benchmark-wrapper

Release 0.0.1

red-hat-performance

Mar 31, 2023

CONTENTS

benchmark-wrapper, aka, **Situation Normal: All F'ed Up (snafu)**, provides a convenient mechanism for launching, processing, and storing data produced by performance benchmarks. Traditionally, benchmark tools have presented users with the following challenges:

1. Ad hoc and/or raw standard output.
2. No method for preserving and exporting results for long term archive.
3. Difficulty at being platform agnostic.

benchmark-wrapper aims to solve these issues by providing a common, streamlined interface for running any performance benchmark required and exporting the results as needed.

The architecture is simple. Let's say that a user wants to run the totally-awesome benchmark, which runs on the CLI and has its own special output format. We start by creating a Python module (referred to as a wrapper), which can understand how to run totally-awesome, interpret its output, and transform the results into JSON-formatted events:

When we then run snafu, it begins by processing user input and ensuring that the specified export locations are up and running. For instance, a user can specify they wish to run the totally-awesome workload with 500 samples, 4 CPU cores, and export the results to an Elasticsearch instance running on localhost. snafu will ensure that all the required arguments for the totally-awesome workload are present and check that the ES instance on localhost is available. After pre-flight checks are complete, snafu will load the totally-awesome wrapper Python module, use it to perform the benchmark, collect the results and export.

A suite of benchmarks comes supported out-of-the-box.

CHAPTER

ONE

INSTALLATION

Installing benchmark-wrapper is all done through pip and git, requiring Python >= 3.6. For instance, to download benchmark-wrapper and install within a new virtual environment:

```
$ git clone https://github.com/cloud-bulldozer/benchmark-wrapper
$ cd benchmark-wrapper
$ python -m venv ./venv
$ pip install .
```

Or, if you want to just install benchmark-wrapper directly into your user site-packages:

```
$ pip install git+https://github.com/cloud-bulldozer/benchmark-wrapper
```

For reproducable installations, use the appropriate pip requirements files for your target version of Python. These requirement files are generating using pip-compile and include dependencies pinned to a specific version. Only after testing are these versions upgraded, preventing errors arising from dependency resolution. As an example, to install benchmark-wrapper for Python 3.6:

```
$ git clone https://github.com/cloud-bulldozer/benchmark-wrapper
$ cd benchmark-wrapper
$ pip install -r requirements/py36-reqs/install.txt
$ pip install .
```

A containerized version of benchmark-wrapper for each workload can also be built using the included Dockerfiles. Each workload is shipped with its own Dockerfile that packages benchmark-wrapper and any tools the benchmark needs to run. These are included under each benchmark's wrapper package within the source code:

```
$ cd benchmark-wrapper
$ find . -name Dockerfile
```

The build context for these Dockerfiles is the project root, so be sure to take this into consideration when building images. For instance, to build the Uperf benchmark container:

```
$ cd benchmark-wrapper
$ podman build . -t my-uperf-container -f snafu/benchmarks/uperf/Dockerfile
```

These images are automatically built on merges to our main branch and published to quay over at quay.io/cloud-bulldozer.

CHAPTER

TWO

EXAMPLES

CHAPTER

THREE

USAGE

CHAPTER

FOUR

WORKLOADS

4.1 uperf

4.2 fio

4.3 CoreMark-Pro

Wrapper for CoreMark-Pro which is a CPU benchmarking tool that provides a single number score for easy comparison across runs.

4.3.1 Overview of Operations

- A path to where CoreMark-Pro has been cloned is provided to benchmark-wrapper since there is no install mechanism.
- Executing the benchmark is done with `make` and also compiles the benchmark if not already done.
- A log folder is created from the exection, where only the `.log` and `.mark` file are processed:

```
coremark-pro/builds/linux64/gcc64/logs
├── linux64.gcc64.log       # Raw logs of the CoreMark Pro run
├── linux64.gcc64.mark      # Results: both individual workloads and overall score
├── progress.log            # Does not process
├── zip-test.run.log        # Does not process
└── zip-test.size.log       # Does not process
```

- The results are ingested into two different Elasticsarch indexes:
  - `*-coremark-pro-summary`: Results from the `.mark` file. Provides the calculated results from CoreMark-Pro.
  - `*-coremark-pro-raw`: Raw logs from the `.log` file. Intended for analyzing the logs manually.

4.3.2 Arguments

Required

- `-p` / `--path` Directory where CoreMark Pro is located.

Optional

- `-c` / `--context`: CoreMark Pro's context argument. Defaults to 1.
- `-w` / `--worker`: CoreMark Pro's worker argument. Defaults to 1.
- `-s` / `--sample`: Number of samples to run. Defaults to 1.
- `-r` / `--result-name`: The name of CoreMark Pro's result files. This includes the path relative to `--path` and does not include the extension. Defaults to `builds/linux64/gcc64/logs/linux64.gcc64`
- `-i` / `--ingest`: Parses and ingest existing results in a CoreMark-Pro log directory. No support for multiple samples and `date` is based on when benchmark-wrapper is run. Mainly used for debugging.

4.3.3 Running inside a container

The Dockerfile has CoreMark-Pro pre-built and is located at `/coremark-pro/`. This will need to be passed to benchmark-wrapper with the `--path` command.

Rest of this section will cover common use-cases that need additional parameters.

Archive file

To create an archive file and make it accessible to the host system, the `WORKDIR` is set to `/output/` and can be mounted to the host system. Example:

```
podman run -it \
-v ./FOLDER_TO_SAVE_ARCHIVE/:/output/ \
coremark-pro run_snafu -t coremark-pro --path /coremark-pro/ --create-archive \
--archive coremarkpro.archive
```

Raw logs

To retrieve the raw logs, mount the log folder from CoreMark-Pro to a **dedicated** folder on the host system otherwise all contents of the folder will be **deleted** when CoreMark-Pro is executed. Default folder is `/coremark-pro/builds/linux64/gcc/logs/`.

Example of logs folder being saved to a `output` folder in the current directory:

```
podman run -it \
-w /coremark-pro/builds/linux64/gcc64/logs/ \
-v ./output/:/coremark-pro/builds/linux64/gcc64/logs/ \
coremark-pro run_snafu -t coremark-pro -p /coremark-pro/
```

4.3.4 Parsing

This section gives a general idea of how CoreMark-Pro output matches with the Elasticsearch fields.

Results

These results are calculated by CoreMark-Pro and read from the `*.mark` file. Each row of the table is ingested as its own record.

