magma-suite

Release 1.0.1

Michele Berselli, Doug Rioux, Soo Lee and CGAP team

Jul 31, 2023

CONTENTS

This is a documentation for CGAP magma-suite, a collection of tools to manage meta-workflows automation.

CHAPTER

ONE

CONTENTS

1.1 Install

1.2 MetaWorkflow[json]

MetaWorkflow[json] is a data format that describes the general structure of a multi-step workflow in json format. The format can store general information for the workflow (e.g. reference files, shared arguments, ...), as well as specific information for each of the steps (*StepWorkflow[json]*). The format is very flexible and allows to extend the minimal set of keys that are required with any number of custom keys.

1.2.1 Structure

```
{
  ## General MetaWorkflow[json] information
  #   These are general fields that are required by the parser,
  #   however, there is no check on the content that can be customized
  "name": <str>, # name for the MetaWorkflow[json]
  "uuid": <str>, # universally unique identifier

  ## General MetaWorkflow[json] arguments
  #   These are general arguments that are used by multiple steps
  #   and can be grouped here to avoid repetition
  "input": [

    # Structure for a file argument
    {
      # These are necessary fields
      "argument_name": <str>,
      "argument_type": "file",
      "files": <...>
      # These are optional fields
      #   It is possible to skip these fields or add custom ones
      "dimensionality": <int>
    },

    # Structure for a parameter argument
    {
      # These are necessary fields
```

(continues on next page)

(continued from previous page)

```
      "argument_name": <str>,
      "argument_type": "parameter",
      "value": <...>
      # These are optional fields
      #   It is possible to skip these fields or add custom ones
      "value_type": <str>
    }

    # Arguments with no value or file uuid can be specified as well
    #   and will need to be provided as input in MetaWorkflowRun[json]

  ],

  ## Steps for the MetaWorkflow[json]
  "workflows": [

    # Structure for StepWorkflow[json]
    {
      # General StepWorkflow[json] information
      #   These are general fields that are required by the parser,
      #   however, there is no check on the content that can be customized
      "name": <str>, # name for the StepWorkflow[json]
      "workflow": <str>, # universally unique identifier
      "config": { # configuration for the StepWorkflow[json]
        # example for AWS and Tibanna
        "instance_type": <str>,
        "ebs_size": <str> | "formula:<formula>",
              # !!! it is possible to specify formulas "formula:<formula>"
              #   values to be replaced must be defined as
              #   parameter arguments in MetaWorkflowRun[json] specific input !!!
        "EBS_optimized": <bool>,
        "spot_instance": <bool>,
        "log_bucket": <str>,
        "run_name": <str>,
        "behavior_on_capacity_limit": <str>
      },

      # Additional StepWorkflow[json] information
      #   Optional fields can be added and customized
      "dependencies": [], # allows to force general dependencies to steps

      # Example of additional Tibanna specific fields
      "custom_pf_fields": {
        # Example for CGAP data model
        # For an updated list of permissible properties, see schema at
        # https://github.com/dbmi-bgm/cgap-portal/blob/master/src/encoded/schemas/meta_
↪workflow.json
        "<filename>": {
             "file_type": <str>,
             "variant_type": <str>,
             "description": <str>,
             "linkto_location": [<str>, ...]
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "custom_qc_fields": {},

    # StepWorkflow[json] arguments
    #   These are the arguments that are used by the StepWorkflow[json]
    "input": [

      # Structure for a file argument
      {
        # These are necessary fields
        "argument_name": <str>,
        "argument_type": "file",

        # Linking fields
        #   These are optional fields
        #   If no source step is specified,
        #     the argument will be matched to general arguments by source_argument_name
        #     or argument_name if source_argument_name is missing
        #   First will try to match to argument in MetaWorkflowRun[json] specific input
        #     if no match is found will try to match to MetaWorkflow[json] default argument
        "source": <str>, # step that is source
        "source_argument_name": <str>,

        # Input dimension
        #   These are optional fields to specify input argument dimensions to use
        #     when creating the MetaWorkflowRun[json] or step specific inputs
        "scatter": <int>, # input argument dimension to use to scatter the step
                          #   !!! this will create multiple shards in the MetaWorkflowRun[json] structure !!!
                          #   the same dimension will be used to subset the input argument when creating the step specific input
        "gather": <int>, # increment for input argument dimension when gathering from previous steps
                         #   !!! this will collate multiple shards in the MetaWorkflowRun[json] structure !!!
                         #   the same increment in dimension will be used when creating the step specific input
        "input_dimension": <int>, # additional dimension used to subset the input argument when creating the step specific input
                                  #   this will be applied on top of scatter, if any, and will only affect the step specific input
                                  #   !!! this will not affect scatter dimension in building the MetaWorkflowRun[json] structure !!!
        "extra_dimension": <int>, # additional increment to dimension used when creating the step specific input
                                  #   this will be applied on top of gather, if any, and will only affect the step specific input
                                  #   !!! this will not affect gather dimension in building the MetaWorkflowRun[json] structure !!!
        # These are optional fields
```

