where `wf.yml` is a file containing a workflow.

2.3 Executing a step interactively

To open a shell in a step defined in the workflow:

```
popper sh step
```

Where `step` is the name of a step contained in the workflow. For given the following workflow:

```
steps:
- id: mystep
  uses: docker://ubuntu:18.04
  runs: ["ls", "-l"]
  env:
    MYENVVAR: "foo"
```

if we want to open a shell that puts us inside the `mystep` above, we run:

```
popper sh mystep
```

And this opens an interactive shell inside that step, where the environment variable `MYENVVAR` is available. Note that the `runs` and `args` attributes are overridden by Popper. By default, `/bin/bash` is used to start the shell, but this can be modified with the `--entrypoint` flag.

2.4 Customizing container engine behavior

By default, Popper instantiates containers in the underlying engine by using basic configuration options. When these options are not suitable to your needs, you can modify or extend them by providing engine-specific options. These options allow you to specify fine-grained capabilities, bind-mounting additional folders, etc. In order to do this, you can provide a configuration file to modify the underlying container engine configuration used to spawn containers. This file is a python script that defines an `ENGINE` dictionary with custom options and is passed to the `popper run` command via the `--conf` flag.

For example, to make Popper spawn Docker containers in privileged mode, we can write the following options:

```
ENGINE = {
  'privileged': True
}
```

Assuming the above is stored in a file called `settings.py`, we pass it to Popper by running:

```
popper run -f wf.yml --conf settings.py
```

NOTE:

1. Currently, the `--conf` option is only supported for the `docker` engine.

2. The `settings.py` file must contain a `dict` type variable with the name `ENGINE` as shown above.

2.5 Continuously validating a workflow

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

```
popper ci --file wf.yml <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
```

2.5.1 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 --file wf.yml 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.

2.5.2 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 --file wf.yml circle
git add .circleci
git commit -m 'Adds CircleCI config files'
git push
```

2.5.3 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 --file wf.yml 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.

2.5.4 Jenkins

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

```
cd my-popper-repo/
popper ci --file wf.yml 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`).

2.6 Visualizing workflows

While `.workflow` files are relatively simple to read, it is nice to have a way of quickly visualizing the steps contained in a workflow. Popper provides the option of generating a graph for a workflow. To generate a graph for a pipeline, execute the following:

```
popper dot -f wf.yml
```

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

```
popper dot -f wf.yml | dot -T png -o wf.png
```

The above generates a `wf.png` file depicting the workflow. Alternatively you can use the http://www.webgraphviz.com/ website to generate a graph by copy-pasting the output of the `popper dot` command.

CHAPTER 3

Workflow Syntax and Execution Runtime

This section introduces the YAML syntax used by Popper, describes the workflow execution runtime and shows how to execute workflows in alternative container engines.

3.1 Syntax

A Popper workflow file looks like the following:

```yaml
steps:
- uses: docker://alpine:3.9
  args: ["ls", "-la"]

- uses: docker://alpine:3.11
  args: ["echo", "second step"]

options:
  env:
    FOO: BAR
  secrets:
  - TOP_SECRET
```

A workflow specification contains one or more steps in the form of a YAML list named `steps`. Each item in the list is a dictionary containing at least a `uses` attribute, which determines the docker image being used for that step. An `options` dictionary specifies options that are applied to the workflow.

3.1.1 Workflow steps

The following table describes the attributes that can be used for a step. All attributes are optional with the exception of the `uses` attribute.

3.1.2 Referencing images in a step

A step in a workflow can reference a container image defined in a `Dockerfile` that is part of the same repository where the workflow file resides. In addition, it can also reference a `Dockerfile` contained in public Git repository. A third option is to directly reference an image published a in a container registry such as DockerHub. Here are some examples of how you can refer to an image on a public Git repository or Docker container registry:

It's strongly recommended to include the version of the image you are using by specifying a SHA or Docker tag. If you don't specify a version and the image owner publishes an update, it may break your workflows or have unexpected behavior.

In general, any Docker image can be used in a Popper workflow, but keep in mind the following:

- When the `runs` attribute for a step is used, the `ENTRYPOINT` of the image is overridden.
- The `WORKDIR` is overridden and `/workspace` is used instead (see *The workspace* section below).
- The `ARG` instruction is not supported, thus building an image from a `Dockerfile` (public or local) only uses its default value.
- While it is possible to run containers that specify `USER` other than root, doing so might cause unexpected behavior.

