tibanna$_f f Documentation$

Release 0.16.0

4DN DCIC

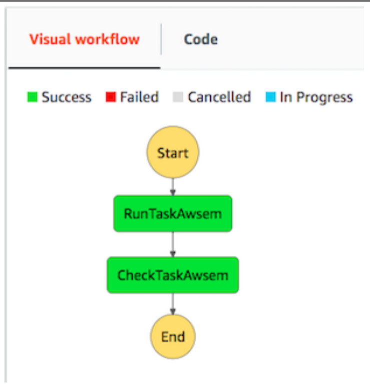
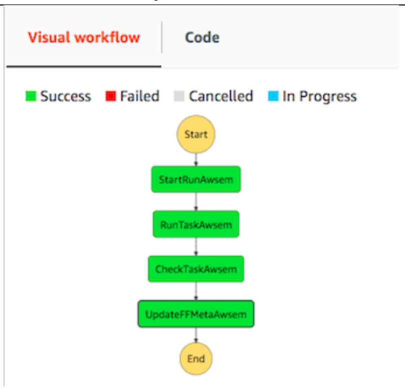
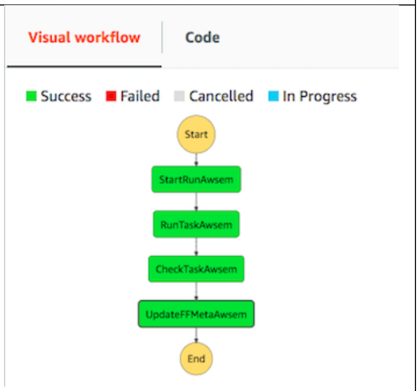
Feb 08, 2021

Contents

Tibanna_ff is an extension of Tibanna that integrates with 4DN/CGAP data portals. Tibanna is a software tool that helps you run genomic pipelines on the the Amazon (AWS) cloud. Tibanna_ff does the same but in integration with 4DN/CGAP data portal.

The generic Tibanna is called Unicorn, whereas the Tibanna_ff components that are specifically designed for 4DN (4DNucleome) and CGAP (Clinical Genome Analysis Platform) are called Pony and Zebra, respectively.

This documentation is written mostly for developers who want to understand the structure of the code arragements and details of the features and behaviors that are intended for Tibanna Unicorn, Pony and Zebra.

Tibanna Unicorn	Tibanna Pony	Tibanna Zebra
 Visual workflow Code Success Failed Cancelled In Progress Start RunTaskAwsem CheckTaskAwsem End	 Visual workflow Code Success Failed Cancelled In Progress Start StartRunAwsem RunTaskAwsem CheckTaskAwsem UpdateFFMetaAwsem End	 Visual workflow Code Success Failed Cancelled In Progress Start StartRunAwsem RunTaskAwsem CheckTaskAwsem UpdateFFMetaAwsem End

Tibanna Pony is an extension of Tibanna Unicorn used specifically for 4DN. Pony has two additional steps that communicate with the 4DN Data Portal and handle 4DN metadata. Pony requires access keys to the 4DN Data Portal and the 4DN DCIC AWS account.

Tibanna Zebra is an extension of Tibanna Unicorn used specifically for CGAP. Zebra has two additional steps that communicate with the CGAP Data Portal and handle CGAP metadata. Zebra requires access keys to the CGAP Data Portal and the CGAP AWS account.

Contents:

CHAPTER 1

News

From Tibanna `0.10.0` (also `tibanna_ff 0.10.0`), Tibanna_ff is no longer part of the `tibanna` package, but is released as a separate package that requires `tibanna` as dependency.

CHAPTER 2

Overview of the Tibanna code structure

Tibanna Pony (`tibanna_4dn`) and Zebra (`tibanna_cgap`) are built upon Tibanna Unicorn (`tibanna`, independent of any data portal). Code for Pony and Zebra uses code for Unicorn by either importing or inheriting. The code shared between Pony and Zebra that are not a part of Unicorn is stored in the shared component `tibanna_ffcommon`. All of these use AWSEM (Automonous Workflow Step Executor Machine) at the core, which is an EC2 instance that is auto-configured by Tibanna that does its job automonously and terminates itself at the end.