(continues on next page)

(continued from previous page)

```
          #   It is possible to skip these fields or add custom ones
          "mount": <bool>,
          "rename": "formula:<parameter_name>",
                #  !!! formula:<parameter_name> can be used to
                #    specify a parameter name that need to be matched
                #    to parameter argument in MetaWorkflowRun[json] specific input
                #    and the value replaced !!!
          "unzip": <str>
        },

        # Structure for a parameter argument
        {
          # These are necessary fields
          "argument_name": <str>,
          "argument_type": "parameter",

          # These are optional fields
          #   If no value is specified,
          #     the argument will be matched to general arguments by source_argument_name
          #     or argument_name if source_argument_name is missing
          #   First will try to match to argument in MetaWorkflowRun[json] specific input
          #     if no match is found will try to match to MetaWorkflow[json] default␣
↪argument
          "value": <...>,
          "source_argument_name": <str>
        }

      ]
    }
  ]
}
```

1.3 MetaWorkflowRun[json]

MetaWorkflowRun[json] is a json data format that describes the structure of a multi-step workflow given the corresponding *MetaWorkflow[json]*, specific input and defined end points. Scatter, gather and dependencies information are used to create and link all the *shards* for individual steps (*WorkflowRun[json]*) that are necessary to reach end points based on the input.

1.3.1 Structure

```
{
  ## General MetaWorkflowRun[json] information
  "meta_workflow": "", # universally unique identifier
                       #   for the corresponding MetaWorkflow[json]

  ## Shards for MetaWorkflowRun[json]
  "workflow_runs" : [
```

(continues on next page)

(continued from previous page)

```
        # WorkflowRun[json] structure
        {
          # These are necessary fields
          "name": "",
          "status": "", # pending | running | completed | failed
          "shard": "", # x 1D | x:x 2D | x:x:x 3D | ...
          # These are optional fields
          #   or fields created during the processing
          "dependencies": [],
          "output": [
            # Structure for a file argument,
            #   only type of argument that can be output of a WorkflowRun[json]
            {
              # These are necessary fields
              "argument_name": "",
              "files": ""
            }
          ],
          # Additonal fields created to link the actual run
          "jobid": "", # run identifier
          "workflow_run": # universally unique identifier
        },

        # Example
        { "name": "step1",
          "workflow_run": "uuid-step1:0-run",
          "status": "complete",
          "output": [
            {
              "argument_name": "out_step1",
              "files": "uuid-out_step1:0"
            }
          ],
          "shard": "0"
        },
        { "name": "step1",
          "workflow_run": "uuid-step1:1-run",
          "status": "complete",
          "output": [
            {
              "argument_name": "out_step1",
              "files": "uuid-out_step1:1"
            }
          ],
          "shard": "1"
        },
        { "name": "step2",
          "workflow_run": "uuid-step2:0-run",
          "status": "running",
          "dependencies": ["step1:0"],
          "shard": "0"
        },
```

(continues on next page)