3.1.3 Referencing private Github repositories

You can reference Dockerfiles located in private Github repositories by defining a `GITHUB_API_TOKEN` environment variable that the `popper run` command reads and uses to clone private repositories. The repository referenced in the `uses` attribute is assumed to be private and, to access it, an API token from Github is needed (see instructions here). The token needs to have permissions to read the private repository in question. To run a workflow that references private repositories:

```
export GITHUB_API_TOKEN=access_token_here
popper run -f wf.yml
```

If the access token doesn't have permissions to access private repositories, the `popper run` command will fail.

3.1.4 Workflow options

The `options` attribute can be used to specify `env` and `secrets` that are available to all the steps in the workflow. For example:

```
options:
  env:
    FOO: var1
    BAR: var2
  secrets: [SECRET1, SECRET2]

steps:
- uses: docker://alpine:3.11
  runs: sh
  args: ["-c", "echo $FOO $SECRET1"]

- uses: docker://alpine:3.11
  runs: sh
  args: ["-c", "echo $ONLY_FOR"]
  env:
    ONLY_FOR: this step
```

The above shows environment variables that are available to all steps that get defined in the `options` dictionary; it also shows an example of a variable that is available only to a single step (second step). This attribute is optional.

3.2 Execution Runtime

This section describes the runtime environment where a workflow executes.

3.2.1 The workspace

When a step is executed, a folder in your machine is bind-mounted (shared) to the `/workspace` folder inside the associated container. By default, the folder being bind-mounted is `$PWD`, that is, the working directory from where `popper run` is being invoked from. If the `-w` (or `--workspace`) flag is given, then the value for this flag is used instead.

For example, let's look at a workflow that writes to a `myfile` in the workspace:

```
steps:
- uses: docker://alpine:3.9
  args: [touch, ./myfile]
```

Assuming the above is stored in a `wf.yml` file, the following writes to the current working directory:

```
cd /tmp
popper run -f /path/to/wf.yml
```

In the above, `/tmp/myfile` is created. If we provide a value for `-w`, the workspace path changes and thus the file is written to that location:

```
cd /tmp
popper run -f /path/to/wf.yml -w /path/to
```

The above writes the `/path/to/myfile`. And, for completeness, the above is equivalent to:

```
cd /path/to
popper run -f wf.yml
```

3.2.2 Filesystem namespaces and persistence

As mentioned previously, for every step Popper bind-mounts (shares) a folder from the host (the workspace) into the `/workspace` folder in the container. Anything written to this folder persists. Conversely, anything that is NOT written in this folder will not persist after the workflow finishes, and the associated containers get destroyed.

3.2.3 Environment variables

A step can define, read, and modify environment variables. A step defines environment variables using the `env` attribute. For example, you could set the variables `FIRST`, `MIDDLE`, and `LAST` using this:

```
steps:
- uses: "docker://alpine:3.9"
  args: ["sh", "-c", "echo my name is: $FIRST $MIDDLE $LAST"]
  env:
```

(continues on next page)

(continued from previous page)

```
    FIRST: "Jane"
    MIDDLE: "Charlotte"
    LAST: "Doe"
```

When the above step executes, Popper makes these variables available to the container and thus the above prints to the terminal:

```
my name is: Jane Charlotte Doe
```

Note that these variables are only visible to the step defining them and any modifications made by the code executed within the step are not persisted between steps (i.e. other steps do not see these modifications).

Git Variables

When Popper executes insides a git repository, it obtains information related to Git. These variables are prefixed with `GIT_` (e.g. to `GIT_COMMIT` or `GIT_BRANCH`).

3.2.4 Exit codes and statuses

Exit codes are used to communicate about a step's status. Popper uses the exit code to set the workflow execution status, which can be `success`, `neutral`, or `failure`:

3.3 Container Engines

By default, Popper workflows run in Docker on the machine where `popper run` is being executed (i.e. the host machine). This section describes how to execute in other container engines. See *next section* for information on how to run workflows on resource managers such as SLURM and Kubernetes.

To run workflows on other container engines, an `--engine <engine>` flag for the `popper run` command can be given, where `<engine>` is one of the supported ones. When no value for this flag is given, Popper executes workflows in Docker. Below we briefly describe each container engine supported, and lastly describe how to pass engine-specific configuration options via the `--conf` flag.

3.3.1 Docker

Docker is the default engine used by the `popper run`. All the container configuration for the docker engine is supported by Popper.

3.3.2 Singularity

Popper can execute a workflow in systems where Singularity 3.2+ is available. To execute a workflow in Singularity containers:

```
popper run --engine singularity
```

Limitations

- The use of `ARG` in `Dockerfiles` is not supported by Singularity.
- The `--reuse` flag of the `popper run` command is not supported.