Example `.mark` file

```
WORKLOAD RESULTS TABLE

                                                MultiCore SingleCore
Workload Name                                    (iter/s)   (iter/s)    Scaling
---------------------------------------------- ---------- ---------- ----------
cjpeg-rose7-preset                                178.57     192.31        0.93

.... truncated rest of the workloads ...

MARK RESULTS TABLE

Mark Name                                       MultiCore SingleCore    Scaling
---------------------------------------------- ---------- ---------- ----------
CoreMark-PRO                                      5708.35    5714.89        1.00
```

Benchmark-wrapper's archive file output

```
{
    "_source": {
        "test_config": {
            "worker": 1,                    # `--worker`
            "context": 1                    # `--context`
            "sample": 1,                    # `--sample`
        },
        "date": "2021-10.1..",              # Time when benchmark-wrapper was executed.
        "sample": 1,                        # `--sample`
        "name": "cjpeg-rose7-preset,        # Name of the CoreMark-Pro workload
        "multicore": 178.57,                # Multi Core result
        "singlecore": 192.31,               # Single Core result
        "scaling": 0.93,                    # Scaling result
        "type": "workload",                 # Type of result, determined by the table␣
↪header
                                            # - `workload`: Data from 'Workload Results␣
↪Table'
                                            # - `mark`: Data from 'Mark Results Table'
        "cluster_name": "laptop",
        "user": "ed",
        "uuid": "3cc2e4a9-bd7f-4394-8d8c-66415ceeb02f",
```

(continues on next page)

(continued from previous page)

```
        "workload": "coremark-pro",
        "run_id": "NA"
    },
}

... The above is repeated for the rest of the workloads and the mark result ...
```

Raw logs

These are the raw logs parsed from the `.log` file. The median results are dropped since they can be derived using Elasticsearch. Each row of results is ingested as its own record.

Excerpt of a log file

```
#UID            Suite Name                                    Ctx Wrk Fails       t(s) ␣
↪     Iter     Iter/s  Codesize   Datasize
#Results for verification run started at 21285:10:58:22 XCMD=-c1 -w0
236760500         MLT cjpeg-rose7-preset                        1   1     0      0.010 ␣
↪        1     100.00    105616     267544
#Results for performance runs started at 21285:10:58:23 XCMD=-c1 -w0
236760500         MLT cjpeg-rose7-preset                        1   1     0      0.081 ␣
↪       10     123.46    105616     267544
... truncated rest of the log ...
```

CoreMark-Pro performs two sets of runs for each workload that are marked by the same `uid`. Each set of runs has a single verification run and three performance runs. The number of runs is non-configurable. Structure of the runs:

```
Set 1: Context = 1 Workload = 1
├── Workload Verification Run
├── Performance run #1
├── Performance run #2
└── Performance run #3

... Repeat for all workloads ...

Set 2: Context and workload specified by user through -w / -c
├── Workload Verification Run
├── Performance run #1
├── Performance run #2
└── Performance run #3

... Repeat for all workloads ...
```

Benchmark-wrapper's archive file output

A `run_index` field was added to ensure performance runs with the same results are not marked as duplicates.

```
{
    "_source": {
        ## Same as the `.mark` file
        "test_config": {
            "worker": 1,,
            "context": 1,
            "sample": 1
        },
        "date": "2021-10.1..",        # Time when benchmark-wrapper was executed.
        "sample": 1,

        ## Results from the logs
        "uid": "236760500",           # A UID generated per workload by CoreMark-Pro
        "suite": "MLT",
        "name": "cjpeg-rose7-preset",
        "ctx": 1,
        "wrk": 1,
        "fails": 0,
        "t(s)": 0.01,
        "iter": 1,
        "iter/s": 100.0,
        "codesize": 105616,
        "datasize": 267544,
        "type": "verification",       # Possible types: verification / performance
        "starttime": "2021-10....",   # The start time for the runs as recorded by␣
↪CoreMark Pro
        "run_index": 0,               # An index of how many runs of the same type.␣
↪Always
                                      # 0 for verification, between 0-2 for performance␣
↪runs.

        ## Same as the `.mark` file
        "cluster_name": "laptop",
        "user": "ed",
        "uuid": "816f7fe9-ab04-45a4-8a1f-ce61c2fe11e6",
        "workload": "coremark-pro",
        "run_id": "NA"
    },
}
```

4.3.5 Limitations

- Limited ability to visualize the data from `*-coremark-pro-raw`, requires additional fields to aggregate the runs.

CHAPTER

FIVE

EXPORTING DATA

CHAPTER

SIX

DESIGN OVERVIEW

CHAPTER

SEVEN

CONTRIBUTING

7.1 Development Environment

7.2 Adding New Workloads

Adding new workloads into benchmark-wrapper is a fairly straightforward process, but requires a bit of work. This page tracks all the changes that a user needs to make when adding in a new benchmark.

---

benchmark-wrapper is currently undergoing a re-write, meaning a lot of change is happening pretty quickly. The information on this page is relevant to the new modifications which change the way in which benchmarks should be added. It may not be consistent to the way in which existing benchmarks are developed.

This page is written within a Jupyter Notebook, however running benchmark-wrapper from within a Jupyter Notebook is not tested nor supported.

7.2.1 Step Zero: Prep

A Benchmark within benchmark-wrapper is essentially just a Python module which handles setting up, running, parsing and tearing down a benchmark. To create our benchmark, we need to understand the following items:

1. What is the human-readable, camel-case-able string name for our benchmark?
2. What arguments does the benchmark wrapper need from the user?
3. Are there any setup tasks that our benchmark wrapper needs to perform?
4. How do we run our benchmark?
5. Are there any cleanup tasks that our benchmark wrapper needs to perform?
6. What data should the benchmark export?

In this example, we'll create a new benchmark wrapper that does a ping test against a list of given hostnames and IPs:

1. We'll call it `pingtest`
2. We need to know which hosts the user wants to ping and how many pings the user wants to perform.
3. We need to verify that the arguments that the user gave us are valid. We'll also create a temp file to show that the benchmark is running.
4. We can run our ping tests using the `ping` shell command.

5. We need to clean up our 'I-am-running' temp file.
6. For each pinged host, we want to output a single result detailing the result of the ping session (RTT information, packet loss %, IP resolution, errors).

7.2.2 Step One: Initialize

To begin, let's create the required files for our benchmark by creating a new Python package under `snafu/benchmarks`:

```
snafu/benchmarks/pingtest/
├── __init__.py
└── pingtest.py
```

Inside `pingtest.py`, we'll create our initial `Benchmark` subclass. Inside this subclass, there are a few class variables which we need to set:

1. `tool_name`: This is the camel-case name for our benchmark.
2. `args`: These are the arguments which our Benchmark will pull from the user through the CLI, OS environment, and/or from a configuration file (CLI is preferred over the OS environment, which is preferred over the configuration file). In the background, snafu uses configargparse to do the dirty-work, which is a helpful wrapper around Python's own argparse. The `args` class variable should be set to a tuple of `snafu.config.ConfigArguments`, which take in arguments just like `configargparse.ArgumentParser.add_argument`. If you have used argparse in the past, this should look super familiar.
3. `metadata`: This is an iterable of strings which represent the metadata that will be exported with the Benchmark's results. Each string corresponds to the attribute name that the argument is stored under by `configargparse`. For instance, creating a new argument under the `args` class variable with `"--my-metadata", dest="mmd"` would result in the attribute name being `mmd`. Adding `mmd` under `metadata` will in turn cause the value for the `--my-metadata` argument to be exported as metadata. Benchmarks will by default specify `cluster_name`, `user` and `uuid` arguments as metadata, but if you want to use your own set of metadata keys it can be set here.