(continued from previous page)

```
        { "name": "step2",
          "workflow_run": "uuid-step2:1-run",
          "status": "running",
          "dependencies": ["step1:1"],
          "shard": "1"
        },
        { "name": "step3",
          "status": "pending",
          "dependencies": ["step2:0", "step2:1"],
          "shard": "0"
        }
  ],

  ## Specific input for MetaWorkflowRun[json]
  "input": [
    # Structure for a file argument
    {
      # These are necessary fields
      "argument_name": "",
      "argument_type": "file",
      "files": ""
    },

    # Structure for a parameter argument
    {
      # These are necessary fields
      "argument_name": "",
      "argument_type": "parameter",
      "value": ""
    }
  ],

  ## Final status
  "final_status": "", # pending | running | completed | failed

  ## Optional general fields for MetaWorkflowRun[json]
  "common_fields": {}
}
```

1.4 *metawfl* module (magma)

This is a module to work with *MetaWorkflow[json]* format.

1.4.1 Import the library

```
from magma import metawfl as wfl
```

1.4.2 Usage

MetaWorkflow object

MetaWorkflow object stores MetaWorkflow[json] general information, together with specific information for each of the steps as *StepWorkflow* objects.

Initialize MetaWorkflow object

```
import json

# Read input json
with open('.json') as json_file:
    data = json.load(json_file)

# Create MetaWorkflow object
wfl_obj = wfl.MetaWorkflow(data)
```

This will read MetaWorkflow[json] `.json` content into a *MetaWorkflow* object and create a *StepWorkflow* object for each of the steps in `workflows`.

Attributes

- `wfl_obj.steps`, stores *StepWorkflow* objects as dictionary.

  ```
  # wfl_obj.steps structure
  {
    step_obj_1.name = step_obj_1,
    step_obj_2.name = step_obj_2,
    ...
  }
  ```

- `wfl_obj.name`, stores `name` content as string, if any.
- `wfl_obj.uuid`, stores `uuid` content as string.
- `wfl_obj.input`, stores `input` content as list.
- `wfl_obj.workflows`, stores `workflows` content as list.

Write MetaWorkflowRun[json]

The method `wfl_obj.write_run(input_structure<str | str list>, [end_steps<str list>])` returns a *MetaWorkflowRun[json]* given specific input_structure and end steps. It is not necessary to specify names for end steps. If missing, shards are automatically calculated to run all the steps.

```
# input is a string or list of strings, up to 3-dimensions
input = [[['file_1', 'file_2'], ['file_3', 'file_4']]]

# end_steps, is a list of the final steps to run
# if missing, the end steps are automatically calculated to run everything
end_steps = ['step_5', 'step_6']

# run wfl_obj.write_run
run_json = wfl_obj.write_run(input, end_steps)
```

StepWorkflow object

Attributes

- `step_obj.name`, stores `name` content as string.
- `step_obj.workflow`, stores `workflow` content as string.
- `step_obj.config`, stores `config` content as dict.
- `step_obj.input`, stores `input` content as list.
- `step_obj.is_scatter`, stores `scatter` dimension for step as int.
- `step_obj.gather_from`, stores increment for input dimension for steps to gather from as dict.

  ```
  # step_obj.gather_from structure
  {
    step_obj_1.name = dimension_1,
    step_obj_2.name = dimension_2,
    ...
  }
  ```

- `step_obj.dependencies`, stores names of steps that are dependency as set, if any.

1.5 *metawflrun* module (magma)

This is a module to work with *MetaWorkflowRun[json]* format.

1.5.1 Import the library

```
from magma import metawflrun as run
```

1.5.2 Usage

MetaWorkflowRun object

MetaWorkflowRun object stores MetaWorkflowRun[json] general information, together with shards information as *WorkflowRun* objects.

Initialize MetaWorkflowRun object

```
import json

# Read input json
with open('.run.json') as json_file:
    data = json.load(json_file)

# Create MetaWorkflowRun object
wflrun_obj = run.MetaWorkflowRun(data)
```

This will read MetaWorkflowRun[json] `.run.json` content into a *MetaWorkflowRun* object and create a *WorkflowRun* object for each of the shards in `workflow_runs`.

