metapipe Documentation

Release 0.1

Brian Schrader

April 13, 2016

Contents

Contents:

CHAPTER 1

Metapipe

A pipeline for building analysis pipelines.

Metapipe is a simple command line tool for building and running complex analysis pipelines. If you use a PBS/Torque queue for cluster computing, or if you have complex batch processing that you want simplified, metapipe is the tool for you.

Metapipe's goal is to improve **readability**, and **maintainability** when building complex pipelines.

In addition to helping you generate and maintain complex pipelines, **metapipe also helps you debug them**! How? Well metapipe watches your jobs execute and keeps tabs on them. This means, unlike conventional batch queue systems like PBS/Torque alone, metapipe can give you accurate error information, and even resubmit failing jobs! Metapipe enhances the power of any PBS/Torque queue!

- What if I *don't use PBS/Torque*, or *a queue system at all?*

1.1 What does it do?

In the bad old days (before metapipe), if you wanted to make an analysis pipeline, you needed to know how to code. **Not anymore!** Metapipe makes it easy to build and run your analysis pipelines! **No more code, just commands!** This makes your pipelines easy to understand and change!

A sample metapipe file can be found in Metapipe Syntax

1.2 No Queue? No Problem!

Lots of people don't use a PBS/Torque queue system, or a queue system at all, and metapipe can help them as well! Metapipe runs locally and will give you all the same benefits of a batch queue system! It runs jobs in parallel, and provide detailed feedback when jobs go wrong, and automatic job re-running if they fail.

To run metapipe locally, see the app's help menu!

```
metapipe --help
```

1.3 Other Queue Systems

Metapipe is a very modular tool, and is designed to support any execution backend. Right now we only support PBS, but if you know just a little bit of Python, you can add support for any queue easily! *More information coming soon!*

CHAPTER 2

Getting Started

This section contains a quick guide for installing, and using metapipe. For the detailed syntax guide, see the Metapipe Syntax

2.1 Installation

Metapipe is available on PyPi so installing is easy.

```
$ pip install metapipe
```

To make it easy, metapipe runs on Python 2.7, 3.4, and 3.5!

2.2 Using metapipe

By default, metapipe is both a command line tool and a Python module that can be used to build and run pipelines in code. This means that whether you're a user, or a developer Metapipe can be adapted to fit your needs.

To see metapipe's help menu, type the following, just as you'd expect.

```
$ metapipe --help
```

2.3 Sample Pipeline

Here's a simple pipeline you can use for testing metapipe. Typically complex pipelines are used for things like bioinformatics, or some batch processing

But first, we need some sample files to work with. Run these commands to generate them.

```
$ echo "SAMPLE DATA 1" > test_file.1.txt
$ echo "SAMPLE DATA 2" > test_file.2.txt
$ echo "SAMPLE DATA 3" > test_file.3.txt
```

Now that we have our data, let's analyze it! Here's our sample pipeline:

```
[COMMANDS]
# Remove the ending number from each of our data files.
cut -f 1-2 -d ' ' {1||2||3} > {o}
```

```
# Paste each of the files together and save it to a final output.
# Since this is our last step, and only 1 output there's no need to have
# metapipe name the output file. We'll call it something ourselves.
paste {1.1,1.2,1.3} > final_output.txt

[FILES]
1. test_file.1.txt
2. test_file.2.txt
3. test_file.3.txt
```

Save that as `sample_pipeline.mp`, open a terminal, and `cd` to that directory.

2.3.1 Run the sample pipeline locally

Local execution is the default for metapipe so you just need to specify your metapipe file and an output destination.

```
$ metapipe -o pipeline.sh sample_pipeline.mp
```

This will generate an output script named `pipeline.sh` which will run the pipeline. Simply run it to start your pipeline!

```
$ sh pipeline.sh
```

That's it! Metapipe will run in the foreground watching your jobs complete until everything finishes.

2.3.2 Run the sample pipeline on PBS

Simply change the metapipe command to the following:

```
$ metapipe -o pipeline.sh -j pbs sample_pipeline.mp
```

Then simply submit metapipe as a job:

```
$ qsub pipeline.sh
```

Metapipe will run as a job on the PBS/Torque queue and submit other jobs to the same queue! It will keep tabs on the running jobs and submit them when they're ready, then exit when all jobs finish.