3.3.3 Host

There are situations where a container runtime is not available and cannot be installed. In these cases, a step can be executed directly on the host, that is, on the same environment where the `popper` command is running. This is done by making use of the special `sh` value for the `uses` attribute. This value instructs Popper to execute the command or script given in the `runs` attribute. For example:

```
steps:
- uses: "sh"
  runs: ["ls", "-la"]

- uses: "sh"
  runs: "./path/to/my/script.sh"
  args: ["some", "args", "to", "the", "script"]
```

In the first step above, the `ls -la` command is executed on the workspace folder (see *"The workspace"* section). The second one shows how to execute a script. Note that the command or script specified in the `runs` attribute are NOT executed in a shell. If you need a shell, you have to explicitly invoke one, for example:

```
steps:
- uses: sh
  runs: [bash, -c, 'sleep 10 && true && exit 0']
```

The obvious downside of running a step on the host is that, depending on the command being executed, the workflow might not be portable.

3.3.4 Custom engine configuration

Other than bind-mounting the `/workspace` folder, Popper runs containers with any default configuration provided by the underlying engine. However, a `--conf` flag is provided by the `popper run` command to specify custom options for the underlying engine in question (see here for more).

3.4 Resource Managers

Popper can execute steps in a workflow through other resource managers like SLURM besides the host machine. The resource manager can be specified either through the `--resource-manager/-r` option or through the config file. If neither of them are provided, the steps are run in the host machine by default.

3.4.1 SLURM

Popper workflows can run on HPC (Multi-Node environments) using Slurm as the underlying resource manager to distribute the execution of a step to several nodes. You can get started with running Popper workflows through Slurm by following the example below.

Let's consider a workflow `sample.yml` like the one shown below.

```yaml
steps:
- id: one
  uses: docker://alpine:3.9
  args: ["echo", "hello-world"]

- id: two
  uses: popperized/bin/sh@master
  args: ["ls", "-l"]
```

To run all the steps of the workflow through slurm resource manager, use the `--resource-manager` or `-r` option of the `popper run` subcommand to specify the resource manager.

```
popper run -f sample.yml -r slurm
```

To have more finer control on which steps to run through slurm resource manager, the specifications can be provided through the config file as shown below.

We create a config file called `config.yml` with the following contents.

```yaml
engine:
  name: docker
  options:
    privileged: True
    hostname: example.local

resource_manager:
  name: slurm
  options:
    two:
      nodes: 2
```

Now, we execute `popper run` with this config file as follows:

```
popper run -f sample.yml -c config.yml
```

This runs the step `one` locally in the host and step `two` through slurm on 2 nodes.

Host

Popper executes the workflows by default using the `host` machine as the resource manager. So, when no resource manager is provided like the example below, the workflow runs on the local machine.

```
popper run -f sample.yml
```

The above assumes `docker` as the container engine and `host` as the resource manager to be used.

CHAPTER 4

Guides

This is a list of guides related to several aspects of working with Popper workflows.

4.1 Choosing a location for your step

If you are developing a docker image for other people to use, we recommend keeping this image in its own repository instead of bundling it with your repository-specific logic. This allows you to version, track, and release this image just like any other software. Storing a docker image in its own repository makes it easier for others to discover, narrows the scope of the code base for developers fixing issues and extending the image, and decouples the image's versioning from the versioning of other application code.

4.2 Using shell scripts to define step logic

Shell scripts are a great way to write the code in steps. If you can write a step in under 100 lines of code and it doesn't require complex or multi-line command arguments, a shell script is a great tool for the job. When defining steps using a shell script, follow these guidelines:

- Use a POSIX-standard shell when possible. Use the `#!/bin/sh` shebang to use the system's default shell. By default, Ubuntu and Debian use the dash shell, and Alpine uses the ash shell. Using the default shell requires you to avoid using bash or shell-specific features in your script.
- Use `set -eu` in your shell script to avoid continuing when errors or undefined variables are present.