Attributes

- `wflrun_obj.meta_workflow`, stores `meta_workflow` content as string.
- `wflrun_obj.input`, stores `input` content as list.
- `wflrun_obj.workflow_runs`, stores `workflow_runs` content as list.
- `wflrun_obj.runs`, stores *WorkflowRun* objects as dictionary.

```
# wflrun_obj.runs structure
{
  run_obj_1.shard_name = run_obj_1,
  run_obj_2.shard_name = run_obj_2,
  ...
}
```

- `wflrun_obj.final_status`, stores `final_status` as string.

Methods

The method `wflrun_obj.to_run()` returns a list of *WorkflowRun* objects that are ready to run (objects status is set to pending and dependencies run completed).

The method `wflrun_obj.running()` returns a list of *WorkflowRun* objects with status set to running.

The method `wflrun_obj.update_attribute(shard_name<str>, attribute<str>, value<any>)` updates *attribute value* for *WorkflowRun* object corresponding to *shard_name* in `wflrun_obj.runs`.

The method `wflrun_obj.runs_to_json()` returns `workflow_runs` as json. Builds `workflow_runs` directly from *WorkflowRun* objects in `wflrun_obj.runs`.

The method `wflrun_obj.to_json()` returns MetaWorkflowRun[json]. Builds `workflow_runs` directly from *WorkflowRun* objects in `wflrun_obj.runs`.

The method `wflrun_obj.reset_step(step_name<str>)` resets attributes value for *WorkflowRun* objects corresponding to step specified as *step_name*. Resets all shards associated to specified step.

The method `wflrun_obj.reset_shard(shard_name<str>)` resets attributes value for *WorkflowRun* object in runs corresponding to shard specified as *shard_name*. Resets only specified shard.

The method `wflrun_obj.update_status()` checks the status for all *WorkflowRun* objects, sets *MetaWorkflowRun* final status accordingly. Returns updated `wflrun_obj.final_status`.

WorkflowRun object

WorkflowRun is an object to represent a shard.

Attributes

- `run_obj.name`, stores `name` content as string.
- `run_obj.status`, stores `status` content as string (*pending* | *running* | *completed* | *failed*).
- `run_obj.shard`, stores `shard` content as string.
- `run_obj.shard_name`, stores `shard_name` (`name` + `shard`) content as string.
- `run_obj.output`, stores `output` content as list, default [].
- `run_obj.dependencies`, stores `dependencies` content as list, default [].

1.6 Other Modules (magma)

Modules to work with *MetaWorkflow* and *MetaWorkflowRun* objects:

- inputgenerator module -> *InputGenerator* object
- runupdate module -> *RunUpdate* object

1.6.1 Import the libraries

```
# Require metawfl and metawflrun modules
#   metawfl -> MetaWorkflow
#   metawflrun -> MetaWorkflowRun
from magma import metawfl as wfl
from magma import metawflrun as run

from magma import inputgenerator as ingen
from magma import runupdate as runupd
```

1.6.2 Usage

InputGenerator object

InputGenerator object allows to combine and use *MetaWorkflow* and *MetaWorkflowRun* objects to map arguments and create input and patching objects in json format.

Initialize InputGenerator object

```
import json

# Read input MetaWorkflow[json]
with open('.json') as json_file:
    data_wfl = json.load(json_file)

# Read input MetaWorkflowRun[json]
with open('.run.json') as json_file:
    data_wflrun = json.load(json_file)

# Creates MetaWorkflow object
wfl_obj = wfl.MetaWorkflow(data_wfl)

# Creates MetaWorkflowRun object
wflrun_obj = run.MetaWorkflowRun(data_wflrun)

# Create InputGenerator object
ingen_obj = ingen.InputGenerator(wfl_obj, wflrun_obj)
```

Create input json to run

The method `ingen_obj.input_generator()` returns a generator of `input_json, update_json` in json format:

- `input_json` stores necessary information to run a shard and can be used as input for *Tibanna*.
- `update_json` stores updated information for `workflow_runs` and `final_status`.

```
for input_json, update_json in ingen_obj.input_generator():
    # input_json -> json input for tibanna zebra
    # update_json -> {
```

(continues on next page)

(continued from previous page)

```
    #             'workflow_runs': [...],
    #             'final_status': 'STATUS'
    #            }
    # DO something
```

RunUpdate object

RunUpdate object allows to update and combine *MetaWorkflowRun* objects.