CHAPTER 3

Metapipe Syntax

The syntax for Pipeline Config files is as follows.

3.1 Section Definitions

In each Metapipe file, there are a number of different sections you can specify. Each has their own purpose and function. Each section is denoted with a header in brackets at the top of the section.

All sections support comments, and in most sections, they are not parsed as input.

3.1.1 Commands

The commands section is the only required Metapipe config section. Specified by the `[COMMANDS]` header, this is where the various steps of the pipeline are specified. Commands are very similar to normal shell commands, and most shell commands are valid. The only difference is in the input/output of each command. For these sections, use Metapipe's command syntax to indicate the location and desired input and output.

Example:

```
[COMMANDS]
# Here we cat a hardcoded input file into sed
# and redirect the output to a metapipe output token.
cat somefile.txt | sed 's/replace me/with me' > {o}
```

Metapipe automatically creates a filename for the given output token and assigns that file an alias. The alias structure is `command_number.command_iteration-output_number`, where the output number is optional.

Important: Commands are *NOT* run sequentially. As commands are parsed, they are evaluated based on what inputs they take in and what outputs they generate. For more information: see *Command Structure*. Commands are run as soon as they are deemed ready and any command that does not specify inputs via Metapipe's input patterns will be run immediately.

3.1.2 Paths

The paths section allows users to simplify their commands by creating aliases or short names to binaries. Paths are structured as a single word alias followed by a space and the rest of the line is considered the path. The paths section is denoted by the `[PATHS]` header.

```
[COMMANDS]
# Here we've aliased Python. When the script is generated,
# the hardcoded path will be substituted in.
python2 my_script.py

# Here we're using the builtin python and using paths
# to simplify the arguments.
python my_script.py somefile

[PATHS]
python2 /usr/local/bin/python2.7.4
somefile /a/long/file/path
```

Paths can also be used to create pseudo-variables for long configuration options. When doing this, it's recommended to use a bash-variable-like syntax because it reminds the reader that the variable is not a literal in the command.

Reminder: Paths are substituted in after the inputs have been processed. This means that `{}` characters are treated as literals and not as input markers.

```
[COMMANDS]
# Here, the braces represent an output token,
# but the $OPTIONS variable will be evaluated
# as a literal {}
python my_script.py -o {o} $OPTIONS

[PATHS]
$OPTIONS -rfg --do-something --no-save --get --no-get -I {}
```

3.1.3 Files

For a given pipeline, there is usually a set of input or auxiliary files. These files go through the analysis and other steps require the output of one command as the input for another. This is where most of the power of Metapipe's syntax comes into play. The files section is denoted as `[FILES]`.

Files are specified using a number followed by a period, and then the path to the given file. The number is the file's alias, and once that alias is assigned, it can be used in commands.

```
[COMMANDS]
cat {1} | sed 's/replace me/with me' > {o}
cat {2} | cut -f 1 | sort | uniq > {o}

[FILES]
1. somefile.1
2. /path/to/somefile.2
```

In this example, we use the aliases of files 1 and 2 to perform different analysis on each file. Then, when the input files need to change, they can be changed in the `[FILES]` section and the pipeline remains the same.

3.1.4 Job Options

The job options section, denoted by `[JOB_OPTIONS]`, is a section that allows the user to specify a global set of options for all jobs. This helps reduce pipeline redundancy.

```
# Each of the commands in this pipeline need to
# be working in a scratch directory.
[COMMANDS]
```

```
cat somefile.1.txt | sed 's/replace me/with me' > {o}
cat somefile.2.txt | sed 's/replace me/with you' > {o}
cat somefile.3.txt | sed 's/replace you/with me' > {o}

[JOB_OPTIONS]
set -e
cd /var/my_project/

# This config will result in the following:
# ------- Job 1 ---------
set -e
cd /var/my_project/
cat somefile.1.txt | sed 's/replace me/with me' > {o}
```

The set of commands in Job Options will be carried over to every job in the pipeline. This can be extremely useful when setting configuration comments for a queue system.

```
# Each of the commands needs 4GB of RAM
[COMMANDS]
cat somefile.1.txt | sed 's/replace me/with me' > {o}
cat somefile.2.txt | sed 's/replace me/with you' > {o}
cat somefile.3.txt | sed 's/replace you/with me' > {o}

[JOB_OPTIONS]
#PBS -l mem=4096mb
```

Job Options allow users to make their pipelines more clear and less redundant by allowing them to follow the DRY principle.