2.1 Repository & Directory structure

- https://github.com/4dn-dcic/tibanna
  - `tibanna` : code for Unicorn
  - `awsf` : code that runs on AWSEM (commonly used by Unicorn, Pony and Zebra)
- https://github.com/4dn-dcic/tibanna_ff
  - `tibanna_4dn` : code for Pony
  - `tibanna_cgap` : code for Zebra
  - `tibanna_ffcommon` : code shared between Pony and Zebra that are not part of Unicorn

Each of the three variants (Unicorn, Pony or Zebra) consists of a core API (`core.py`), CLI (`__main__.py`), lambdas (`/lambdas`) and set of python modules that are used by the former three.

2.2 AWSEM

The code in `awsf` is not a part of any Python package, but the scripts in the folder is pulled by an AWSEM EC2 instance directly from the public tibanna Github repo. Currently, `awsf` is still using Python 2.7, whereas all the other code is based on Python 3.6. The reason `awsf` uses Python 2.7 is because it runs on the pre-built Tibanna AMI which is based on Python 2.7 and we haven't updated the AMI yet.

2.3 AWS Lambda

The code for individual AWS Lambda functions is defined in individual `.py` files inside the `lambdas` directory under the package directory (e.g. `tibanna_4dn`, `tibanna_cgap`, or in case of unicorn, `tibanna` in the `tibanna` repo.)

2.3.1 Unicorn

A Unicorn consists of two AWS Lambda functions - `run_task_awsem` and `check_task_awsem`.

2.3.2 Pony

A Pony step function consists of four Lambda functions - in addition to `run_task_pony`, `check_task_pony`, it has `start_run_pony` and `update_ffmeta_pony`. Additionally, `tibanna_4dn`'s `deploy_pony` and `deploy_core` functions allow deploying other Lambdas that are not a part of a Pony step function. These include the following:

- `run_workflow_pony` : a Lambda function that triggers a workflow run on the `tibanna_pony` step function, that serves as a fourfront endpoint.
- `validate_md5_s3_trigger_pony` : a Lambda function that gets triggered upon file upload to a fourfront bucket. Once triggered, it invokes `tibanna_initiator` step function which in turn invokes `validate_md5_s3_initiator_pony` Lambda.
- `validate_md5_s3_initiator_pony` : a Lambda function that triggers `md5sum` and `fastqc` workflow runs on the `tibanna_pony_tmp_md5` step function.
- `status_wfr_pony` : a mysterious Lambda function that does something

The `.py` files for the Lambdas do not have the suffix `pony` in their file names, but the Lambdas do always have the suffix, to differentiate them from unicorn or zebra Lambdas.

2.3.3 Zebra

Like Pony, a Zebra step function consists of four Lambda functions - in addition to `run_task_zebra`, `check_task_zebra`, it has `start_run_zebra` and `update_ffmeta_zebra`. Additionally, `tibanna_cgap`'s `deploy_zebra` and `deploy_core` functions allow deploying other Lambdas that are not a part of a Zebra step function. These include the following:

- `run_workflow_zebra` : a Lambda function that triggers a workflow run on the `tibanna_pony` step function, that serves as a fourfront endpoint.
- `validate_md5_s3_trigger_zebra` : a Lambda function that gets triggered upon file upload to a cgap bucket. Once triggered, it invokes `tibanna_initiator_zebra` step function which in turn invokes `validate_md5_s3_initiator_zebra` Lambda.
- `validate_md5_s3_initiator_zebra` : a Lambda function that triggers `md5sum` and `fastqc` workflow runs on the `tibanna_zebra_tmp_md5` step function.
- `status_wfr_zebra` : a mysterious Lambda function that does something

The `.py` files for the Lambdas do not have the suffix `zebra` in their file names, but the Lambdas do always have the suffix, to differentiate them from unicorn or pony Lambdas.

