Contributor CI Documentation

Release 0.0.14

Vanessa Sochat

Mar 04, 2022

GETTING STARTED

Contributor CI provides tools for assessing collaboration. You can use the associated GitHub actions to collect metrics on a regular basis, or the visualization tools to look at them. You might want to use Contributor CI if you are interested in collecting data to assess collaboration over time.

CHAPTER ONE

SUPPORT

- For **bugs and feature requests**, please use the issue tracker.
- For **contributions**, visit Contributor CI on Github.

RESOURCES

GitHub Repository The code for Contributor CI on GitHub.

2.1 Getting Started

Contributor CI is a set of tools for assessing and visualizing contributions. If you have any questions or issues, please let us know

2.1.1 Installation

Contributor CI can be installed from pypi, or from source. For either, it's recommended that you create a virtual environment, if you have not already done so.

Install from Pypi

To install from pypi:

```
$ pip install contributor-ci
```

Install from GitHub

If you want to install directly from GitHub (perhaps a development release or a tag) you can clone the repository:

```
$ git clone git@github.com:vsoch/contributor-ci
# you can also do git clone https://github.com/vsoch/contributor-ci
$ cd contributor-ci
$ python setup.py install
# pip install . or pip install -e . for development
```

Installation of adds an executable, *cci* to your path.

`` `bash $ which cci /opt/conda/bin/cci ` ``

See the *Getting Started* pages for next steps.

2.1.2 User Guide

Contributor CI was created with the intention to help measure collaboration in one or more repositories or an organization over time. If you haven't read *Installation* you should do that first.

Quick Start

```
# install contributor ci
$ pip install contributor-ci

# Export a GitHub Personal Access Token
$ export GITHUB_TOKEN=xxxxxxxxxxxxxx

# Generate a contributor-ci.yaml (recommended in an empty directory)
$ mkdir -p cci-vsoch
$ cci init vsoch

# See metric extractors available
$ cci list

# Run an extraction
$ cci extract repos

# Generate a contributor friendliness assessment template
$ cci cfa https://github.com/vsoch/salad
```

Configuration

Since the majority of the command line interactions should be automated, we use a main configuration file called `contributor-ci.yaml`, that is looked for in the present working directory where you call the client. A simple example is shown here:

```
# The output directory to create a structure of results
# defaults to $PWD/.cci if not set
outdir: null

# memberOrgs will be used to label associated members as "internal"
member_orgs:
  - llnl

# all repos in these orgs are considered in your institution
orgs:
 - cdat
 - llnl
 - mfem

# Additional repos to add to the set (possibly not under an org above)
repos:
 - alpine-dav/ascent
 - atomconsortium/ampl
 - ceed/benchmarks

# do not include these repos in the assessment
exclude_repos:
 - mfem/github-actions
```

This file should have the following fields:

Table 1: Title

Name	Description	Default	Required
member_orgs	A list of GitHub organizations that are core to your institution. If you are concerned about contibutions, everyone that is a member here is labeled as an internal contributor.	unset	yes
orgs	A list of repos your organization members contribute to, but aren't necessarily owned by your institution.	unset	yes
repos	A list of loose repos to add to the ones that are discovered under the orgs already provided.	unset	false
exclude_repos	One or more repos to exclude given that they are found anywhere.	unset	false
outdir	An output directory (must exist) to save results.	$PWD/.cci	true
editor	An editor to use for cci config edit	vim	false

You can make this file manually if you like (e.g., copy paste and edit the above) or you can use `cci init <name>` to initialize one, either for an organization or your username. To generate it for your username, you can run init as follows:

```
$ export GITHUB_TOKEN=xxxxxxxxx
$ cci init user:vsoch
```

And for an organization of interest:

```
$ export GITHUB_TOKEN=xxxxxxxxx
$ cci init org:spack
```

The place where you run this init depends on your use case. If you intent to create a CCI interface, you might want to run this init in an empty directory. If you just want to run cci extract commands to generate data, it can be in an existing repository.

```
$ mkdir -p cci-vsoch
$ cd cci-vsoch
$ cci init user:vsoch
```

After having your file, if you want to create an interface that visualizes data and contributor friendliness assessments (CFAs), you should see *User Interface (UI)*. For basic extraction or generation of CFAs, you should see *Contributor Friendliness Assessment*.