Initialize RunUpdate object

```
# Read input MetaWorkflowRun[json]
with open('.run.json') as json_file:
    data_wflrun = json.load(json_file)

# Creates MetaWorkflowRun object
wflrun_obj = run.MetaWorkflowRun(data_wflrun)

# Create RunUpdate object
runupd_obj = runupd.RunUpdate(wflrun_obj)
```

Methods

The method `runupd_obj.reset_steps(steps_name<str list>)` resets *WorkflowRun* objects corresponding to steps specified in *steps_name*. Resets all shards associated to specified steps. Returns updated `workflow_runs` and `final_status` information as json.

The method `runupd_obj.reset_shards(shards_name<str list>)` resets *WorkflowRun* objects corresponding to shards specified in *shards_name*. Resets only specified shards. Returns updated `workflow_runs` and `final_status` information as json.

The method `runupd_obj.import_steps(wflrun_obj<MetaWorkflowRun obj>, steps_name<str list>)` updates current *MetaWorkflowRun* object information, imports and use information from specified *wflrun_obj*. Updates *WorkflowRun* objects up to all steps specified in *steps_name*. Returns updated MetaWorkflowRun[json].

1.7 *checkstatus* module (magma_ff)

1.7.1 Check Status and Output

```
from magma_ff import checkstatus

wflrun_obj = None  # or any actual wflrun_obj
job_id = 'RBwlMTyOWvpZ' # JOBID for the run

# Create a CheckStatus object with an environment name
cs_obj = checkstatus.CheckStatusFF(wflrun_obj, env='fourfront-cgap')
```

(continues on next page)

(continued from previous page)

```
# get_status funtion can be used as stand-alone
cs_obj.get_status(job_id)
#'complete'

# get_output funtion can be used as stand-alone
cs_obj.get_output(job_id)
#[{'workflow_argument_name': 'sorted_bam',
#  'uuid': '07ae8a9c-260e-4b1b-80ae-ae59a624d746'}]

# Create a generator for check_running
cr = cs_obj.check_running()

# Iterate (needs a non-empty wflrun_obj for this to work)
next(cr)
```

1.8 *parser* module (magma_ff)

1.8.1 ParserFF object

While magma uses MetaWorkflow[json] or MetaWorkflowRun[json] formats, the portal uses slightly different formats where arguments are encoded as string. *ParserFF* provides methods to allow compatibility and convert between the portal and magma arguments representations.

Initialize ParserFF object

```
from magma_ff import parser

#input_json
#   -> portal representation of MetaWorkflow[json] or MetaWorkflowRun[json]

pff_obj = parser.ParserFF(input_json)
```

Methods

The method `pff_obj.arguments_to_json()` parses the portal representation of MetaWorkflow[json] or MetaWorkflowRun[json] stored in `self.in_json` attribute. If `input` key is found, converts and replaces arguments in `input` from *portal* string format to *magma* format. If `workflows`, for each steps converts and replaces arguments in `input` from *portal* string format to *magma* format. Updates and returns `self.in_json`.