2.4 AWS Step Functions

The code that describes a step function structure is in `stepfunction.py` in `tibanna`, `tibanna_ffcommon`, `tibanna_4dn` and `tibanna_cgap`. The step function class of `tibanna_4dn` (class `StepFunctionPony`) and `tibanna_cgap` (class `StepFunctionZebra`) inherit from that of `tibanna_ffcommon` (class `StepFunctionFFAbstract`) which in turn inherits from that of `tibanna` (class `StepFunctionUnicorn`). Class `StepFunctionFFAbstract` is not an actually functional step function but works as a common component that both `StepFunctionPony` and `StepFunctionZebra` can inherit from.

CHAPTER 3

Installation and dependencies

3.1 Installation

To install `tibanna`,

```
pip install tibanna
```

If `tibanna` is installed correctly, you can do the following.

```
> import tibanna
```

```
# this requires AWS credential set up as well
tibanna -h
```

To install `tibanna_4dn` and `tibanna_cgap`,

```
pip install tibanna_ff
# or pip install tibanna-ff
```

If you install `tibanna_ff`, `tibanna` will also be installed as its dependency. (no need to install `tibanna` separately)

If `tibanna_ff` is installed correctly, you can do the following.

```
> import tibanna
> import tibanna_4dn
> import tibanna_cgap
> import tibanna_ffcommon
```

```
# these require AWS credential set up as well
tibanna -h
tibanna_4dn -h
tibanna_cgap -h
```

3.2 Environment variables

The following environment variables are required for `tibanna`, unless `.aws/credentials` and `.aws/config` are set up.

```
export AWS_ACCESS_KEY_ID=<aws_key>
export AWS_SECRET_ACCESS_KEY=<aws_secret_key>
export AWS_DEFAULT_REGION=<aws_region>
```

To use `tibanna_4dn` or `tibanna_cgap`, the following environment variable is additionally required. (This is available only for the 4DN/CGAP developer team.)

```
export S3_ENCRYPT_KEY=<fourfront_s3_encrypt_key>
```

Optionally, for both cases, the following environment variable can be set up to be able to skip specifying `--sfn <step_function_name>` for most functions including `run_workflow` and `stat`.

```
export TIBANNA_DEFAULT_STEP_FUNCTION_NAME=<step_function_name>
```

For example,

```
export TIBANNA_DEFAULT_STEP_FUNCTION_NAME=tibanna_unicorn_monty
```

CHAPTER 4

Key CLI functions

CLI entrypoints for unicorn, pony and zebra are as follows. With `-h` option, one can see the list of subcommands available for each.

```
tibanna -h
tibanna_4dn -h
tibanna_cgap -h
```

4.1 Tibanna Deployment

```
tibanna deploy_unicorn [options]
```

```
tibanna_4dn deploy_pony [options]
```

```
tibanna_cgap deploy_zebra [options]
```

The above three are *not* interchangeable and each should be used to deploy a tibanna step function, lambdas and IAM permissions of its own kind.

Even for deploying a single lambda function, we should use the right entry point as below.

```
tibanna deploy_core -n <lambda_name> [options]
```

```
tibanna_4dn deploy_core -n <lambda_name> [options]
```

```
tibanna_cgap deploy_core -n <lambda_name> [options]
```

4.2 Running Workflow

```
tibanna run_workflow -i <input_json> --sfn=<stepfunctionname>
```

```
tibanna_4dn run_workflow -i <input_json> --sfn=<stepfunctionname>
```

```
tibanna_cgap run_workflow -i <input_json> --sfn=<stepfunctionname>
```

The above three can be used interchageably, as long as the correct step function name is used. i.e. the following command still works and would submit a job to `tibanna_pony` even if the entry point `tibanna_cgap` was used.

```
tibanna_cgap run_workflow -i <input_json> --sfn=tibanna_pony
```

CHAPTER 5

Behaviors of workflow runs

This section describes the expected behavior of a workflow run in the context of various options and features that Tibanna_pony or Tibanna_zebra supports. Overall, Pony and Zebra behave in a very similar way, with just a few very specific differences (see below). The features and behaviors of Pony is tightly associated with `fourfront` (https://github.com/4dn-dcic/fourfront) and those of Zebra with `cgap-portal` (https://github.com/dbmi-bgm/cgap-portal).