Commands

Once you have your configuration file, and exported a GitHub personal access token:

```
# Export a GitHub Personal Access Token
$ export GITHUB_TOKEN=xxxxxxxxxxxxxx
```

...the following commands are available! For any command, you can specify a custom configuration file or output directory:

```
$ cci --config-file <config-file> --out-dir <out-dir> <command> <args>
```

User Interface (UI)

The most straight forward thing you might want is a user interface to explore your repositories. Once you have generated your contributor-ci.yaml in an otherwise empty directory (and it's recommended to edit it after generation to make sure it looks okay) creating an interface is as simple as running:

```
$ cci ui generate
```

By default, generation happens in the directory where you run the command. To change this:

```
$ cci ui generate ./docs
```

Whether you are generating or updating, if you want to also generate new files for the contributor friendliness assessment, you can add –cfa:

```
$ cci ui --cfa generate
```

Once you have an interface, it has a GitHub action that will run an update command on a nightly basis to generate new data for it. But you can also run this locally or manually. It takes the same argument for a directory or defaults to the present working directory.

```
$ cci ui update
```

And akin to generate, you can specify to generate new CFA files (default will not):

```
$ cci ui --cfa update
```

For update and the GitHub action, the default will be to run all extractors. However for large projects you might instead want to choose a random subset:

```
$ cci ui update random:4
```

Once you have your interface, it's recommended to update your `GITHUB_TOKEN` to a personal access token `CCI_GITHUB_TOKEN` that will work for all extractors. You are of course free to customize the interface to your pleasing. For example, you will likely want to change the site baseurl (the name of your repository where you will serve it), site metadata, and the sidebar highlight color:

```
name: "Contributor CI Software Portal"
author: "Contributor CI <vsoch@users.noreply.github.com>"
title: Contributor CI Software Portal
description: "Contributor CI Software Portal"

# Change this to your baseurl
baseurl: "/contributor-ci"
url: ""

# Change this to the color you want the sidebar to highlight to
# it defaults to a bright green to match contributor-ci
sidebar_highlight_color: "#00d100"
```

For content, the main page is located in `pages/index.md`, and you can delete any graph that you don't want to show up by deleting the corresponding file from `_graphs`. If there is a graph that does not exist that you'd like, or another site feature (e.g., posts or other content type) please open an issue. Finally, you'll want to push your interface to GitHub and ensure that GitHub pages is turned on for the root or subfolder where you have your site.

Visualizations

The interface (by default) will generate the following files, each linked to a specific set of data and javascript files. If you want to remove any particular visualization from your interface, you can simply delete the markdown file. All javascript files are located in `assets/js/extractors`

Table 2: Title

Markdown	Description	Data	Javascript
activity_commits.md	A graph that shows activity across all repos (default branches) for a year (activity commits example).	`activity_commits.js`	`cci-activity_commits.json` and `cci-repos.json`
dependencies.md	A hierarchical DAG that shows depdencies (dependencies example).	`dependencies.js`	`cci-repo_dependencies.json` and `cci-dependencies.json`
languages.md	A donut circle graph that shows a breakdown of repository languages (languages example).	`languages.js`	`cci-languages.json`
licenses.md	A donut circle graph that shows a breakdown of repository licenses (licenses example).	`licenses.js`	`cci-repos.json`
member_repos.md	Counts of organization members and repositories (member repos example).	`member_repos.js`	`cci-internal-users.json` and `cci-member_repos.json`
pack_users.md	A zoomable pack hierarchy that shows organizations and contributors (pack users example).	`pack_users.js`	`cci-internal-users.json` and `cci-external-users.json`
repos_issues.md	Scatterplot showing repository closed. vs open issues (issues example).	`repos_issues.js`	`cci-repos.json`
repos.md	Scatterplot of repository creation history by year (creation history example).	`creation_history.js`	`cci-creation-history.json`
repos_pulls.md	Scatterplot showing repository open. vs merged pull requests (pulls example).	`repos_pulls.js`	`cci-repos.json`
repos_size.md	Bubble chart of repositories by popularity size (number of stars) (size example).	`repos_size.js`	`cci-repos.json`
star_history.md	Number of stars over time (stars example).	`star_history.js`	`cci-stars.json`
topics.md	Tag map of repository topics (topics example).	`topics.js`	`cci-topics.json`