3.2 Command Structure

Now that all of the concepts and supported sections have been explained, it's time to take a look at the command structure and how to take advantage of Metapipe's advanced features.

3.2.1 Input Patterns

Consider the following command:

```
[COMMANDS]
python somescript {1||2||3}

[FILES]
1. some_file1.txt
2. some_file2.txt
3. some_file3.txt
```

This command will run the python script 3 times in parallel, once with each file specified. The output will look something like this:

```
# Output
# ------

python somescript some_file1.txt
python somescript some_file2.txt
python somescript some_file3.txt
```

Running a script with multiple inputs

Let's say that you have a script with takes multiple files as input. In this case the syntax becomes:

```
[COMMANDS]
python somescript {1,2,3}

[FILES]
1. some_file1.txt
2. some_file2.txt
3. some_file3.txt

# Output
# ------

python somescript some_file1.txt some_file2.txt some_file3.txt
```

3.2.2 Output Patterns

Whenever a script would take an explicit output filename you can use the output pattern syntax to tell metapipe where/what it should use.

```
[COMMANDS]
python somescript -o {o} {1||2||3}

[FILES]
1. some_file1.txt
2. some_file2.txt
3. some_file3.txt

# Output
# ------

python somescript -o mp.1.1.output some_file1.txt
python somescript -o mp.1.2.output some_file2.txt
python somescript -o mp.1.3.output some_file3.txt
```

Metapipe will generate the filename with the command's alias inside. An upcoming feature will provide more useful output names.

Implicit or Hardcoded output

In a case where the script or command you want to use generates an output that is not passed through the command, but you need to use for another step in the pipeline, you can use output patterns to tell metapipe what to look for.

Consider this:

```
[COMMANDS]
# This command doesn't provide an output filename
# so metapipe can't automatically track it.
./do_count {1||2}
./analyze.sh {1.*}

[FILES]
1. foo.txt
2. bar.txt
```

This set of commands is invalid because the second command (`./analyze.sh`) doesn't know what the output of command 1 is because it isn't specified. The split command generates output based on the input filenames it is given.

Since we wrote the `./do_count` script, we know that it generates files with a `.counts` extension. But since we don't explicitly specify the files, in this case Metapipe cannot assume the file names generated by step 1 and this config file is invalid.

We can tell metapipe what the output should look like by using an output pattern.

```
[COMMANDS]
# We've now told Metapipe what the output file name
# will look like. It can now track the file as normal.
./do_counts {1||2} #{o:*.counts}
./analyze.sh {2.*}

[FILES]
1. foo.txt
2. bar.txt
```

The above example tells metapipe that the output of command 1, which is hardcoded in the script will have an output that ends in `.counts`. Now that the output of command 1 is known, command 2 will wait until command 1 finishes.

When the output marker has the form `{o}`, then metapipe will insert a pregenerated filename to the command. The output marker `{o:<pattern>}` means that the output of the script is *not* determined by the input of the script, but it *will* match given pattern. This means that later commands will be able to reference the files by name.

3.2.3 Multiple Inputs and Outputs

Often times a given shell command will either take multiple dynamic files as input, or generate multiple files as output. In either case, metapipe provides a way to manage and track these files.

For multiple inputs, metapipe expects the number of inputs per command to be the same, and will iterate over them in order.

Example:

```
# Given the following:
[COMMANDS]
bash somescript {1||2||3} --conf {4||5||6}  > {o}

[FILES]
1. somefile.1
2. somefile.2
3. somefile.3

# Metapipe will return this:
bash somescript somefile.1 --conf somefile.4  > mp.1.1.output
bash somescript somefile.2 --conf somefile.5  > mp.1.2.output
bash somescript somefile.3 --conf somefile.6  > mp.1.3.output
```

Metapipe will name the multiple output files as follows (in order from left to right):

`mp.{command_number}.{sub_command_number}-{output_number}`

Example:

```
# Given an input like the one below:
[COMMANDS]
bash somescript {1||2||3} --log {o} -r {o}
```

```
[FILES]
1. somefile.1
2. somefile.2
3. somefile.3

# metapipe will generate the following:
bash somescript somefile.1 --log mp.1.1-1.output -r mp.1.1-2.output
bash somescript somefile.2 --log mp.1.2-1.output -r mp.1.2-2.output
bash somescript somefile.3 --log mp.1.3-1.output -r mp.1.3-2.output
```

