Popper Documentation

Release 2.x

Ivo Jimenez

Feb 18, 2020

Contents

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Create a Git repository . . . 4
- 1.3 Create a workflow . . . 4
- 1.4 Run your workflow . . . 5
- 1.5 Link to GitHub repository . . . 5
- 1.6 Continuously Run Your Workflow on Travis . . . 5

2 Workflow Language and Runtime **7**
- 2.1 Language . . . 7
- 2.2 Execution Runtime . . . 8

3 CLI feautures **11**
- 3.1 New workflow initialization . . . 11
- 3.2 Executing a workflow . . . 12
- 3.3 Customizing container engine behavior . . . 12
- 3.4 Environment Variables . . . 12
- 3.5 Reusing existing workflows . . . 13
- 3.6 Searching for actions . . . 13
- 3.7 Continuously validating a workflow . . . 13
- 3.8 Visualizing workflows . . . 15

4 Guides **17**
- 4.1 Creating a new action . . . 17
- 4.2 Creating a Docker container . . . 19
- 4.3 Implementing a workflow for an existing set of scripts . . . 19

5 Other Resources **23**

6 FAQ **25**
- 6.1 How can I create a virtual environment to install Popper . . . 25
- 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? . . . 25
- 6.3 How can Popper capture more complex workflows? For example, automatically restarting failed tasks? 26
- 6.4 Can I follow Popper in computational science research, as opposed to computer science? . . . 26
- 6.5 How to apply the Popper protocol for applications that take large quantities of computer time? . . . 26

7 Contributing **27**
7.1 Code of Conduct 27
7.2 Install from source 27
7.3 Contributing CLI features 28
7.4 Contributing example pipelines 28

8 Indices and tables **29**

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

CHAPTER 1

Getting Started

Popper is a workflow execution engine based on Github Actions (GHA) written in Python. With Popper, you can execute HCL syntax workflows locally on your machine without having to use Github's platform.

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. Here we present three alternatives that cover most of these alternatives:

```
# 1) virtualenv 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

# 2) virtualenv installed via Python 3.6+ built-in 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 is ended (when we close the terminal window or tab), the environment is 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'
```

1.3 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 workflow language 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.

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.

CHAPTER 2

Workflow Language and Runtime

This section introduces the HCL-based workflow language used by Popper and also describes the execution runtime.

> **NOTE**: The workflow language employed by Popper is **NOT** supported by the official Github Actions platform. The HCL syntax for workflows was introduced by Github on 02/2019 and later deprecated on 09/2019. The Popper project still uses the HCL syntax.

2.1 Language

The following example workflow contains one workflow block and two action blocks.

```
workflow "IDENTIFIER" {
  resolves = "ACTION2"
}

action "ACTION1" {
  uses = "docker://image1"
}

action "ACTION2" {
  needs = "ACTION1"
  uses = "docker://image2"
}
```

In this example, the workflow invokes `ACTION2` using `resolves`, but because it `needs` `ACTION1`, the `ACTION1` block executes first. `ACTION2` will execute once `ACTION1` has successfully completed. For more information on why this happens, see "Workflow attributes" and "Action attributes" below.

2.1.1 Workflow blocks

A workflow file contains only one `workflow` blocks, each with a unique identifier and the attributes outlined in the workflow attributes table.

Workflow attributes

2.1.2 Action blocks

A workflow file may contain any number of action blocks. Action blocks must have a unique identifier and must have a `uses` attribute. Example action block:

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

Action attributes

2.1.3 Using a `Dockerfile` image in an action

When creating an action block, you can use an action defined in the same repository as the workflow, a public repository, or in a published Docker container image. An action block refers to the images using the `uses` attribute. It's strongly recommended to include the version of the action you are using by specifying a SHA or Docker tag number. If you don't specify a version and the action owner publishes an update, it may break your workflows or have unexpected behavior. Here are some examples of how you can refer to an action on a public Git repository or Docker container registry:

2.1.4 Referencing private Github repositories in an action

We can make use of actions located in private Github repositories by defining a `GITHUB_API_TOKEN` environment variable that the `popper run` command reads and uses to clone private Github repositories. To accomplish this, 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
```

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

2.2 Execution Runtime

This section describes the runtime environment where a workflow executes.

2.2.1 Environment variables

An action can create, read, and modify environment variables. When you create an action in a workflow, you can define environment variables using the `env` attribute in your action block. For example, you could set the variables `FIRST_NAME`, `MIDDLE_NAME`, and `LAST_NAME` using this example action block:

```
action "hello world" {
  uses = "./my-action"
  env = {
```

(continues on next page)

(continued from previous page)