5.1 Metadata (overview)

The database schemas implemented in `fourfront` and `cgap-portal` that are relevant to Pony and Zebra are the following:

- Created by Tibanna Pony and Zebra
  - WorkflowRun
  - FileProcessed
  - QualityMetric (and those inherited from QualityMetric)
    - QualityMetricWorkflowrun (resource metric report for the run itself)
    - processed QualityMetric (e.g. QualityMetricFastqc, QualityMetricBamcheck, …)
- May be handled as an input of a workflow run
  - FileFastq
  - FileReference
  - FileProcessed (a processed file created from a previous run can be an input)
  - Workflow (workflow itself is an input of a workflow run)

For every workflow run, Pony and zebra both create a `WorkflowRun` object (`WorkflowRunAwsem` more specifically, which inherits from `WorkflowRun`) with a new `uuid` and an `awsem_job_id` that matches the job id of the run. They also create `FileProcessed` items for output files that we want to keep (`Output processed file`) that has a legit file format defined in the portal (e.g. `bam`), sometimes has an accompanying file (`extra_file`), again with a legit file format (e.g. `bai`). Not all workflow runs create a processed file output and depending on the type of output, a `QualityMetric` object may be created (`Output QC file`) or some field of the input file may be filled (e.g. `md5sum` and `file_size`) (`Output report file`) or a new extra file of an input file (`Output to-be-extra-input file`) may be created. Each of the `FileProcessed` and `QualityMetric` objects created is assigned a new `uuid`. Input files, processed files and `QualityMetric` objects are linked from the current `WorkflowRun` object and the `QualityMetric` objects are linked from a specified file (either input or processed).

If you rerun the same workflow run, it will not overwrite the existing `WorkflowRun`, `FileProcessed` or `QualityMetric` objects, but will create new ones. However, if a `QualityMetric` item is linked from any input file, this link would be replaced by the new `QualityMetric`. The old `QualityMetric` will still exist but just not linked from the input file any more. However, if the workflow run creates a new `extra_file` of an input file, a rerun will replace the file on `S3` without changing the metadata of the input file. This is harder to trace, so to be safe, one can use an option `"overwrite_input_extra" : true` to allow the overwrite - without this option, by default, the rerun will fail to start.

The metadata are created at the beginning of a workflow run except `QualityMetric` objects. At the end of a run, they are patched with the status. If the run is successful, the `WorkflowRun` object is patched a status `complete`. If there was an error, it is patched a status `error`.

A resource metric report is linked from `WorkflowRun` at the end of each run as a `QualityMetricWorkflowrun` object.

5.2 Workflow Run Identifier

Tibanna provides two identifiers for each workflow run: AWSEM Job ID (or in short, job id) and Execution ARN. They look like below for example:

- AWSEM Job ID (job id): 12-letter random string; each EC2 instance is tagged with a corresponding AWSEM Job ID.
  - e.g. `Aq1vSsEAEiSM`
  - e.g. `esAlf8LRZvNd`
  - e.g. `XbYhNcbtFKwG`
- Execution ARN: AWS ARN-style string that contains info about AWS account, region, step function and execution name on the step function.
  - e.g. `arn:aws:states:us-east-1:643366669028:execution:tibanna_pony:hic-bam-run_workflo`

In addition to these two identifiers, a Tibanna Pony or Zebra run is also associated with a WorkflowRun uuid, which is uniquely created for each workflow run. It is an identifier for a WorkflowRun object.

- WorkflowRun uuid
  - e.g. `b94e6891-c649-4178-88e9-fad59f04dd7a`
    - One can access the workflow run on the 4DN data portal if it's released to public or if you're logged in and have permission to view this object. https://data.4dnucleome.org/b94e6891-c649-4178-88e9-fad59f04dd7a

5.3 Config

The `config` of pony/zebra input json is directly passed to unicorn and is pretty much the same. There are some additional `fields` for pony and zebra that can be specified in `config`.