Config

Contributor CI provides an easy way to interact with your configuration file, the file `contributor-ci.yaml`. First, to edit the file, you can do:

```
$ cci config edit
```

By default, the editor chosen is vim. If you add an `editor` field to that same file, you can choose an editor of your choice. You can also quickly sort your file in the case that you made a bunch of additions and want to ensure they are sorted. Note that sorting happens automatically when you do an add or remove operation.

```
$ cci config sort
```

Next, you might want to add a repository or organization to a list. You can use add and remove to do this. You should provide the key first (e.g. member_orgs) followed by one more entries to add or remove.

```
$ cci config add member_orgs vsoch
$ cci config remove member_orgs vsoch
```

List

You likely want to start with an extraction. An extraction means that you are extracting metadata for the current data, and for your current set of repos. But first you need to know what your options are! For this purpose you can use `list`:

```
$ cci list
    creation_history: extract creation history for repositories.
              topics: extract repository topics.
   repo_dependencies: extract repository dependencies.
           languages: extract languages for a repository.
    activity_commits: extract internal repository commit activity.
            releases: extract repository releases.
               stars: extract repository stars.
        member_repos: extract repositories that belong to members not within org.
      activity_lines: extract internal repository activity via lines of code.
        dependencies: extract dependencies.
       repo_metadata: gather repository metadata from several extractors.
               repos: extract repository metrics.
               users: extract user metrics for a repository.
          repo_users: extract repositories worked on for external and internal users.
```

Extract

You next likely want to run an extractor. The default output directory used will be a directory named `.cci` for "contributor CI" in the present working directory.

```
$ cci extract repos
Retrieving organization info for cdat
Checking GitHub API token... Token validated.
Auto-retry limit for requests set to 10.
Reading '/home/vanessa/Desktop/Code/contributor-ci/contributor_ci/main/extractors/
↪collection/repos/org-repos-info.gql' ... File read!
Page 1
Sending GraphQL query...
Checking response...
HTTP STATUS 200 OK
```

When it finished, you can inspect the output in the present working directory ".cci" folder (unless you changed the path in the config or on the command line). It is a tree organized by year, month, and day:

```
 $ tree .cci/
.cci/data/
└── 2021
    └── 6
        └── 13
            └── cci-repos.json
```

You'll notice that the extracted data is saved in a "data" subfolder. This is because there are other output types that can be saved here. You can also ask CCI to run more than one extractor at once:

```
$ cci --out-dir _data extract repos repo_metadata topics
```

Finally, if you want to change the output organization (which defaults to `year/month/day` under the data folder) you can add `--save-format`:

```
$ cci --out-dir _data extract --save-format year/month repos repo_metadata topics
```

Note that since CCI uses its directories as a cache, changing the default save format will change this behavior to generate the data no matter what, as we cannot be confident when the data was actually generated. If you find that you don't want this behavior, it's recommended to run with the default save format and then clean up or organize the data directory as you see fit.

Extractors

The following extractors are available.

