Popper Documentation

Release 2.x

Ivo Jimenez

Mar 09, 2020

Contents

Popper is a Github Actions (GHA) workflow execution engine that allows you to execute GHA workflows locally on your machine and on CI services.

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
```

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

1.2 Create a workflow

First, we create an example `.workflow` file with a pre-defined workflow:

```
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.'
```

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.

1.3 Run your workflow

To execute the workflow you just created:

```
popper run
```

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

1.4 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
```

1.5 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 Searching for actions

The popper CLI is capable of searching for premade actions that you can use in your workflows.

You can use the `popper search` command to search for actions based on a search keyword. For example, to search for npm based actions, you can simply run:

```
$ popper search npm
Matched actions :

> actions/npm
```

Additionally, when searching for an action, you may choose to include the contents of the readme in your search by using the `--include-readme` flag.

Once `popper search` runs, it caches all the metadata related to the search. So, to get the latest releases of the actions, you might want to update the cache using the `--update-cache` flag.

By default, popper searches for actions from a list present here. To help the list keep growing, you can add Github organization names or repository names(org/repo) and send a pull request to the upstream repository.

To get the details of a searched action, use the `popper info` command. For example,

```
popper info popperized/cmake
An action for building CMake projects.
```

2.6 Continuously validating a workflow

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
```

2.6.1 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.

2.6.2 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
```

2.6.3 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.

2.6.4 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`).

2.6.5 Specifying which workflows to run via commit messages

When a CI service executes a popper workflow by invoking `popper run` on the CI server, it does so without passing any flags and hence we cannot specify which workflow to skip or execute. To make this more flexible, popper provides the ability to control which workflows to be executed by looking for special keywords in commit messages.

The `popper:whitelist[<list-of-workflows>]` keyword can be used in a commit message to specify which workflows to execute among all the workflows present in the project. For example,

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

popper:whitelist[/path/to/workflow/a.workflow]
```

The above commit message specifies that only the workflow `a` will be executed and any other workflow will be skipped. A comma-separated list of workflow paths can be given in order to request the execution of more than one workflow. Alternatively, a skip list is also supported with the `popper:skip[<list-of-workflows>]` keyword to specify the list of workflows to be skipped.

2.7 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
```

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

Guides

This is a list of guides related to several aspects of working with Github Action (GHA) workflows.

3.1 Implementing a Workflow for an Existing Set of Scripts

This guide exemplifies how to define a Github Action (GHA) 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:

```
workflow "co2 emissions" {
  resolves = "validate results"
}

action "download data" {
  uses = "actions/bin/sh@master"
  args = ["scripts/download-data.sh"]
}

action "run analysis" {
  needs = "download data"
  uses = "actions/bin/sh@master"
  args = ["workflows/minimal-python/scripts/get_mean_by_group.py", "5"]
}
```

(continues on next page)

(continued from previous page)

```
action "validate results" {
  needs = "run analysis"
  uses = "actions/bin/sh@master"
  args = [
    "workflows/minimal-python/scripts/validate_output.py",
    "workflows/minimal-python/data/global_per_capita_mean.csv"
  ]
}
```

The above runs every script within a Docker container, whose image is the one associated to the `actions/bin/sh` action (see corresponding Github repository here). As you would expect, this workflow fails to run since the `actions/bin/sh` 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 action that has all the dependencies we need, or create an action ourselves.

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

```
workflow "co2 emissions" {
  resolves = "validate results"
}

action "download data" {
  uses = "actions/bin/curl@master"
  args = [
    "--create-dirs",
    "-Lo workflows/minimal-python/data/global.csv",
    "https://github.com/datasets/co2-fossil-global/raw/master/global.csv"
  ]
}

action "run analysis" {
  needs = "download data"
  uses = "jefftriplett/python-actions@master"
  args = [
    "workflows/minimal-python/scripts/get_mean_by_group.py",
    "workflows/minimal-python/data/global.csv",
    "5"
  ]
}

action "validate results" {
  needs = "run analysis"
  uses = "jefftriplett/python-actions@master"
  args = [
    "workflows/minimal-python/scripts/validate_output.py",
    "workflows/minimal-python/data/global_per_capita_mean.csv"
  ]
}
```

The above workflow runs correctly anywhere where Github Actions workflow can run.

> ***NOTE**: The `download-data.sh` contained just one line invoking CURL, so we make that call directly in the action block and remove the bash script.*

3.1.1 When no container runtime is available

In scenarios where a container runtime is not available, the special `sh` value for the `uses` attribute of action blocks can be used. This value instructs Popper to execute actions directly on the host machine (as opposed to executing in a container runtime). The example workflow above would be rewritten as:

```
workflow "co2 emissions" {
  resolves = "validate results"
}

action "download data" {
  uses = "sh"
  args = [
    "curl", "--create-dirs",
    "-Lo workflows/minimal-python/data/global.csv",
    "https://github.com/datasets/co2-fossil-global/raw/master/global.csv"
  ]
}

action "run analysis" {
  needs = "download data"
  uses = "sh"
  args = [
    "workflows/minimal-python/scripts/get_mean_by_group.py",
    "workflows/minimal-python/data/global.csv",
    "5"
  ]
}

action "validate results" {
  needs = "run analysis"
  uses = "sh"
  args = [
    "workflows/minimal-python/scripts/validate_output.py",
    "workflows/minimal-python/data/global_per_capita_mean.csv"
  ]
}
```

The obvious downside of running actions directly on the host is that dependencies assumed by the scripts might not be available in other environments where the workflow is being re-executed. Since there are no container images associated to actions that use `sh`, this will likely break the portability of the workflow. In this particular example, if the workflow above runs on a machine without CURL or on Python 2.7, it will fail.