5.3.1 Additional fields for pony and zebra

- `"overwrite_input_extra" : true|false` (default `false`) : if an output file type is `Output to-be-extra-input file`, a rerun of the same workflow run will fail to start unless this flag is set to be `true`, to protect an existing extra file of an input file created by a previous run or an existing run that will create an extra file of an input file. One should use this flag only if one is sure that the previous or the current run has a problem and the output needs to be overwritten.
- `"email" : true|false` (default `false`) : if this flag is set to be `true`, it will send an email from `4dndcic@gmail.com` to itself (in case of `pony`) or `cgap.everyone@gmail.com` to itself (in case of `zebra`). To enable this to work, I had manually registered and verified these two emails on AWS Simple Email Service (SES). Since, it requires a manual registration of an email, it is not currently supported by Unicorn.

There are also recommended fields for pony and zebra, even though they are not pony/zebra-specific (unicorn also supports these features).

5.3.2 Recommended fields for pony and zebra

- `"public_postrun_json" : true` : it is recommended to set this flag to `true` so that the postrun json files are open to public (and can be accessed through Web Browser). Both public and private postrun json files can be accessed through CLI command `tibanna log -j jobid -p` given the right AWS credentials are set up.
- `"key_name": "4dn-encoded"` : for security reasons, it is recommended to use the `4dn-encoded` key rather than just passwords for sshing to an AWSEM instance.

5.4 Input file handling

5.4.1 Dimension

An input file may have dimension 0~3 (single element, a 1D array, a 2D array, or a 3D array).

5.4.2 Extra files

An input file may have extra files. Extra files are equivalent to secondary files in CWL, and usually includes index files (e.g. `px2`, `idx`, `tbi`, `bai`, `fai`, `...`). If there are multiple extra files, they should have different formats (extensions). The workflow objects and Tibanna input jsons do not have to specify any extra files and all the extra files associated with a specified input file's File object is automatically transferred along with the file itself to the AWSEM instance.

However, it is required that the input file's File object does contain a corresponding extra file, if CWL requires a secondary file for that input.

5.4.3 Renaming files

The file key on S3 follows the convention `<uuid>/<accession>.<extension>`. Some workflows require some input files to have specific names and to handle this problem, we use the field `rename` in the individual input file dictionary in the input json to specify the target name. When the file is downloaded to the AWSEM instance, before running the workflow, the file will be renamed to this target name. By default, it will be the same as the key on S3.

5.5 Output file handling

There are four types of output - `processed file`, `QC file`, `report file` and `to-be-extra-input file`.

5.5.1 Output processed file handling

Tibanna creates a FileProcessed item for each processed file output in the beginning of the workflow run (through `start_run`) with status `to be uploaded by workflow`. At the end of the run, it patches the `FileProcessed` objects with `status` (`uploaded`), `md5` and `file_size` (through `update_ffmeta`).

If an output processed file has an extra file, likewise the metadata for the extra files will also be created in the beginning of the run, with status `to be uploaded by workflow`. At the end of the run, the extra files will be patched with `status` (`uploaded`), `md5` and `file_size` (through `update_ffmeta`). In order for an output processed file to have an extra file(s), the `secondary_file_formats` must be specified in the `workflow arguments` field for the corresponding output processed file.

5.5.2 Quality metric handling

For QC type output, Tibanna does not create a FileProcessed item but instead creates a QualityMetric item. The quality metric item is created at the *end* of a workflow run, not at the *beginning*, since it is linked from one of the File items (either input or output) involved and if we create a new QualityMetric object in the beginning, it would inevitably replace the existing one, and if the run failed, the new one would remain linked despite the fact that the run failed.

Format of QC output

An example QC type output is the output of a `fastqc` run or a `pairsqc` run, which is a zipped file containing an html file, some text files and image files to be used by the html. However, a regular, non-QC workflow may also create a QC-type output. For example, each of the first few steps of the CGAP upstream pipeline creates a bam file along with a simple QC called `bam-check` which simply checks that the bam file has a header and is not truncated. These workflows have two (or more, in case there are additional output) output files, one `Out processed file` which is the `bam` file and one `Output QC file` which is the `bam-check` report. This `bam-check` report does not have any html file and is not zipped. It's a single text file, which is parsed to create a `QualityMetricBamcheck` object.

