# Popper Documentation

*Release 2.x*

**Ivo Jimenez**

**Jul 23, 2019**

# Contents

Popper is a workflow execution engine based on Github Actions (GHA) that allows you to execute GHA workflows locally on your machine. Popper workflows are defined in HCL syntax and behave like GHA workflows. The main difference with respect to GHA workflows is that, through some extensions to the GHA syntax, a Popper workflow can execute actions in other runtimes in addition to Docker.

# CHAPTER 1

# Getting Started

Popper is a workflow execution engine based on Github Actions written in Python. With Popper, you can execute workflows locally on your machine without having to use Github's platform. To get started, we first need to install the CLI tool using Pip:

```
pip install popper
```

Show which version you installed:

```
popper version
```

**NOTE**: Any version greater than 2.0 is currently officially supported.

To get a list of available commands:

```
popper --help
```

## 1.1 Create a Git repository

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'
```

## 1.2 Link to GitHub repository

First, 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
```

## 1.3 Create a workflow

We need to create a `.workflow` file:

```
popper scaffold
```

The above generates an example workflow that you can use as the starting point of your project. We first commit the files that got generated:

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

To learn more about how to modify this workflow in order to fit your needs, please take a look at the official documentation, read this tutorial or take a look at some examples.

## 1.4 Run your workflow

To execute the workflow you just created:

```
popper run
```

You should see the output of actions printed to the terminal.

## 1.5 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 --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 the TravisCI website to see your experiments being executed.

# 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:

```
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
```

or to execute all the workflows in a project:

```
popper run --recursive
```

## 2.3 Environment Variables

Popper defines the same environment variables that are defined by the official Github Actions runner. To see the values assigned to these variables, run the following workflow:

```
workflow "env workflow" {
  resolves = "show env"
}

action "show env" {
  uses = "actions/bin/sh@master"
  args = ["env"]
}
```

## 2.4 Reusing existing workflows

Many times, when starting an experiment, it is useful to be able to use an existing workflow as a scaffold for the one we wish to write. The `popper-examples` repository contains a list of example workflows and actions for the purpose of both learning and to use them as a starting point. Another examples can be found on Github's official `actions` organization.

Once you have found a workflow you're interested in importing, you can use the `popper add` command to obtain a workflow. For example:

```
cd myproject/
mkdir myworkflow
popper add https://github.com/popperized/popper-examples/workflows/cloudlab-iperf-test
Downloading workflow data-science as data-science...
Workflow docker-data-science has been added successfully.
```

This will download the contents of the workflow and all its dependencies to your project tree.

## 2.5 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
```

### 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 --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.

### 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 --service 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 --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.

### 2.5.4 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`).

## 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 this pipeline, execute the following:

```
popper dot
```

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

```
popper dot | 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

# Extensions

This section describes the extensions Popper brings on top of Github Actions.

**NOTE**: These extensions are not supported by the official Github Actions platform.

## 3.1 Downloading actions from arbitrary Git repositories

The syntax for defining actions in a workflow is the following:

```
action "IDENTIFIER" {
  needs = "ACTION1"
  uses = "docker://image2"
}
```

The `uses` attribute references Docker images, filesystem paths or github repositories (see syntax documentation for more). In the case where an action references a public repository, Popper extends the syntax in the following way:

```
{url}/{user}/{repo}/{path}@{ref}
```

The `{url}` can reference any Git repository, allowing workflows to reference to actions outside of Github. For example:

```
action "myaction on gitlab" {
  uses = "git@gitlab.com:user/repo/path/to/my/action@master"
}

action "another one on bitbucket" {
  uses = "https://bitbucket.com/user/repo/action@master"
}
```

The above references actions hosted on Gitlab and Bitbucket, respectively.

## 3.2 Other Runtimes

By default, actions in Popper workflows run in Docker, similarly to how they run in the Github Actions platform. Popper adds the ability of running actions in other runtimes by extending the interpretation of the `uses` attribute of action blocks.

> **NOTE**: As part of our roadmap, we plan to add support for Vagrant and Conda runtimes. Open a new issue to request another runtime you would Popper to support.

### 3.2.1 Singularity

> **NOTE**: This feature requires Singularity 2.6+.

An action executes in a Singularity container when:

- A singularity image is referenced. For example: `shub://myimage` will pull the container from the singularity hub.
- A `Singularity` file is found in the action folder. For example, if `./actions/mycontainer` is the value of the `uses` attribute in an action block, and a `Singularity` is found, Popper builds and executes a singularity container.
- A `Singularity` is found in the public repository of the given action. If an action resides in a public Git repository, and the path to the action contains a `Singularity` file, it will get executed in Singularity.

### 3.2.2 Host

There are situations where a container engine is not available and cannot be installed. In these cases, actions can execute directly on the host where the `popper` command is running. If an action folder does not contain a `Dockerfile` or `Singularity` file (be it a local path or the path in a public repository), the action will be executed on the host. Popper looks for an `entrypoint.sh` file and executes it if found, otherwise an error is thrown. Alternatively, if the action block specifies a `runs` attribute, the script corresponding to it is executed. For example:

```
action "run on host" {
  uses = "./myactions/onhost"
}

action "another execution on host" {
  uses = "./myactions/onhost"
  runs = "myscript"
}
```

In the above example, the `run on host` action is executed by looking for an `entrypoint.sh` file on the `./myactions/onhost/` folder. The `another execution on host` action will instead execute the `myscript` script (located in `./myactions/onhost/`).

Another way of executing actions on the host is to use the special `sh` value for the `uses` attribute. For example:

```
action "run on host" {
  uses = "sh"
  args = ["ls", "-la"]
}

action "another execution on host" {
  uses = "sh"
```

(continues on next page)

(continued from previous page)

```
  runs = "myscript"
  args = "args"
}
```

The above args `ls -la` on the root of the repository folder (the repository storing the `.workflow` file). This option allows users to execute arbitrary commands or scripts contained in the repository without having to define an action folder. The downside of this approach is that, depending on the command being executed, the workflow might not be portable.

> **NOTE**: The working directory for actions that run on the host depends on how the action is defined. If an action folder is present, the `$PWD` is set to the action folder. If `uses='sh'` is used, the `$PWD` is set to the root of the project. Thus, to ensure portability, scripts should use paths relative to the root of the folder. If absolute paths are needed, the `$GITHUB_WORKSPACE` variable can be used.

CHAPTER 4

# Examples

A list of example workflows can be found at https://github.com/popperized/popper-examples. Other examples can be found on Github's official `actions` organization.

CHAPTER 5

# Other Resources

- Official Github Actions documentation.
- Awesome-actions list.
- Self-paced hands-on tutorial.

CHAPTER 6

# FAQ

## 6.1 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.

## 6.2 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.

## 6.3 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. See the examples section for more.

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

The `popper run` takes an optional `action` argument that can be used to execute a workflow up to a certain step. See here.

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 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.

## 7.3 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.

### 7.3.1 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.

### 7.3.2 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 8

# Indices and tables

- genindex
- modindex
- search