3.3 Sample config.mp file

```
[COMMANDS]
# Here we run our analysis script on every gzipped file
# in the current directory and output the results to a file.
python my_custom_script.py -o {o} {*.gz||}

# Take all the outputs of step 1 and feed them to cut.
cut -f 1 {1.*||} > {o}

# Oh no! You hardcode the output name? No problem! Just tell metapipe
# what the filename is.
python my_other_custom_code.py {2.*} #{o:hardcoded_output.csv}

# Now you want to compare your results to some controls? Ok!
# Metapipe wil compare your hardcoded_output to all 3
# controls at the same time!
python my_compare_script.py -o {o} $OPTIONS --compare {1||2||3} {3.1}

# Finally, you want to make some pretty graphs? No problem!
# But wait! You want R 2.0 for this code? Just create an alias for R!
Rscript my_cool_graphing_code.r {4.*} > {o}

[FILES]
1. controls.1.csv
2. controls.2.csv
3. controls.3.csv

[PATHS]
Rscript ~/path/to/my/custom/R/version
$OPTIONS -rne --get --no-get -v --V --log-level 1
```

CHAPTER 4

Scripting Metapipe

In addition to being a command line tool, metapipe is also a Python module. You can use this module to extend, or script metapipe to fit your specific uses. This section will discuss scripting metapipe, and building/running jobs using Python. For information on how to extend metapipe's builtin job types or queue system, see Extending Metapipe.

4.1 The Run Interface

The first, and easiest way to script Metapipe is by invoking it via the Python interface.

```
from metapipe import run

config_text = get_config_text()
run(config_text)
```

For detailed information, see the run method's docstring.

CHAPTER 5

Extending Metapipe

Metapipe provides 2 extension points for developers to extend it's functionality: custom Queues and custom Job Types. In most cases, custom queues are an advanced feature that most users and developers will not need to worry about, but if you must, it is there.

To add support for a queue system not included with metapipe, all you need to do is add a job type.

5.1 Custom Job types

All job types are subclasses of the `metapipe.models.Job` class. The base job class implements a lot of the functionality that is common between all job types, and has method stubs for the required functionality that needs to be implemented by any subclass. This section will cover what duty job subclasses have, how to subclass the main `Job` and what to fill in.

5.1.1 The Root Job Class

The code for the main job class can be found here. To create your own job type, simply subclass this as follows:

```
from metapipe.models import Job

class MyCustomJob(Job):

    def __repr__(self):
        return '<MyCustomJob: {}>'.format(self.cmd)
```

There are 6 methods you need to fill in to have a complete job class. Your full job subclass should have the following form:

```
class MyCustomJob(Job):

    def __repr__(self):
        return '<MyCustomJob: {}>'.format(self.cmd)

    # Override these...

    @property
    def cmd(self):
        """ Returns the command needed to submit the calculations.
        Normally, this would be just running the command, however if
        using a queue system, then this should return the command to
```

```
        submit the command to the queue.
        """
        pass

    def submit(self):
        """ Submits the job to be run. If an external queue system is used,
        this method submits itself to that queue. Else it runs the job itself.
        :see: call
        """
        pass

    def is_running(self):
        """ Returns whether the job is running or not. """
        pass

    def is_queued(self):
        """ Returns whether the job is queued or not.
        This function is only used if jobs are submitted to an external queue.
        """
        pass

    def is_complete(self):
        """ Returns whether the job is complete or not. """
        pass

    def is_error(self):
        """ Checks to see if the job errored out. """
        pass
```

The duty of the job types is to submit the jobs when asked by the queue, and to inform the queue about the status of jobs. The queue needs to know when a job is running, queued, complete, or when an error has occurred.

Each of the `is_*` callbacks should return a boolean value, and the cmd property should return the bash command (as an array of strings) that can be called to run the job. The job class has an attribute `filename` that contains the value of the bash script containing the job command (i.e. `['bash', self.filename]`).

IMPORTANT: All of the above handlers are required for custom job types to function properly.