[1]:
```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Ping hosts and export results."""
from snafu.config import ConfigArgument
from snafu.benchmarks import Benchmark


class PingTest(Benchmark):
    """Wrapper for the Ping Test benchmark."""

    tool_name = "pingtest"
    args = (
        ConfigArgument(
            "--host",
            help="Host(s) to ping. Can give more than one host by separating them with "
                 "spaces on the CLI and in config files, or by giving them in a "
                 "pythonic-list format through the OS environment",
            dest="host",
            nargs="+",
            env_var="HOST",
            type=str,
            required=True,
        ),
```

(continues on next page)

(continued from previous page)

```
        ConfigArgument(
            "--count",
            help="Number of pings to perform per sample.",
            dest="count",
            env_var="COUNT",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--samples",
            help="Number of samples to perform.",
            dest="samples",
            env_var="SAMPLES",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--htlhcdtwy",
            help="Has The Large Hadron Collider Destroyed The World Yet?",
            dest="htlhcdtwy",
            env_var="HTLHCDTWY",
            default="no",
            type=str,
            choices=["yes", "no"]
        ),
    )
    # don't care about Cluster Name, but the Hadron Collider is serious business
    metadata = ("user", "uuid", "htlhcdtwy")

    def setup(self):
        """Setup the Ping Test Benchmark."""
        pass

    def collect(self):
        """Run the Ping Test Benchmark and collect results."""
        pass

    def cleanup(self):
        """Cleanup the Ping Test Benchmark."""
        pass
```

Let's check that we're ready to move on by trying to parse some configuration parameters. Let's load up Python!

benchmark-wrapper includes a special variable called `snafu.registry.TOOLS` which will map a benchmark's camel-case string name to its wrapper class. Let's use this to create an instance of our benchmark and parse some configuration.

[2]:
```
from snafu.registry import TOOLS
from pprint import pprint
pingtest = TOOLS["pingtest"]()

# Set some config parameters
# Config file
```

(continues on next page)

(continued from previous page)

```
!echo "samples: 3" > my_config.yaml
!echo "count: 5" >> my_config.yaml
# OS ENV
import os
os.environ["HOST"] = "[www.google.com,www.bing.com]"

# Parse arguments and print result
# Since we aren't running within the main script (run_snafu.py),
# need to add the config option manually
pingtest.config.parser.add_argument("--config", is_config_file=True)
pingtest.config.parse_args(
    "--config my_config.yaml --labels=notebook=true --uuid 1337 --user snafu "
    "--htlhcdtwy=no".split(" ")
)
pprint(vars(pingtest.config.params))

del pingtest
!rm my_config.yaml
```

```
{'cluster_name': None,
 'config': 'my_config.yaml',
 'count': 5,
 'host': ['www.google.com', 'www.bing.com'],
 'htlhcdtwy': 'no',
 'labels': {'notebook': 'true'},
 'samples': 3,
 'user': 'snafu',
 'uuid': '1337'}
```

Now that we have our configuration all ready to go, let's start filling in our benchmark.

7.2.3 Step Two: Setup Method

Each benchmark is expected to have a `setup` method, which will return `True` if setup tasks completed successfully and otherwise return `False`.

For our use case, let's write a file to `/tmp` that can signal other programs that our benchmark is running. We'll also check if our temporary file exists before writing it, which would indicate that something is wrong.

`Remeber, we are still working in our module found at snafu/benchmarks/pingtest.py`

---

[3]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Ping hosts and export results."""
from snafu.config import ConfigArgument
from snafu.benchmarks import Benchmark

# We'll also import this helpful function from the config module
import os
```

(continues on next page)

(continued from previous page)

```
from snafu.config import check_file


class PingTest(Benchmark):
    """Wrapper for the Ping Test benchmark."""

    tool_name = "pingtest"
    args = (
        ConfigArgument(
            "--host",
            help="Host(s) to ping. Can give more than one host by separating them with "
                 "spaces on the CLI and in config files, or by giving them in a "
                 "pythonic-list format through the OS environment",
            dest="host",
            nargs="+",
            env_var="HOST",
            type=str,
            required=True,
        ),
        ConfigArgument(
            "--count",
            help="Number of pings to perform per sample.",
            dest="count",
            env_var="COUNT",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--samples",
            help="Number of samples to perform.",
            dest="samples",
            env_var="SAMPLES",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--htlhcdtwy",
            help="Has The Large Hadron Collider Destroyed The World Yet?",
            dest="htlhcdtwy",
            env_var="HTLHCDTWY",
            default="no",
            type=str,
            choices=["yes", "no"]
        ),
    )
    # don't care about Cluster Name, but the Hadron Collider is serious business
    metadata = ("user", "uuid", "htlhcdtwy")

    TMP_FILE_PATH = "/tmp/snafu-pingtest"

    def setup(self) -> bool:
        """
```

(continues on next page)

(continued from previous page)

```
        Setup the Ping Test Benchmark.

        This method creates a temporary file at ``/tmp/snafu-pingtest`` to let others
        know that the benchmark is currently running.

        Returns
        -------
        bool
            True if the temporary file was created successfully, othewise False. Will
            also return False if the temporary file already exists.
        """

        if check_file(self.TMP_FILE_PATH):
            # The benchmark base class exposes a logger at self.logger which we can use
            self.logger.critical(
                f"Temporary file located at {self.TMP_FILE_PATH} already exists."
            )
            return False

        try:
            tmp_file = open(self.TMP_FILE_PATH, "x")
            tmp_file.close()
        except Exception as e:
            self.logger.critical(
                f"Unable to create temporary file at {self.TMP_FILE_PATH}: {e}",
                exc_info=True
            )

            return False
        else:
            self.logger.info(
                f"Successfully created temp file at {self.TMP_FILE_PATH}"
            )
            return True

    def collect(self):
        """Run the Ping Test Benchmark and collect results."""
        pass

    def cleanup(self):
        """Cleanup the Ping Test Benchmark."""
        pass
```

Let's test it out and make sure our setup method works properly:

[4]:
```
from snafu.registry import TOOLS
pingtest = TOOLS["pingtest"]()

!rm -f /tmp/snafu-pingtest

# No file exists
```

(continues on next page)

(continued from previous page)

```
print(f"Setup result is: {pingtest.setup()}")

# File exists
print(f"Setup result is: {pingtest.setup()}")

# Create failure in open
!rm -f /tmp/snafu-pingtest
open_bak = open
open = lambda file, mode: int("I'm a string")
print(f"Setup result is: {pingtest.setup()}")

# Cleanup
open = open_bak
!rm -f /tmp/snafu-pingtest
del pingtest
```

```
Temporary file located at /tmp/snafu-pingtest already exists.
```

```
Setup result is: True
Setup result is: False
```

```
Unable to create temporary file at /tmp/snafu-pingtest: invalid literal for int() with␣
↪base 10: "I'm a string"
Traceback (most recent call last):
  File "/tmp/ipykernel_687/1395185701.py", line 81, in setup
    tmp_file = open(self.TMP_FILE_PATH, "x")
  File "/tmp/ipykernel_687/3683524747.py", line 15, in <lambda>
    open = lambda file, mode: int("I'm a string")
ValueError: invalid literal for int() with base 10: "I'm a string"
```

```
Setup result is: False
```

7.2.4 Step Three: Cleanup Method

Now let's go ahead and populate our cleanup method. The `cleanup` method has the same usage as the setup method: return `True` if the cleanup was successfull, otherwise `False`. For the ping test benchmark, we just need to remove our temporary file:

[5]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Ping hosts and export results."""
import os
from snafu.config import ConfigArgument, check_file
from snafu.benchmarks import Benchmark


class PingTest(Benchmark):
    """Wrapper for the Ping Test benchmark."""

    tool_name = "pingtest"
    args = (
        ConfigArgument(
            "--host",
```

(continues on next page)