Table 3: Contributor CI Extractors

Name	Description	Depends On
repos	Extract repository metadata	none
users	Extract internal and external contributors lists	repos
repo_dependencies	Extract repository dependencies	repos
dependencies	Extract dependency metadata	repo_dependencies
releases	Extract releases for repositories	repos
languages	Extract languages for repositories	repos
activitycommits	Extract weekly number of repository commits to reflect activity	repos
repo_users	Extract users and repositories contributed to (internal and external)	users
creation_history	Extract creation history (first commit) of repositories	repos
stars	Extract repository stars	repos
member_repos	Extract repositories of members not associated with the organization	users
topics	Extract repository topics	repos
repo_metadata	Combine repository metadata across repps and topics extractors	repos, topics

Contributor Friendliness Assessment

The Contributor Friendliness Assessment (CFA) is an effort to identify aspects of a repository that can be improved to make the repository more contributor friendly. The assessment derives a list of criteria to assess how easy it is to contribute to a project. This means arriving at a project repository and having an easy time going from knowing nothing to opening a pull request, and also how well the project attracts new contributors. Generally, we assess the repository for:

- `CFA-branding`: Does the project have branding?
- `CFA-popularity`: How popular is the project?
- `CFA-description`: Does the project have a clear description (What is it for)?
- `CFA-need`: Does the project have a compelling set of use cases, or statement of need (Should I use it)? This is a fork in the visitor's decision tree, because if the answer is yes they will continue exploring, otherwise they will not.
- `CFA-license`: The GitHub repository has an OSI-approved open-source license.

- `CFA-build`: Methods to build or install the software or service are clearly stated.
- `CFA-examples`: Does the README.md have a quick example of usage?
- `CFA-documentation`: Does the project have documentation?
- `CFA-support`: Does the project make it easy to ask for help?
- `CFA-developer`: Process and metadata is provided for the developer to understand and make changes.
- `CFA-quality`: The code quality of the project.
- `CFA-tests`: The project has testing.
- `CFA-coverage`: The project reports code coverage.
- `CFA-format`: The project adheres to a language specific format.
- `CFA-outreach`: Is the project active at conferences or otherwise externally presented?

Each of the items above has a more detailed description, rationale, and list of criteria – some of which are automated. Currently, the assessment is under development so running the `cfa` tool for a repository:

```
# Generate a contributor friendliness assessment template and print to terminal
$ cci cfa --terminal https://github.com/vsoch/salad

# Save to local .cci directory
$ cci cfa https://github.com/vsoch/salad

# Pipe into file
$ cci cfa --terminal https://github.com/LLNL/b-mpi3 > _cfa/cfa-LLNL-b-mpi3.md
```

For the latter, your cfa template (with some fields populated) will be saved to your .cci output directory, as specified in your config or on the command line:

```
$ tree .cci/cfa/
└── cfa-vsoch-salad.md
```

Will simply output the template to be filled in. This will be updated with automation and allowing for save in the `.cci` output folder, allowing for creating new assessments, and updating previously created assessments. We will also provide a GitHub action for generating assessment files and opening a pull request when new repositories are found that have not been assessed.

CFA Background

The author of CCI noticed that there are many good software projects, but they don't do a good job of explaining use cases. She also noticed that small details like branding, documentation, and ease of use were hugely important variables for making it easy to contribute. You can imagine a sequence of events (a decision tree) that models a user interaction:

1. Arrive at the repository.
2. Assess project for branding and popularity.
3. What does it do?
4. Does it help with a problem that I have yes –> continue, no–> leave?
5. Does it have a license that I like?
6. Install / build the software to try out
7. Look for a getting started guide or examples

8. Make changes to the repository, sometimes look for contributing guide.
9. Run local tests, formatting, etc.
10. Open a pull request

GitHub Action

Contributor CI comes with a GitHub action that will be more developed as the library is developed. Currently, you can use it to run one or more extractors for a `contributor-ci.yaml` in your repository. For example, let's say we want to run all extractors:

```yaml
name: Contributor CI Extract
on:
  schedule

    # Every Sunday
    - cron: 0 0 * * 0

jobs:
  run:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout Actions Repository
      uses: actions/checkout@v2
    - name: Extract
      uses: vsoch/contributor-ci@main
      env:
        CCI_GITHUB_TOKEN: ${{ secrets.CCI_GITHUB_TOKEN }}
      with:
        extract: repos
        results_dir: .cci
        config_file: contributor-ci.yaml

    - name: Check that results exist
      run: tree .cci

    - name: Upload results
      if: success()
      uses: actions/upload-artifact@v2-preview
      with:
        name: cci-results
        path: .cci
```