Here is the code for the `cmd` property of the `PBSJob` class:

```
class PBSJob(Job):
    #...
    @property
    def cmd(self):
        return ['qsub', self.filename]
    #...
```

The `submit` call should do any logic pertaining to submitting the job or tracking the number of total submissions. For example, here is the code for submitting a job to the PBS queue:

```
class PBSJob(Job):
    #...
    def submit(self, job):
        if self.attempts == 0:
            job.make()
        self.attempts += 1
        out = call(job.cmd)
        self.waiting = False
        self.id = out[:out.index('.')]
    #...
```

As you can see, it keeps track of the number of times the job was submitted, and then calls the `call` function, provided in the root job module, to execute the job. Since PBS assigns job ids to each job at submission-time, it also captures that information and saves it for later use.

5.2 Custom Queues

In the event that your analysis requires more control over the submission process for jobs, the metapipe module also allows for the customization of queue logic by subclassing `metapipe.models.Queue`. This section will cover how to subclass the root queue, but it is left to the reader to determine why you might want to do this. From personal experience, customizing the queue should be a very rare requirement.

5.2.1 The Root Queue class

As is the case for custom job types, all queues inherit from the root Queue in `metapipe.models.Queue`, including the main `JobQueue` that is used by the metapipe command line tool.

To customize the response of the queue to various types of events subclass it and fill in the following methods, all the methods are optional so just omit any handlers that you don't need.

```
class MyCustomQueue(object):

    def __repr__(self):
        return '<MyCustomQueue: jobs=%s>' % len(self.queue)

    # Callbacks...

    def on_start(self):
        """ Called when the queue is starting up. """
        pass

    def on_end(self):
        """ Called when the queue is shutting down. """
        pass

    def on_locked(self):
        """ Called when the queue is locked and no jobs can proceed.
        If this callback returns True, then the queue will be restarted,
        else it will be terminated.
        """
        return True

    def on_tick(self):
        """ Called when a tick of the queue is complete. """
        pass

    def on_ready(self, job):
        """ Called when a job is ready to be submitted.
        :param job: The given job that is ready.
        """
        pass

    def on_submit(self, job):
        """ Called when a job has been submitted.
        :param job: The given job that has been submitted.
        """
```

```
        pass

    def on_complete(self, job):
        """ Called when a job has completed.
        :param job: The given job that has completed.
        """
        pass

    def on_error(self, job):
        """ Called when a job has errored.
        :param job: The given job that has errored.
        """
        pass
```

5.3 Using Your Custom Code

Once you have subclassed and filled in the required code for your custom job type or queue, it is time to use your code. If your code adapts metapipe to work on a common computing platform, or system then please consider contributing to the metapipe project. This helps the rest of the community use a broader range of hardware to solve our problems!

5.3.1 Building your custom pipeline

Use the following code to build your pipeline. This code is taken directly from [metapipe's app.py][app] tool which is the command line tool that metapipe uses to build pipelines.

```
import MyCustomJob

JOB_TYPES = {
    'my_custom_job_type': MyCustomJob
}

parser = Parser(config)
try:
    command_templates = parser.consume()
except ValueError as e:
    raise SyntaxError('Invalid config file. \n%s' % e)

pipeline = Runtime(command_templates, JOB_TYPES, 'my_custom_job_type')
```

IMPORTANT: Adding custom queues is coming soon!

For more information on how to script metapipe once you have custom jobs, see Scripting Metapipe

CHAPTER 6

Measuring Pipeline Progress

While Metapipe runs your pipeline, it writes updates to `stdout`. These can be helpful, but most times it can be more helpful to get additional information in a more helpful format.

Metapipe provides a few different methods of visualizing the progress of your pipeline. These options are specified by the `--report-type` option.

6.1 Text based reporting

```
--report-type text
```

This option is the default. Metapipe will write to `stdout` and this can be redirected to a file.

6.2 HTML based reporting

```
--report-type html
```

Using this option, Metapipe will generate an HTML report of the pipeline as it runs. This static report represents the current state of the pipeline and what steps have already been completed. The report also includes a progress bar that reports a visualization of the rough progress of the pipeline.

Important: This progress indicator is based on the number of overall steps to be completed and represents the number of steps remaining. This has no correlation with the amount of time remaining, as that depends on the length of time each step takes.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search