(continued from previous page)

```
            help="Host(s) to ping. Can give more than one host by separating them with "
                 "spaces on the CLI and in config files, or by giving them in a "
                 "pythonic-list format through the OS environment",
            dest="host",
            nargs="+",
            env_var="HOST",
            type=str,
            required=True,
        ),
        ConfigArgument(
            "--count",
            help="Number of pings to perform per sample.",
            dest="count",
            env_var="COUNT",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--samples",
            help="Number of samples to perform.",
            dest="samples",
            env_var="SAMPLES",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--htlhcdtwy",
            help="Has The Large Hadron Collider Destroyed The World Yet?",
            dest="htlhcdtwy",
            env_var="HTLHCDTWY",
            default="no",
            type=str,
            choices=["yes", "no"]
        ),
    )
    # don't care about Cluster Name, but the Hadron Collider is serious business
    metadata = ("user", "uuid", "htlhcdtwy")

    TMP_FILE_PATH = "/tmp/snafu-pingtest"

    def setup(self) -> bool:
        """
        Setup the Ping Test Benchmark.

        This method creates a temporary file at ``/tmp/snafu-pingtest`` to let others
        know that the benchmark is currently running.

        Returns
        -------
        bool
            True if the temporary file was created successfully, othewise False. Will
            also return False if the temporary file already exists.
```

(continues on next page)

(continued from previous page)

```
        """


        if check_file(self.TMP_FILE_PATH):
            # The benchmark base class exposes a logger at self.logger which we can use
            self.logger.critical(
                f"Temporary file located at {self.TMP_FILE_PATH} already exists."
            )
            return False

        try:
            tmp_file = open(self.TMP_FILE_PATH, "x")
            tmp_file.close()
        except Exception as e:
            self.logger.critical(
                f"Unable to create temporary file at {self.TMP_FILE_PATH}: {e}",
                exc_info=True
            )

            return False
        else:
            self.logger.info(
                f"Successfully created temp file at {self.TMP_FILE_PATH}"
            )
            return True

    def collect(self):
        """Run the Ping Test Benchmark and collect results."""
        pass

    def cleanup(self) -> bool:
        """
        Cleanup the Ping Test Benchmark.

        This method removes the temporary file at ``/tmp/snafu-pingtest`` to let others
        know that the benchmark has finished running.

        Returns
        -------
        bool
            True if the temporary file was deleted successfully, otherwise False.
        """

        try:
            os.remove(self.TMP_FILE_PATH)
        except Exception as e:
            self.logger.critical(
                f"Unable to remove temporary file at {self.TMP_FILE_PATH}: {e}",
                exc_info=True
            )

            return False
```

(continues on next page)

(continued from previous page)

```
        else:
            self.logger.info(
                f"Successfully removed temp file at {self.TMP_FILE_PATH}"
            )
            return True
```

And again, some quick tests just to verify it works as expected:

[6]:
```
from snafu.registry import TOOLS
pingtest = TOOLS["pingtest"]()

!rm -f /tmp/snafu-pingtest

# No file exists, so should error
print(f"Cleanup result is {pingtest.cleanup()}")

# Create the file using setup(), then cleanup()
print(f"Setup result is {pingtest.setup()}")
print(f"Cleanup result is {pingtest.cleanup()}")

# Cleanup
del pingtest
```

```
Unable to remove temporary file at /tmp/snafu-pingtest: [Errno 2] No such file or␣
↪directory: '/tmp/snafu-pingtest'
Traceback (most recent call last):
  File "/tmp/ipykernel_687/1198626359.py", line 112, in cleanup
    os.remove(self.TMP_FILE_PATH)
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/snafu-pingtest'
```

```
Cleanup result is False
Setup result is True
Cleanup result is True
```

Now we have our setup and cleanup methods good to go, let's get to the fun part.

7.2.5 Part Four: Collect Method

The collect method is an iterable that returns a special dataclass that is shipped with benchmark-wrapper, called a `BenchmarkResult`. BenchmarkResult holds important information about a benchmark's resulting data, such as the configuration, metadata, labels and numerical data. It also understands how to prepare itself for export. All Benchmarks are expected to return their results using this dataclass in order to support a common interface for data exporters, reduce code reuse, and reduce extra overhead.

The base Benchmark class includes a helpful method called `create_new_result`, which we will use in the example below.

For our ping test benchmark, our collect method needs to run the ping command, parse its output, and yield a new BenchmarkResult. To help prevent the collect method itself from becoming super large and out of control, we'll create some new methods in our wrapper class that the collect method will call to help do its thing. Benchmarks can have any number of additional methods, just as long as they have setup, collect and cleanup.

One last note here before the code: benchmark-wrapper ships with another helpful module called `process`, which contains functions and classes to facilitate running subprocesses. In particular, we'll be using the `get_process_sample`

function.

[7]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Ping hosts and export results."""
import os
from snafu.config import ConfigArgument, check_file
from snafu.benchmarks import Benchmark

# Grab the get_process_sample function, the BenchmarkResult class, stuff
# for type hints, and dataclasses for storing our ping results
from snafu.process import get_process_sample, ProcessSample, ProcessRun
from snafu.benchmarks import BenchmarkResult
from typing import Iterable, Optional
from dataclasses import dataclass, asdict
# Also shlex and subprocess for creating our ping command
import shlex
import subprocess
# And finally re for regex
import re


@dataclass
class PingResult:
    ip: Optional[str] = None
    success: Optional[bool] = None
    fail_msg: Optional[str] = None
    host: Optional[str] = None
    transmitted: Optional[int] = None
    received: Optional[int] = None
    packet_loss: Optional[float] = None
    packet_bytes: Optional[int] = None
    time_ms: Optional[float] = None
    rtt_min_ms: Optional[float] = None
    rtt_avg_ms: Optional[float] = None
    rtt_max_ms: Optional[float] = None
    rtt_mdev_ms: Optional[float] = None


class PingTest(Benchmark):
    """Wrapper for the Ping Test benchmark."""

    tool_name = "pingtest"
    args = (
        ConfigArgument(
            "--host",
            help="Host(s) to ping. Can give more than one host by separating them with "
                 "spaces on the CLI and in config files, or by giving them in a "
                 "pythonic-list format through the OS environment",
            dest="host",
            nargs="+",
            env_var="HOST",
            type=str,
```

(continues on next page)

(continued from previous page)

```
            required=True,
        ),
        ConfigArgument(
            "--count",
            help="Number of pings to perform per sample.",
            dest="count",
            env_var="COUNT",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--samples",
            help="Number of samples to perform.",
            dest="samples",
            env_var="SAMPLES",
            default=1,
            type=int,
        ),
        ConfigArgument(
            "--htlhcdtwy",
            help="Has The Large Hadron Collider Destroyed The World Yet?",
            dest="htlhcdtwy",
            env_var="HTLHCDTWY",
            default="no",
            type=str,
            choices=["yes", "no"]
        ),
    )
    # don't care about Cluster Name, but the Hadron Collider is serious business
    metadata = ("user", "uuid", "htlhcdtwy")

    TMP_FILE_PATH = "/tmp/snafu-pingtest"

    HOST_RE = r"PING ([a-zA-Z0-9.-]+) \(([0-9.]+)\) \d+\(([\d]+)\) bytes of data\."
    RTT_STATS_RE = r"rtt min\/avg\/max\/mdev = ([\d.]+)\/([\d.]+)\/([\d.]+)\/([\d.]+) ms"
    PACKET_RE = r"(\d+) packets transmitted, (\d+) received, ([\d.]+)\% packet loss, " \
                r"time (\d+)ms"

    def setup(self) -> bool:
        """
        Setup the Ping Test Benchmark.