4.3 Hello world step example

You can create a new step by adding a `Dockerfile` to the directory in your repository that contains your step code. This example creates a simple step that writes arguments to standard output (`stdout`). An step declared in a `main.workflow` would pass the arguments that this step writes to `stdout`. To learn more about the instructions used in the Dockerfile, check out the official Docker documentation. The two files you need to create an step are shown below:

./step/Dockerfile

```
FROM debian:9.5-slim

ADD entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

./step/entrypoint.sh

```
#!/bin/sh -l

sh -c "echo $*"
```

Your code must be executable. Make sure the entrypoint.sh file has execute permissions before using it in a workflow. You can modify the permission from your terminal using this command:

```
chmod +x entrypoint.sh
```

This echos the arguments you pass the step. For example, if you were to pass the arguments "Hello World", you'd see this output in the command shell:

```
Hello World
```

4.4 Creating a Docker container

Check out the official Docker documentation.

4.5 Implementing a workflow for an existing set of scripts

This guide exemplifies how to define a Popper workflow for an existing set of scripts. Assume we have a project in a myproject/ folder and a list of scripts within the myproject/scripts/ folder, as shown below:

```
cd myproject/
ls -l scripts/

total 16
-rwxrwx---  1 user  staff   927B Jul 22 19:01 download-data.sh
-rwxrwx---  1 user  staff   827B Jul 22 19:01 get_mean_by_group.py
-rwxrwx---  1 user  staff   415B Jul 22 19:01 validate_output.py
```

A straight-forward workflow for wrapping the above is the following:

```
- uses: docker://alpine:3.12
  runs: "/bin/bash"
  args: ["scripts/download-data.sh"]

- uses: docker://alpine:3.12
  args: ["./scripts/get_mean_by_group.py", "5"]

- uses: docker://alpine:3.12
```

(continues on next page)

(continued from previous page)

```
  args [
    "./scripts/validate_output.py",
    "./data/global_per_capita_mean.csv"
  ]
```

The above runs every script within a Docker container. As you would expect, this workflow fails to run since the `alpine:3/12` image is a lightweight one (contains only Bash utilities), and the dependencies that the scripts need are not be available in this image. In cases like this, we need to either use an existing docker image that has all the dependencies we need, or create a docker image ourselves.

In this particular example, these scripts depend on CURL and Python. Thankfully, docker images for these already exist, so we can make use of them as follows:

```
- uses: docker://byrnedo/alpine-curl:0.1.8
  args: ["scripts/download-data.sh"]

- uses: docker://python:3.7
  args: ["./scripts/get_mean_by_group.py", "5"]

- uses: docker://python:3.7
  args [
    "./scripts/validate_output.py",
    "./data/global_per_capita_mean.csv"
  ]
```

The above workflow runs correctly anywhere where Docker containers can run.

CHAPTER 5

Other Resources

- A list of example workflows can be found at https://github.com/popperized/popper-examples.
- Self-paced hands-on tutorial.

CHAPTER 6

FAQ

6.1 How can I create a virtual environment to install Popper

The following creates a virtual environment in a `$HOME/venvs/popper` folder:

```
# create virtualenv
virtualenv $HOME/venvs/popper

# activate it
source $HOME/venvs/popper/bin/activate

# install Popper in it
pip install popper
```

The first step is is only done once. After closing your shell, or opening another tab of your terminal emulator, you'll have to reload the environment (`activate it` line above). For more on virtual environments, see here.

6.2 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.

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

A Popper pipeline is a simple sequence of "containerized 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.

6.4 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. See the https://github.com/popperized/popper-examples repository for examples.

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

The `popper run` takes an optional `STEP` argument that can be used to execute a workflow up to a certain step. Run `popper run --help` for more.

CHAPTER 7

Contributing

7.1 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.

7.2 Install from source

To install Popper in "development mode", we suggest the following approach:

```
cd $HOME/

# create virtualenv
python -m virtualenv $HOME/virtualenvs/popper

# load virtualenv
source $HOME/virtualenvs/popper/bin/activate

# clone popper
git clone git@github.com:systemslab/popper
cd popper

# install popper from source
pip install -e cli/[dev]
```

The `-e` flag passed to `pip` tells it to install the package from the source folder, and if you modify the logic in the popper source code you will see the effects when you invoke the `popper` command. So with the above approach you have both (1) popper installed in your machine and (2) an environment where you can modify popper and test the results of such modifications.

NOTE: The virtual environment created above needs to be reloaded every time you open a new terminal window (`source` commmand), otherwise the `popper` command will not be found by your shell.

7.3 Running tests

To run tests on your machine:

```
cd popper/

# activate the virtualenv
source $HOME/venvs/popper/bin/activate

# run all tests
python -X tracemalloc -m unittest -f cli/test/test_*

# run only one
python -X tracemalloc -m unittest -f cli/test/test_runner.py
```

7.4 Codestyle

Popper's code is formatted using the black style. If code does not conform to this style, merges are prevented to the master and this is checked as a CI step.

To apply black to your code, run black from the root Popper directory:

```
cd popper
black .
```

7.5 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.

7.6 Contributing example pipelines

We invite anyone to implement and document Github Action workflows. To add an example, you can fork an open a PR on the https://github.com/popperized/popper-examples repository.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search