1.9 Examples

1.9.1 Example 1.

This is a real example on how to create a new MetaWorkflowRun[json] from a MetaWorkflow[json], and import steps information from an old MetaWorkflowRun[json]. The code use magma_ff for compatibility with the portal.

```
# Libraries
from magma_ff import metawfl as wfl
from magma import metawflrun as run
# Using metawflrun from magma allows to keep the original json
#     as it is returned by the portal,
#     to apply the parser use magma_ff instead
from magma_ff import runupdate as runupd

# Get MetaWorkflow[json] from the portal
#   --> wfl_json

# Create MetaWorkflow object
wfl_obj = wfl.MetaWorkflow(wfl_json)

# Get old MetaWorkflowRun[json] from the portal
#   --> run_json_toimport

# Create MetaWorkflowRun object for old MetaWorkflowRun[json]
run_obj_toimport = run.MetaWorkflowRun(run_json_toimport)

# Create the new MetaWorkflowRun[json] from MetaWorkflow object
input_structure = [['a'],['b'],['c']]
run_json = wfl_obj.write_run(input_structure)

# Create MetaWorkflowRun object for new MetaWorkflowRun[json]
run_obj = run.MetaWorkflowRun(run_json)

# Create RunUpdate object
runupd_obj = runupd.RunUpdate(run_obj)

# Import information
steps_name = ['fastqc-r1', 'fastqc-r2', 'workflow_samplegeno', 'cgap-bamqc', 'workflow_
↪granite-mpileupCounts']
run_json_updated = runupd_obj.import_steps(run_obj_toimport, steps_name)
```

1.10 FF_functions

1.10.1 create_metawfr

The class `MetaWorkflowRunFromSampleProcessing(sp_uuid<str>, metawf_uuid<str>, ff_key<key>, expect_family_structure=True)` can be used to create a MetaWorkflowRun[json] from a *SampleProcessing* object on the portal. Initializing the class will automatically create the correct MetaWorkflowRun[json] that will be stored as attribute. The method `post_and_patch()` can be used to post the MetaWorkflowRun[json] as *MetaWorkflowRun* object on the portal and patch it to the *SampleProcessing*.

The class `MetaWorkflowRunFromSample` works similarly for *Sample* portal items.

```
from magma_ff import create_metawfr

# UUIDs
metawf_uuid = '' # uuid for MetaWorkflow[portal]
sp_uuid = '' # uuid for SampleProcessing[portal]

# ff_key
#   authorization key for the portal

# expect_family_structure
#   True | False
#   if True a family structure is expected,
#     samples are sorted for a trio analysis

# Create MetaWorkflowRunFromSampleProcessing object
#   this will automatically create the correct MetaWorkflowRun[json]
#   and store it as .meta_workflow_run attribute
create_metawfr_obj = create_metawfr.MetaWorkflowRunFromSampleProcessing(sp_uuid, metawf_
↪uuid, ff_key, expect_family_structure=True)

# Post and patch the MetaWorkflowRun[json] on the portal
create_metawfr_obj.post_and_patch()
```

1.10.2 run_metawfr

The function `run_metawfr(metawfr_uuid<str>, ff_key<key>, verbose=False, sfn='tibanna_zebra', env='fourfront-cgap', maxcount=None, valid_status=None)` can be used to run a *MetaWorkflowRun* object on the portal. Calculates which shards are ready to run, starts the runs with Tibanna and patches the metadata.

```
from magma_ff import run_metawfr

# UUIDs
metawfr_uuid = '' # uuid for MetaWorkflowRun[portal]

# ff_key
#   authorization key for the portal

# env
#   environment to use to access metadata
env = 'fourfront-cgap'

# sfn
#   step function to use for Tibanna
sfn = 'tibanna_zebra'

run_metawfr.run_metawfr(metawfr_uuid, ff_key, verbose=False, sfn=sfn, env=env,␣
↪maxcount=None, valid_status=None)
```

1.10.3 status_metawfr

The function `status_metawfr(metawfr_uuid<str>, ff_key<key>, verbose=False, env='fourfront-cgap', valid_status=None)` can be used to check and patch status for a *MetaWorkflowRun* object on the portal. Updates the status to `completed` or `failed` for finished runs. Updates *MetaWorkflowRun* final status accordingly. Patches the metadata.

```
from magma_ff import status_metawfr

# UUIDs
metawfr_uuid = '' # uuid for MetaWorkflowRun[portal]

# ff_key
#   authorization key for the portal

# env
#   environment to use to access metadata
env = 'fourfront-cgap'

status_metawfr.status_metawfr(metawfr_uuid, ff_key, verbose=False, env=env, valid_
↪status=None)
```