        This method creates a temporary file at ``/tmp/snafu-pingtest`` to let others
        know that the benchmark is currently running.

        Returns
        -------
        bool
            True if the temporary file was created successfully, othewise False. Will
            also return False if the temporary file already exists.
        """
```

(continues on next page)

(continued from previous page)

```
        if check_file(self.TMP_FILE_PATH):
            # The benchmark base class exposes a logger at self.logger which we can use
            self.logger.critical(
                f"Temporary file located at {self.TMP_FILE_PATH} already exists."
            )
            return False

        try:
            tmp_file = open(self.TMP_FILE_PATH, "x")
            tmp_file.close()
        except Exception as e:
            self.logger.critical(
                f"Unable to create temporary file at {self.TMP_FILE_PATH}: {e}",
                exc_info=True
            )

            return False
        else:
            self.logger.info(
                f"Successfully created temp file at {self.TMP_FILE_PATH}"
            )
            return True

    def parse_host_info(self, stdout: str, store: PingResult) -> None:
        """
        Parse the host line of ping stdout.

        Expected format is: ``PING host (ip) data_bytes(ICMP_data_bytes) ...``.

        Parameters
        ----------
        stdout : str
            ping stdout to parse
        store : PingResult
            PingResult instance to store parsed variables into
        """

        result = re.compile(self.HOST_RE).search(stdout)
        if result is None:
            self.logger.warning(f"Unable to parse host info!")
            return
        host, ip, data_size = result.groups()
        data_size = int(data_size)

        if host == ip:
            host = None  # user pinged an IP rather than a host

        store.host = host
        store.ip = ip
        store.packet_bytes = data_size
```

(continues on next page)

(continued from previous page)

```
    def parse_packet_stats(self, stdout: str, store: PingResult) -> None:
        """
        Parse the packet statistics line of ping stdout.

        Expected format is:
        ``A packets transmitted, B received, C% packet loss, time Dms``

        Parameters
        ----------
        stdout : str
            ping stdout to parse
        store : PingResult
            PingResult instance to store parsed variables into
        """

        result = re.compile(self.PACKET_RE).search(stdout)
        if result is None:
            self.logger.warning(
                f"Unable to parse packet stats!"
            )
            return
        transmitted, received, packet_loss, time_ms = map(int, result.groups())

        store.transmitted = transmitted
        store.received = received
        store.packet_loss = packet_loss
        store.time_ms = time_ms

    def parse_rtt_stats(self, stdout: str, store: PingResult) -> None:
        """
        Parse the RTT statistics line of ping stdout.

        Expected format is: ``rtt min/avg/max/mdev = A/B/C/D ms``

        Parameters
        ----------
        stdout : str
            ping stdout to parse
        store : PingResult
            PingResult instance to store parsed variables into
        """

        result = re.compile(self.RTT_STATS_RE).search(stdout)
        if result is None:
            self.logger.warning(f"Unable to parse rtt stats!")
            return
        rtt_min, rtt_avg, rtt_max, rtt_mdev = map(float, result.groups())
        store.rtt_min_ms = rtt_min
        store.rtt_avg_ms = rtt_avg
        store.rtt_max_ms = rtt_max
        store.rtt_mdev_ms = rtt_mdev
```

(continues on next page)

(continued from previous page)

```
    def parse_stdout(self, stdout: str) -> PingResult:
        """
        Parse the stdout of the ping command.

        Tested against ping from iputils 20210202 on Fedora Linux 34

        Returns
        -------
        PingResult
        """

        # We really only care about the first line, and the last two lines
        lines = stdout.strip().split("\n")

        # Check if we got an error
        if len(lines) == 1:
            msg = lines[0]
            return PingResult(
                fail_msg=msg,
                success=False
            )


        result = PingResult(success=True)
        self.parse_host_info(stdout, result)
        self.parse_packet_stats(stdout, result)
        self.parse_rtt_stats(stdout, result)

        return result

    def ping_host(self, host: str) -> Iterable[BenchmarkResult]:
        """
        Run the ping test benchmark against the given host.

        Parameters
        ----------
        host : str
            Host to ping

        Returns
        -------
        iterable
            Iterable of BenchmarkResults
        """

        self.logger.info(f"Running ping test against host {host}")
        cmd = shlex.split(f"ping -c {self.config.count} {host}")
        self.logger.debug(f"Using command: {cmd}")

        # A config instance allows for accessing params directly,
        # therefore self.config.samples == self.config.params.samples
        for sample_num in range(self.config.samples):
```

(continues on next page)

(continued from previous page)

```
            self.logger.info(f"Collecting sample {sample_num}")

            # We'll use the LiveProcess context manager to run ping
            # LiveProcess will expose the Popen object at 'process',
            # create a queue with lines from stdout at 'stdout',
            # and create a snafu.process.ProcessRun instance at `attempt`

            # Here we will tell get_process_sample to send stdout and stderr
            # to the same pipe
            process_sample: ProcessSample = get_process_sample(
                cmd, self.logger, stdout=subprocess.PIPE, stderr=subprocess.STDOUT
            )

            self.logger.debug(f"Got process sample: {vars(process_sample)}")
            if not process_sample.success:
                self.logger.warning(f"Process was unsuccessful")
                process_run: ProcessRun = process_sample.failed[0]
            else:
                self.logger.info(f"Process was successful!")
                process_run: ProcessRun = process_sample.successful

            result: PingResult = self.parse_stdout(process_run.stdout)
            # manually set host if we fail, since it won't always be parsable
            # through stdout
            if result.success is False:
                result.host = host
            self.logger.info(f"Got sample: {vars(result)}")

            yield self.create_new_result(
                # We use vars here because create_new_result expects
                # dict objects, not dataclasses
                data=vars(result),
                config={"samples": self.config.samples, "count": self.config.count},
                # tag is a method for labeling results for exporters
                # right now it specifies the ES index to export to
                tag="jupyter"
            )

        plural = "s" if self.config.samples > 1 else ""
        self.logger.info(
            f"Finised collecting {self.config.samples} sample{plural} against {host}"
        )

    def collect(self) -> Iterable[BenchmarkResult]:
        """
        Run the Ping Test Benchmark and collect results.
        """

        self.logger.info("Running pings and collecting results.")
        self.logger.debug(f"Using config: {vars(self.config.params)}")
        if isinstance(self.config.host, str):
            yield from self.ping_host(self.config.host)
```

(continues on next page)

(continued from previous page)

```
        else:
            for host in self.config.host:
                yield from self.ping_host(host)
        self.logger.info("Finished")

    def cleanup(self) -> bool:
        """
        Cleanup the Ping Test Benchmark.

        This method removes the temporary file at ``/tmp/snafu-pingtest`` to let others
        know that the benchmark has finished running.