Note that `CCI_GITHUB_TOKEN` is recommended to be a personal access token, which is needed for some of the queries to look at organizations. If you just need repository metadata, the standard `GITHUB_TOKEN` provided in actions will suffice. You can either save as an artifact as shown above, or just push directly to a branch:

```yaml
- name: Push Results
  run:
    git config --global user.name "github-actions"
    git config --global user.email "github-actions@users.noreply.github.com"
    git add _cci

    set +e
    git status | grep modified
    if [ $? -eq 0 ]; then
```

(continues on next page)

(continued from previous page)

```
        set -e
        printf "Changes\n"
        git commit -m "Automated push with new data results $(date '+%Y-%m-%d')" ||␣
↪exit 0
        git push origin main
    else
      set -e
      printf "No changes\n"
    fi
```

You can also use a pull request action to open a pull request instead. The action can also support generating Contributor Friendliness Assessment (markdown) files. Since these might warrant being populated into an interface, if you select a `results_dir` here, the markdown files will explicitly be written there. If not, then they will be written to the default in `.cci/cfa`.

```
name: Contributor CI Update Contributor Friendliness Assessment
on:
  schedule

    # Every Sunday
    - cron: 0 0 * * 0

jobs:
  run:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout Actions Repository
      uses: actions/checkout@v2
    - name: Update CFAs
      uses: vsoch/contributor-ci@main
      env:
        CCI_GITHUB_TOKEN: ${{ secrets.CCI_GITHUB_TOKEN }}
      with:
        cfa: true
        results_dir: ./cfa
        config_file: contributor-ci.yaml

    - name: Check that results exist
      run: tree ./cfa
```

Finally, you can ask to run more than one extractor, akin to how you can on the command line!

```
jobs:
  extraction:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2
    - name: Update Data
      uses: vsoch/contributor-ci@main
      env:
        GITHUB_TOKEN: ${{ secrets.CCI_GITHUB_TOKEN }}
      with:
        results_dir: _data/
        extract: repo_metadata topics languages releases stars activity_commits␣
↪activity_lines
```

2.1.3 Developer Guide

This developer guide includes instructions for how to write an extractor. If you haven't read *Installation* you should do that first.

Writing an Extractor

An extractor is a directory in the folder `contributor_ci/main/extractors`. The directory should correspond to the name of the extractor (e.g., users or repos) and within should minimally be an `extract.py` file.

Extractor Base Classes

You can use an extractor base class to get access to all the functions to save, load, and otherwise run an extraction. Specifically, if you are using a GitHub extractor, you can import the `GitHubExtractorBase`:

```
from contributor_ci.main.extractor import GitHubExtractorBase
```

If you don't require GitHub and are extracting metadata in some other way, the `ExtractorBase` should be sufficient.

```
from contributor_ci.main.extractor import ExtractorBase
```

GitHub Extractor Base

The `GitHubExtractorBase` required a `GITHUB_TOKEN` to be exported in the environment, and comes with a `self.manager` that is a query manager from this scraper tool. This tool uses graphQL queries that should be located in the same directory as the extractor. For example, the users extractor has several query files (extension *.gql) as you can see here:

```
$ tree contributor_ci/main/extractors/users/
contributor_ci/main/extractors/users/
├── extract.py
├── __init__.py
├── org-members.gql
└── repo-users.gql
```

There are several helper functions to support loading files, and reading any previously extracted dependency files. As long as you add `depends_on` to your extractor, there is an `ExtractorResolver` class that will make sure your dependency data is produced before the extractor is run. You can do any of the following:

```
# Load the dependency file named cci-repos.json
# repos.data will have the loaded data
repos = self.load_dependency_file("repos")

# Load the query filename org-repos-info.gql in the extractor directory
org_query = self.get_local_query("org-repos-info.gql")
```

For running queries, it's recommended that you look at already existing GitHub extractors for examples.