1.10.4 update_cost_metawfr

The function `update_cost_metawfr(metawfr_uuid<str>, ff_key<key>, verbose=False)` can be used to compute and patch the cost for a *MetaWorkflowRun* object on the portal (includes failed runs).

```
from magma_ff import update_cost_metawfr

# UUIDs
metawfr_uuid = '' # uuid for MetaWorkflowRun[portal]

# ff_key
#   authorization key for the portal

update_cost_metawfr.update_cost_metawfr(metawfr_uuid, ff_key, verbose=False)
```

1.10.5 import_metawfr

The function `import_metawfr(metawf_uuid<str>, metawfr_uuid<str>, sp_uuid<str>, steps_name<str list>, ff_key<key>, post=False, verbose=False, expect_family_structure=True)` can be used to create a new MetaWorkflowRun[json] from a older *MetaWorkflowRun* object on the portal. The specified *SampleProcessing* is used to create the basic structure for the new MetaWorkflowRun[json]. The steps listed in `steps_name` are then imported from the older *MetaWorkflowRun* object specified as `metawfr_uuid`. Returns the new MetaWorkflowRun[json]. Can automatically post the new MetaWorkflowRun[json] as *MetaWorkflowRun* object on the portal and patch it to the *SampleProcessing*.

```
from magma_ff import import_metawfr

# UUIDs
metawf_uuid = '' # uuid for MetaWorkflow[portal]
```

(continues on next page)

(continued from previous page)

```
metawfr_uuid = '' # uuid for old MetaWorkflowRun[portal] to import
sp_uuid = '' # uuid for SampleProcessing[portal]

# Post
#   post=True to automatically post new MetaWorkflowRun[json] object on the portal

# ff_key
#   authorization key for the portal

# steps_name
steps_name = ['workflow_granite-mpileupCounts', 'workflow_gatk-ApplyBQSR-check']

metawfr_json = import_metawfr.import_metawfr(metawf_uuid, metawfr_uuid, sp_uuid, steps_
↪name, ff_key, expect_family_structure=True)
```

1.10.6 reset_metawfr

The function `reset_status(metawfr_uuid<str>, status<str | str list>, step_name<str | str list>, ff_key<key>, verbose=False, valid_status=None)` can be used to re-set runs for a *MetaWorkflowRun* object on the portal that correspond to step/steps specified as `step_name` and with status in `status`.

```
from magma_ff import reset_metawfr

# UUIDs
metawfr_uuid = '' # uuid for MetaWorkflowRun[portal]

# ff_key
#   authorization key for the portal

# step_name
#   name or list of names for steps that need to be reset
step_name = ['workflow_granite-mpileupCounts', 'workflow_gatk-ApplyBQSR-check']

# status
#   status or list of status to reset
status = 'failed' # running | completed | failed

reset_metawfr.reset_status(metawfr_uuid, status, step_name, ff_key, verbose=False, valid_
↪status=None)
```

The function `reset_all(metawfr_uuid<str>, ff_key<key>, verbose=False, valid_status=None)` can be used to re-set all runs for a *MetaWorkflowRun* object on the portal.

```
from magma_ff import reset_metawfr

# UUIDs
metawfr_uuid = '' # uuid for MetaWorkflowRun[portal]

# ff_key
#   authorization key for the portal
```

(continues on next page)

(continued from previous page)

```
reset_metawfr.reset_all(metawfr_uuid, ff_key, verbose=False, valid_status=None)
```

The function `reset_failed(metawfr_uuid<str>, ff_key<key>, verbose=False, valid_status=None)` can be used to re-set all runs for a *MetaWorkflowRun* object on the portal with status `failed`.

```
from magma_ff import reset_metawfr

# UUIDs
metawfr_uuid = '' # uuid for MetaWorkflowRun[portal]

# ff_key
#   authorization key for the portal

reset_metawfr.reset_failed(metawfr_uuid, ff_key, verbose=False, valid_status=None)
```