        Returns
        -------
        bool
            True if the temporary file was deleted successfully, otherwise False.
        """

        try:
            os.remove(self.TMP_FILE_PATH)
        except Exception as e:
            self.logger.critical(
                f"Unable to remove temporary file at {self.TMP_FILE_PATH}: {e}",
                exc_info=True
            )

            return False
        else:
            self.logger.info(
                f"Successfully removed temp file at {self.TMP_FILE_PATH}"
            )
            return True
```

We have finished our new ping test benchmark! Let's try it out! We'll ping three hosts:

- `www.google.com`: Depending on the build environment, this domain will show either 100% success (ICMP enabled) or 100% failure (ICMP disabled).
- `www.idontexist.heythere`: A domain name which doesn't exist. Ping should exit with a failure saying that the host couldn't be resolved
- `localhost`: We'll ping localhost, as we know regardless of environment we'll be able to ping ourselves.

[8]:
```
from snafu.registry import TOOLS
from pprint import pprint
import logging
pingtest = TOOLS["pingtest"]()

# All Benchmark loggers work under the "snafu" logger
logger = logging.getLogger("snafu")
if not logger.hasHandlers():
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.DEBUG)
```

(continues on next page)

(continued from previous page)

```
!rm -rf /tmp/snafu-pingtest
!rm -f my_config.yaml

# Set some config parameters
# Config file
!echo "samples: 1" > my_config.yaml
!echo "count: 5" >> my_config.yaml
# OS ENV
import os
os.environ["HOST"] = "[www.google.com,www.idontexist.heythere,localhost]"

# Parse arguments and print result
# Since we aren't running within the main script (run_snafu.py),
# need to add the config option manually
pingtest.config.parser.add_argument("--config", is_config_file=True)
pingtest.config.parse_args(
    "--config my_config.yaml --labels=notebook=true --uuid 1337 --user snafu "
    "--htlhcdtwy=no".split(" ")
)

# The base benchmark class includes a run method that runs setup -> collect -> cleanup
results = list(pingtest.run())

!rm -rf /tmp/snafu-pingtest
!rm -f my_config.yaml
```

```
Starting pingtest wrapper.
Running setup tasks.
Successfully created temp file at /tmp/snafu-pingtest
Collecting results from benchmark.
Running pings and collecting results.
Using config: {'config': 'my_config.yaml', 'labels': {'notebook': 'true'}, 'cluster_name
↪': None, 'user': 'snafu', 'uuid': '1337', 'host': ['www.google.com', 'www.idontexist.
↪heythere', 'localhost'], 'count': 5, 'samples': 1, 'htlhcdtwy': 'no'}
Running ping test against host www.google.com
Using command: ['ping', '-c', '5', 'www.google.com']
Collecting sample 0
Running command: ['ping', '-c', '5', 'www.google.com']
Using args: {'stdout': -1, 'stderr': -2}
On try 1
Finished running. Got attempt: ProcessRun(rc=0, stdout='PING www.google.com (142.250.190.
↪36) 56(84) bytes of data.\n64 bytes from ord37s33-in-f4.1e100.net (142.250.190.36):␣
↪icmp_seq=1 ttl=45 time=16.4 ms\n64 bytes from ord37s33-in-f4.1e100.net (142.250.190.
↪36): icmp_seq=2 ttl=45 time=16.4 ms\n64 bytes from ord37s33-in-f4.1e100.net (142.250.
↪190.36): icmp_seq=3 ttl=45 time=16.5 ms\n64 bytes from ord37s33-in-f4.1e100.net (142.
↪250.190.36): icmp_seq=4 ttl=45 time=16.4 ms\n64 bytes from ord37s33-in-f4.1e100.net␣
↪(142.250.190.36): icmp_seq=5 ttl=45 time=16.5 ms\n\n--- www.google.com ping statistics␣
↪---\n5 packets transmitted, 5 received, 0% packet loss, time 4005ms\nrtt min/avg/max/
↪mdev = 16.459/16.493/16.522/0.142 ms\n', stderr=None, time_seconds=4.029713, hit_
↪timeout=False)
Got return code 0, expected 0
```

(continues on next page)

(continued from previous page)

```
Command finished with 1 attempt: ['ping', '-c', '5', 'www.google.com']
Got process sample: {'expected_rc': 0, 'success': True, 'attempts': 1, 'timeout': None,
↪'failed': [], 'successful': ProcessRun(rc=0, stdout='PING www.google.com (142.250.190.
↪36) 56(84) bytes of data.\n64 bytes from ord37s33-in-f4.1e100.net (142.250.190.36):␣
↪icmp_seq=1 ttl=45 time=16.4 ms\n64 bytes from ord37s33-in-f4.1e100.net (142.250.190.
↪36): icmp_seq=2 ttl=45 time=16.4 ms\n64 bytes from ord37s33-in-f4.1e100.net (142.250.
↪190.36): icmp_seq=3 ttl=45 time=16.5 ms\n64 bytes from ord37s33-in-f4.1e100.net (142.
↪250.190.36): icmp_seq=4 ttl=45 time=16.4 ms\n64 bytes from ord37s33-in-f4.1e100.net␣
↪(142.250.190.36): icmp_seq=5 ttl=45 time=16.5 ms\n\n--- www.google.com ping statistics␣
↪---\n5 packets transmitted, 5 received, 0% packet loss, time 4005ms\nrtt min/avg/max/
↪mdev = 16.459/16.493/16.522/0.142 ms\n', stderr=None, time_seconds=4.029713, hit_
↪timeout=False)}
Process was successful!
Got sample: {'ip': '142.250.190.36', 'success': True, 'fail_msg': None, 'host': 'www.
↪google.com', 'transmitted': 5, 'received': 5, 'packet_loss': 0, 'packet_bytes': 84,
↪'time_ms': 4005, 'rtt_min_ms': 16.459, 'rtt_avg_ms': 16.493, 'rtt_max_ms': 16.522,
↪'rtt_mdev_ms': 0.142}
Finised collecting 1 sample against www.google.com
Running ping test against host www.idontexist.heythere
Using command: ['ping', '-c', '5', 'www.idontexist.heythere']
Collecting sample 0
Running command: ['ping', '-c', '5', 'www.idontexist.heythere']
Using args: {'stdout': -1, 'stderr': -2}
On try 1
Finished running. Got attempt: ProcessRun(rc=2, stdout='ping: www.idontexist.heythere:␣
↪Name or service not known\n', stderr=None, time_seconds=0.010389, hit_timeout=False)
Got return code 2, expected 0
Got bad return code from command: ['ping', '-c', '5', 'www.idontexist.heythere'].
After 1 attempts, unable to run command: ['ping', '-c', '5', 'www.idontexist.heythere']
Got process sample: {'expected_rc': 0, 'success': False, 'attempts': 1, 'timeout': None,
↪'failed': [ProcessRun(rc=2, stdout='ping: www.idontexist.heythere: Name or service not␣
↪known\n', stderr=None, time_seconds=0.010389, hit_timeout=False)], 'successful': None}
Process was unsuccessful
Got sample: {'ip': None, 'success': False, 'fail_msg': 'ping: www.idontexist.heythere:␣
↪Name or service not known', 'host': 'www.idontexist.heythere', 'transmitted': None,
↪'received': None, 'packet_loss': None, 'packet_bytes': None, 'time_ms': None, 'rtt_min_
↪ms': None, 'rtt_avg_ms': None, 'rtt_max_ms': None, 'rtt_mdev_ms': None}
Finised collecting 1 sample against www.idontexist.heythere
Running ping test against host localhost
Using command: ['ping', '-c', '5', 'localhost']
Collecting sample 0
Running command: ['ping', '-c', '5', 'localhost']
Using args: {'stdout': -1, 'stderr': -2}
On try 1
Finished running. Got attempt: ProcessRun(rc=0, stdout='PING localhost (127.0.0.1)␣
↪56(84) bytes of data.\n64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.
↪012 ms\n64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.027 ms\n64 bytes␣
↪from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.033 ms\n64 bytes from localhost␣
↪(127.0.0.1): icmp_seq=4 ttl=64 time=0.028 ms\n64 bytes from localhost (127.0.0.1):␣
↪icmp_seq=5 ttl=64 time=0.028 ms\n\n--- localhost ping statistics ---\n5 packets␣
↪transmitted, 5 received, 0% packet loss, time 4086ms\nrtt min/avg/max/mdev = 0.012/0.
↪025/0.033/0.009 ms\n', stderr=None, time_seconds=4.096055, hit_timeout=False)
```

(continues on next page)

(continued from previous page)