```
    FIRST_NAME  = "Mona"
    MIDDLE_NAME = "Lisa"
    LAST_NAME   = "Octocat"
  }
}
```

When an action runs, Popper also sets these environment variables in the runtime environment:

2.2.2 Naming conventions

Any new environment variables you set that point to a location on the file system should have a `_PATH` suffix. The `HOME` and `GITHUB_WORKSPACE` default variables are exceptions to this convention because the words "home" and "workspace" already imply a location.

2.2.3 Filesystem

Two directories are bind-mounted on the `/github` path prefix. These two directories are shared from the host machine to the containers running in a workflow:

Exit codes and statuses

You can use exit codes to provide an action's status. Popper uses the exit code to set the workflow execution status, which can be `success`, `neutral`, or `failure`:

2.2.4 Alternative container engines

By default, actions in Popper workflows run in Docker. In addition to Docker, Popper can execute workflows in other runtimes by interacting with other container engines. A `--engine <engine>` flag for the `popper run` can be used to invoke alternative engines (where `<engine>` is one of the supported engines). When no value for the `--engine` option is given, Popper executes workflows in Docker.

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

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.
- Currently, the `--reuse` functionality of the `popper run` command is not available when running in Singularity.

Vagrant

While technically not a container engine, executing workflows inside a VM allows users to run workflows in machines where a container engine is not available. In this scenario, Popper uses Vagrant to spawn a VM provisioned with Docker. It then executes workflows by communicating with the Docker daemon that runs inside the VM. To execute a workflow in Vagrant:

```
popper run --engine vagrant
```

Limitations

Currently, only one workflow can be executed at the time in Vagrant runtime, since popper assumes that there is only one VM running at any given point in time.

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 3

CLI feautures

3.1 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'
```

3.2 Executing a workflow

To run the workflow:

```
popper run
```

or to execute all the workflows in a project:

```
popper run --recursive
```

3.3 Customizing container engine behavior

By default, Popper instantiates containers in the underlying engine by using basic configuration options (see here). 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_configuration` dictionary with custom options and is passed to the `popper run` command via the `--engine-conf` flag.

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

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

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

```
popper run --engine-conf settings.py
```

NOTE:

1. Currently, the `--engine-conf` option is only supported for the `docker` engine.
2. The `settings.py` file must contain a `dict` type variable with the name `engine_configuration` as shown above.

3.4 Environment Variables

Popper defines a set of environment variables (see Environment Variables section) that are available to all actions. To see the values assigned to these variables, run the following workflow:

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

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

To define new variables, the `env` keyword can be used (see Action Attributes for more).

3.5 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.

3.6 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 :

> popperized/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.
```

3.7 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
```

3.7.1 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.

3.7.2 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 circle
git add .circleci
git commit -m 'Adds CircleCI config files'
git push
```

3.7.3 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 gitlab
git add .gitlab-ci.yml
git commit -m 'Adds GitLab-CI config file'
git push
```

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

3.7.4 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`).

3.7.5 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.

3.8 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 4

Guides

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

4.1 Creating a new action

You can create actions in a repository you own by adding a `Dockerfile`. To share GitHub Actions with the GitHub community, your repository must be public. All actions require a `Dockerfile`. An action may also include an `entrypoint.sh` file, to execute arguments, and any other files that contain the action's code. For example, an action called `action-a` might have this directory structure:

```
|-- hello-world (repository)
|   |__ main.workflow
|   |__ action-a
|       |__  Dockerfile
|       |__  README.md
|       |__  entrypoint.sh
|
```

To use an action in your repository, refer to the action in your `.workflow` using a path relative to the repository directory. For example, if your repository had the directory structure above, you would use this relative path to use `action-a` in a workflow for the `hello-world` repository:

```
action "action a" {
  uses = "./action-a/"
}
```

Every action should have a `README.md` file in the action's subdirectory that includes this information:

- A detailed description of what the action does.
- [Environment variables][env-vars] the action uses.
- [Secrets][secrets] the action uses. Production secrets should not be stored in the API during the limited public beta period.

- Required arguments.
- Optional arguments.

See Creating a Docker container to learn more about creating a custom Docker container and how you can use `entrypoint.sh`.

4.1.1 Choosing a location for your action

If you are developing an action for other people to use, GitHub recommends keeping the action in its own repository instead of bundling it with other application code. This allows you to version, track, and release the action just like any other software. Storing an action in its own repository makes it easier for the GitHub community to discover the action, narrows the scope of the code base for developers fixing issues and extending the action, and decouples the action's versioning from the versioning of other application code.

4.1.2 Using shell scripts to create actions

Shell scripts are a great way to write the code in GitHub Actions. If you can write an action 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 writing actions using a shell script, following 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.

4.1.3 Hello world action example

You can create a new action by adding a `Dockerfile` to the directory in your repository that contains your action code. This example creates a simple action that writes arguments to standard output (`stdout`). An action declared in a `main.workflow` would pass the arguments that this action 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 action are shown below:

./action-a/Dockerfile

```
FROM debian:9.5-slim

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

./action-a/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 action `echos` the arguments you pass the action. For example, if you were to pass the arguments `"Hello World"`, you'd see this output in the command shell:

```
Hello World
```

4.2 Creating a Docker container

Check out the official Docker documentation.

4.3 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 = "popperized/bin/sh@master"
  args = ["scripts/download-data.sh"]
}

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

action "validate results" {
  needs = "run analysis"
  uses = "popperized/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 `popperized/bin/sh` action (see corresponding Github repository here). As you would expect, this workflow fails to run since the `popperized/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 = "popperized/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.*

4.3.1 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"
  ]
}
```

(continues on next page)

(continued from previous page)

```
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 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. 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 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