Extractor Metadata

Each extractor is required to have a set of properties that help to identify it. Specifically:

Table 4: Title

Name	Description	Required
name	The extractor name, which should match the folder it lives in.	true
description	A description of the extractor.	true
filenames	The filename identifiers that the extractor is expected to save. E.g., if the "repos" extractor saves a file called "cci-repos.json", you would provide a list with "repos."	true
depends_on	A list of other extractor names that this extractor depends on	false

You will also want to name your extractor the same as the directory and name, but uppercase. This is how the class is discovered. As an example, here is the "users" extractor.

```
class Users(GitHubExtractorBase):

    name = "users"
    description = "extract user metrics for a repository."
    depends_on = ["repos"]
    filenames = ["internal-users", "external-users"]
```

This extractor requires that the "repos" extractor is run first (the depends_on field) because we need a list of organization repositories to find members in. This means that if someone runs:

```
cci extract users
```

The "repos" extractor will be run first as the dependency. You'll also notice that filenames include "internal-users" and "external-users," and these will generate output files in the nested output directory named accordingly. After running this extractor, you'll see:

```
$ tree .cci/
.cci/
└── 2021
    └── 6
        ├── 3
        │   └── cci-repos.json
        └── 5
            ├── cci-external-users.json
            ├── cci-internal-users.json
            └── cci-repos.json

4 directories, 4 files
```

Extractor Functions

Your extractor is required to have one main function called `extract` to do whatever extraction is needed and save results to `self._data`. Importantly, the keys to `self._data` should correspond with the file key you intend to save. For the repos extractor, this means we save data to `self._data["repos"]`` or just `self._data[self.name]` and for the users extractor we expect to find data keys "internal-users" and "external-users." That's it! As long as you have a function to extract, provide the necessary metadata, and populate the data into `self._data` correctly, you should be good to go.

2.2 Contributor CI

These sections detail the internal functions for Contributor CI.

2.3 Internal API

These pages document the entire internal API of Contributor CI.

2.3.1 cci package

Submodules

contributor_ci.client module

`contributor_ci.client.`**`get_parser`**`()`

`contributor_ci.client.`**`run`**`()`

contributor_ci.logger module

`class` `contributor_ci.logger.`**`ColorizingStreamHandler`**`(`*nocolor=False, stream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>, use_threads=False*`)`

Bases: `logging.StreamHandler`

`BLACK = 0`

`BLUE = 4`

`BOLD_SEQ = '\x1b[1m'`

`COLOR_SEQ = '\x1b[%dm'`

`CYAN = 6`

`GREEN = 2`

`MAGENTA = 5`

`RED = 1`

`RESET_SEQ = '\x1b[0m'`

WHITE = 7

YELLOW = 3

can_color_tty()

colors = {'CRITICAL': 1, 'DEBUG': 4, 'ERROR': 1, 'INFO': 2, 'WARNING': 3}

decorate(*record*)

emit(*record*)

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

property is_tty

class contributor_ci.logger.**Logger**

Bases: object

cleanup()

debug(*msg*)

error(*msg*)

exit(*msg*, *return_code=1*)

handler(*msg*)

info(*msg*)

location(*msg*)

progress(*done=None*, *total=None*)

set_level(*level*)

set_stream_handler(*stream_handler*)

shellcmd(*msg*)

text_handler(*msg*)

The default snakemake log handler. Prints the output to the console. :param msg: the log message dictionary :type msg: dict

warning(*msg*)

contributor_ci.logger.**setup_logger**(*quiet=False*, *printshellcmds=False*, *nocolor=False*, *stdout=False*, *debug=False*, *use_threads=False*, *wms_monitor=None*)

contributor_ci.main module

contributor_ci.main.extractor module

contributor_ci.main.schemas module

contributor_ci.main.settings module

contributor_ci.main.extractors module

PYTHON MODULE INDEX

INDEX