```
Got return code 0, expected 0
Command finished with 1 attempt: ['ping', '-c', '5', 'localhost']
Got process sample: {'expected_rc': 0, 'success': True, 'attempts': 1, 'timeout': None,
↪'failed': [], 'successful': ProcessRun(rc=0, stdout='PING localhost (127.0.0.1) 56(84)␣
↪bytes of data.\n64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.012 ms\
↪n64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.027 ms\n64 bytes from␣
↪localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.033 ms\n64 bytes from localhost (127.0.
↪0.1): icmp_seq=4 ttl=64 time=0.028 ms\n64 bytes from localhost (127.0.0.1): icmp_seq=5␣
↪ttl=64 time=0.028 ms\n\n--- localhost ping statistics ---\n5 packets transmitted, 5␣
↪received, 0% packet loss, time 4086ms\nrtt min/avg/max/mdev = 0.012/0.025/0.033/0.009␣
↪ms\n', stderr=None, time_seconds=4.096055, hit_timeout=False)}
Process was successful!
Got sample: {'ip': '127.0.0.1', 'success': True, 'fail_msg': None, 'host': 'localhost',
↪'transmitted': 5, 'received': 5, 'packet_loss': 0, 'packet_bytes': 84, 'time_ms': 4086,
↪ 'rtt_min_ms': 0.012, 'rtt_avg_ms': 0.025, 'rtt_max_ms': 0.033, 'rtt_mdev_ms': 0.009}
Finised collecting 1 sample against localhost
Finished
Cleaning up
Successfully removed temp file at /tmp/snafu-pingtest
```

[9]:
```
print(f"Got {len(results)} results")
for result in results[:5]:
    pprint(vars(result))
```

```
Got 3 results
{'config': {'count': 5, 'samples': 1},
 'data': {'fail_msg': None,
          'host': 'www.google.com',
          'ip': '142.250.190.36',
          'packet_bytes': 84,
          'packet_loss': 0,
          'received': 5,
          'rtt_avg_ms': 16.493,
          'rtt_max_ms': 16.522,
          'rtt_mdev_ms': 0.142,
          'rtt_min_ms': 16.459,
          'success': True,
          'time_ms': 4005,
          'transmitted': 5},
 'labels': {'notebook': 'true'},
 'metadata': {'htlhcdtwy': 'no', 'user': 'snafu', 'uuid': '1337'},
 'name': 'pingtest',
 'tag': 'jupyter'}
{'config': {'count': 5, 'samples': 1},
 'data': {'fail_msg': 'ping: www.idontexist.heythere: Name or service not '
                      'known',
          'host': 'www.idontexist.heythere',
          'ip': None,
          'packet_bytes': None,
          'packet_loss': None,
          'received': None,
          'rtt_avg_ms': None,
```

(continues on next page)

(continued from previous page)

```
          'rtt_max_ms': None,
          'rtt_mdev_ms': None,
          'rtt_min_ms': None,
          'success': False,
          'time_ms': None,
          'transmitted': None},
 'labels': {'notebook': 'true'},
 'metadata': {'htlhcdtwy': 'no', 'user': 'snafu', 'uuid': '1337'},
 'name': 'pingtest',
 'tag': 'jupyter'}
{'config': {'count': 5, 'samples': 1},
 'data': {'fail_msg': None,
          'host': 'localhost',
          'ip': '127.0.0.1',
          'packet_bytes': 84,
          'packet_loss': 0,
          'received': 5,
          'rtt_avg_ms': 0.025,
          'rtt_max_ms': 0.033,
          'rtt_mdev_ms': 0.009,
          'rtt_min_ms': 0.012,
          'success': True,
          'time_ms': 4086,
          'transmitted': 5},
 'labels': {'notebook': 'true'},
 'metadata': {'htlhcdtwy': 'no', 'user': 'snafu', 'uuid': '1337'},
 'name': 'pingtest',
 'tag': 'jupyter'}
```

And that's that! As soon as you have your benchmark working that way you'd like, submit a PR and we'll give it a LGTM.

7.3 Adding New Exports

7.4 Testing

benchmark-wrapper uses tox and pytest for unit testing as well as documentation build testing.

As a quick reminder, unit testing is defined as follows by guru99.com:

> UNIT TESTING is a type of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected.

The goal for unit testing within benchmark-wrapper specifically is to ensure that all shared units (common modules and functionality) behave as expected, in order to create a solid foundation that all benchmarks may be based upon.

For documentation build testing, the goal is to ensure that the documentation can build without errors and that there are no broken external links.

Here are the main takeaways:

- To write unit tests for a module, place them in `tests/unit/test_<module>.py` or within docstrings for small, basic functions.

- Use Tox to invoke pytest unit tests and documentation build tests, by choosing from the following environments: `py{36,37,38,39}-{unit,docs}`.
- Use coverage reports to ensure thorough code testing.

7.4.1 Tox Usage

There are eight distinct environments that Tox is configured for (through the `tox.ini` file within the project root), which is a permutation across four versions of Python and our two testing goals. These environments can be invoked by picking values from the following sets:

```
tox -e py{36,37,38,39}-{unit,docs}
```

For instance, using `py36-unit` will run unit tests for Python 3.6, while `py38-docs` will run a documentation build test for Python 3.8. To ensure reproducability across test runs, dependencies are installed using the included pip requirements files named `tests.txt` for each Python version. These requirements files are built on top of the `install.txt`requirements files and include extra packages needed for testing.

7.4.2 Writing Unit Tests

Unit tests are placed under the `tests/unit` directory, container within individual Python modules. To keep a consistent structure, each module under this directory should correspond to one module within benchmark-wrapper. As an example, to create unit tests for `snafu.module`, place them within `tests/unit/test_module.py`.

Tests can also live within docstrings, which will be invoked using pytest's doctest functionality. Docstring unit tests should be reserved for small, simple functions and/or to demonstrate example usage for users. This allows for automated regression testing and breaking change detection, as, if the example fails, then there must have been some change which will impact how the code is used. For numpy-style docstrings, docstring tests should live under the "Examples" section.

For more information on how pytest can be leveraged to write unit tests, please check read the pytest documentation.

7.4.3 Code Coverage

When unit tests are invoked using tox, pytest-cov will be used to generate a coverage report, showing which lines were covered. Additionally, a coverage file will be placed in the project root with the tox environment used as the file extension (i.e. `.coverage.py{36,37,38,39}-unit`). Please use these coverage resources to help write unit tests for your PRs as needed. Note that code for benchmark wrappers are not included in these coverage reports, as benchmarks will be tested with functional tests, rather than unit tests.

7.5 Docker Image Testing

7.5.1 GitHub Workflow Walkthrough

benchmark-wrapper uses GitHub Actions to automate builds of wrapper Docker images. This is done through the workflow defined in build.yaml. The basic structure of the workflow is as follows:

1. Use ci/build_matrix.py to discover Dockerfile candidates for build and create two GitHub Actions Job Matrices. One will be a 'build matrix' that will organize building each selected Dockerfile candidate on each supported architecture. The other will be a 'manifest matrix' that will organize creating a manifest for each selected Dockerfile candidate that includes the built images across the supported architectures.

2. For each Dockerfile on each architecture, build the Dockerfile, and if on the master branch, then tag with `latest` and push to quay. If the build was manually triggered by a user, then instead of tagging with `latest`, tag with the given input tag and only push to quay if the user says to do so.
3. If we pushed to quay in the previous step, then for each Dockerfile, create a manifest that includes each successfully built architecture image and then push it to quay.

Note that:

- A Dockerfile candidate is a Dockerfile within the benchmark-wrapper repository that sits underneath the parent directory for each wrapper. It's expected that each wrapper has a single Dockerfile that is compatible across architectures, rather than having multiple Dockerfiles split by architecture.
- The build matrix Python script will dynamically discover all available Dockerfiles within the benchmark-wrapper repository and then only select those that need to be built based on what has changed.
- Right now we support `amd64` are currently working on `arm64` images.

Here's a diagram summarizing all those words:

And now let's work through an example, just to drive it all home.

Let's say that I push two changes to the master branch, one within the uperf wrapper under `snafu/benchmarks/uperf` and one within the fio wrapper under `snafu/fio_wrapper`. Our GHA Workflow will start by kicking off a job to execute the build matrix Python script. The script will discover that the fio and uperf benchmarks have changed and that their Docker images need to be rebuilt. Four jobs will then be kicked off to do the following in parallel with fail-fast disabled:

1. Build uperf on `amd64`, tag with `amd64-latest`, and push to quay
2. Build uperf on `arm64`, tag with `arm64-latest`, and push to quay
3. Build fio on `amd64`, tag with `amd64-latest`, and push to quay
4. Build fio on `arm64`, tag with `arm64-latest`, and push to quay

After these jobs have completed, regardless of pass/fail for each job, two more jobs will be kicked off to do the following (in parallel again with no fast-fail):

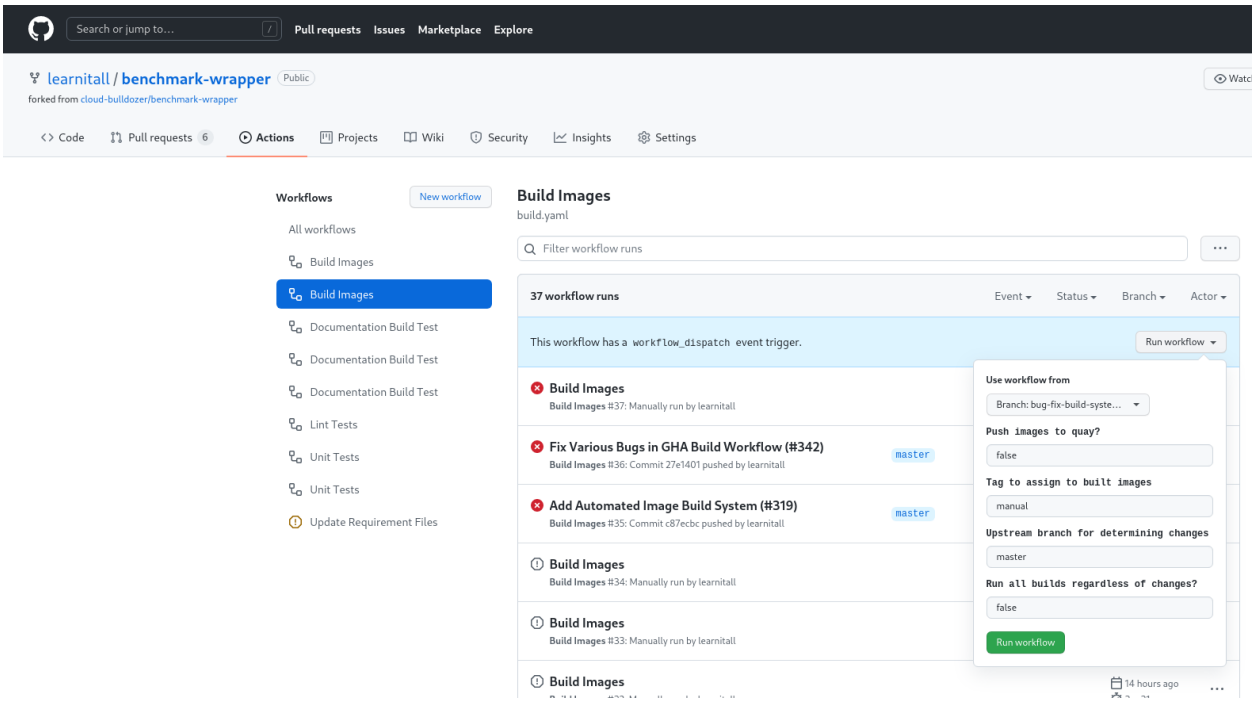
1. Create a manifest named `latest` for uperf with the `amd64-latest` and `arm64-latest` images and push to quay
2. Create a manifest named `latest` for fio with the `amd64-latest` and `arm64-latest` images and push to quay

If the uperf image on `arm64` fails to build, then the `amd64` image will still be built and pushed to quay. The manifest will be updated with the new `amd64` image and still reference the existing `arm64` image within quay.

Now let's say that I push to the main branch a change within `run_snafu.py`, our main entrypoint script. Our GHA Workflow will again kick off a job to execute the build matrix Python script. This time the script sees that a core component of snafu has changed and that a rebuild is required for each wrapper image. The necessary image build and manifest build jobs will then be kicked off.

7.5.2 Automated Build Testing

What's great about GHA workflows is that they can be used for free within forks. Our build.yaml workflow includes a manual trigger which allows for these builds to be kicked off manually as needed. This can be used to test Dockerfile builds before creating a PR or for any 'thought experiment' changes. For instance, if I have a new branch called `my-new-feature` that modifies a core component of the snafu code base, I can test the image build for each wrapper with this change by going to my fork in GitHub, selecting the "Actions" tab, clicking on the "Build Images" workflow and then selecting the "Run Workflow" button:



These are the paramemters available:

- **Use workflow from**: Branch to checkout and run the workflow from. Defaults to your fork's default branch.
- **Upstream branch for determining changes**: Branch used to compare the source branch against. This is normally the branch that the source branch above will be merged into. Defaults to `master`.
- **Run all builds regardless of changes?**: If `true` then all available Dockerfile candidates will be built, regardless of whether or not they actually need to be based on the changes in the source branch. Defaults to `false`.

And these are the secrets that need to be set within your fork:

- **QUAY_ORG**: Organization or user within quay to publish the built images to.
- **QUAY_USER**: Quay user with push permissions to the target organization.
- **QUAY_TOKEN**: Token to authenticate the above quay user.

More information on getting started with quay can be found here.

7.6 Editing Documentation