> ***NOTE**: The `uses = "sh"` special value is not supported by the Github Actions platform. This workflow will fail to run on GitHub's infrastructure and can only be executed using Popper.*

CHAPTER 4

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. Workflows using these extensions will fail to run on Github's infrastructure and can only be executed using Popper.

4.1 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 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 shows an example of a workflow referencing actions hosted on Gitlab and Bitbucket, respectively.

4.2 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 providing a `--runtime` flag to the `popper run` command.

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

4.2.1 Singularity

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

```
popper run --runtime singularity
```

When no `--runtime` option is supplied, Popper executes workflows in Docker.

Limitations

- The use of `ARG` in `Dockerfiles` is not supported by Singularity.
- Currently, the `--reuse` functionality of the `popper run` command is not available when running in Singularity.

4.2.2 Host

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

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

action "another execution on host" {
  uses = "sh"
  runs = "./path/to/my/script.sh"
  args = "args"
}
```

In the first example action above, the `ls -la` command is executed on the root of the repository folder (the repository storing the `.workflow` file). In the second one shows how to execute a script. The obvious downside of running actions on the host is that, depending on the command being executed, the workflow might not be portable.

> **NOTE**: The working directory (the value of `$PWD` when a command or script is executed) is the root of the project. Thus, to ensure portability, scripts should make use of paths relative to the root of the folder. If absolute paths are needed, the `$GITHUB_WORKSPACE` variable can be used.

CHAPTER 5

Other Resources

- Official Github Actions documentation.
- 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.
- Awesome-actions list.
- 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 `action` argument that can be used to execute a workflow up to a certain step. See here.

6.6 Installing Popper shows a `pyhcl` error

This project uses `pyhcl`, and when `pip` installs Popper, in some cases the below error is reported but it can be safely ignored.

```
Building wheels for collected packages: pyhcl
  Building wheel for pyhcl (setup.py) ... error
  ERROR: Complete output from command /Users/ivo/virtualenvs/test/bin/python3.7 -u -c
↪'import setuptools, tokenize;__file__='"'"'/private/var/folders/6c/pl43v
kgd0f5c29ffsnvkwvth0000gn/T/pip-install-kv3rwdd9/pyhcl/setup.py'"'"';
↪f=getattr(tokenize, '"'"'open'"'"', open)(__file__);code=f.read().replace('"'"'\r\n'
↪"'"',
 '"'"'\n'"'"');f.close();exec(compile(code, __file__, '"'"'exec'"'"'))' bdist_wheel -
↪d /private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-wheel-8m6v
ve9q --python-tag cp37:
  ERROR: running bdist_wheel
  running build
  running build_py
  Generating parse table...
  Traceback (most recent call last):
    File "<string>", line 1, in <module>
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/setup.py", line 101, in <module>
      "Topic :: Text Processing",
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/core.py", line 148, in setup
      dist.run_commands()
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/dist.py", line 966, in run_commands
      self.run_command(cmd)
```

(continues on next page)

(continued from previous page)

```
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/dist.py", line 985, in run_command
      cmd_obj.run()
    File "/Users/ivo/virtualenvs/test/lib/python3.7/site-packages/wheel/bdist_wheel.py
↪", line 192, in run
      self.run_command('build')
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/cmd.py", line 313, in run_command
      self.distribution.run_command(command)
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/dist.py", line 985, in run_command
      cmd_obj.run()
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/command/build.py", line 135, in run
      self.run_command(cmd_name)
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/cmd.py", line 313, in run_command
      self.distribution.run_command(command)
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/dist.py", line 985, in run_command
      cmd_obj.run()
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/setup.py", line 39, in run
      self.execute(_pre_install, (), msg="Generating parse table...")
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/cmd.py", line 335, in execute
      util.execute(func, args, msg, dry_run=self.dry_run)
    File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/
↪lib/python3.7/distutils/util.py", line 286, in execute
      func(*args)
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/setup.py", line 31, in _pre_install
      import hcl
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/src/hcl/__init__.py", line 1, in <module>
      from .api import dumps, load, loads
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/src/hcl/api.py", line 2, in <module>
      from .parser import HclParser
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/src/hcl/parser.py", line 4, in <module>
      from .lexer import Lexer
    File "/private/var/folders/6c/pl43vkgd0f5c29ffsnvkwvth0000gn/T/pip-install-
↪kv3rwdd9/pyhcl/src/hcl/lexer.py", line 3, in <module>
      import ply.lex as lex
  ModuleNotFoundError: No module named 'ply'
  ----------------------------------------
  ERROR: Failed building wheel for pyhcl
  Running setup.py clean for pyhcl
Failed to build pyhcl
```

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
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/
```

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.

7.3 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.

7.4 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