To allow flexibility in the format of QC type output, certain qc flags are specified in the `Workflow` object (*not* in the tibanna input json), in the `arguments` field. There may be multiple QC type output files for a single workflow run, and for each, the following must be specified

- `"qc_zipped": true|false` : the output file is zipped
- `"qc_html": true|false` : the output file is an html file
- `"qc_json": true|false` : the output file is a json file
- `"qc_table": true|false` : the output file is a table file (tab-delimited text file)
- `"qc_zipped_html": <name_of_html_file>` : the name of the html file in case the output zipped file contains an html file
- `"qc_zipped_tables": <array_of_name(or_suffix)_of_table_files>` : the name of the table files in case the output zipped file contains table files.
- `"qc_type": <name_of_quality_metric_type>` : name of the `QualityMetric` item type (e.g. `quality_metric_fastqc`, `quality_metric_bamcheck`). This field can be skipped which means that no `QualityMetric` item will be created even though the other QC processings (e.g. unzipping the

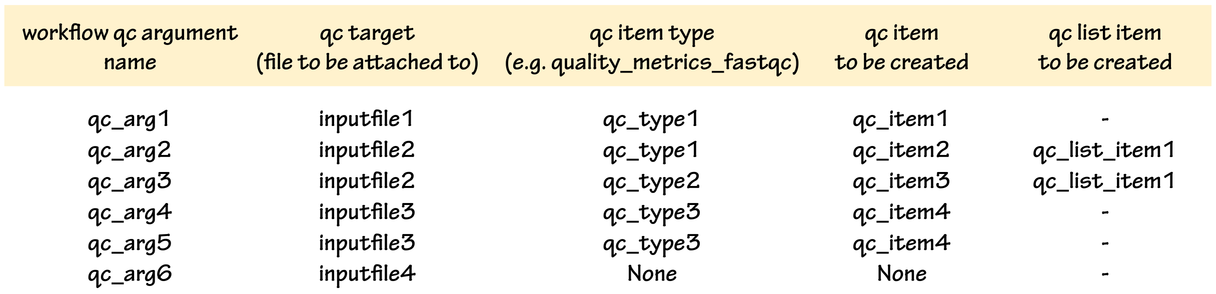
contents, moving the file to a specific location and creating an html, etc) may still happen. This None option was added originally to be able to handle bamsnap output files as QC files without generating a `QualityMetric` item. However, we ended up moving the bamsnap handling to EC2 since it frequently hit lambda memory and runtime limit while unzipping the output (see `qc_unzip_from_ec2`)

- `"argument_to_be_attached_to": <argument>` : the workflow argument name of the file (either input or output) from which the `QualityMetric` object should be linked. (e.g. if the QualityMetric object will be link to the processed bam file whose argument name is `raw_bam`, this field can be set to `raw_bam`.) This is a required parameter.
- `"qc_unzip_from_ec2": true|false` : whether the output zip file should be unzipped to s3 directly from ec2 (default false). This is relevant only if the qc output is zipped and we want the contents of the zip file to be extracted to a folder in S3.

As you can see above, a text-style QC output can either be a JSON or a TSV format. The main difference is that if the output is a TSV format, the corresponding fields must exist and be specified in the schema of the QualityMetric item. A JSON-format output goes directly to the QualityMetric item, and to allow this, the schema must have `additional_properties` to be set `true`.

Behavior of Tibanna-ff given different QC parameters

The following table is an example scenario where a workflow generates six different QC outputs, which is highly unlikely to happen in the real world, but it is introduced to illustrate the behaviors.



workflow qc argument name	qc target (file to be attached to)	qc item type (e.g. quality_metrics_fastqc)	qc item to be created	qc list item to be created
qc_arg1	inputfile1	qc_type1	qc_item1	-
qc_arg2	inputfile2	qc_type1	qc_item2	qc_list_item1
qc_arg3	inputfile2	qc_type2	qc_item3	qc_list_item1
qc_arg4	inputfile3	qc_type3	qc_item4	-
qc_arg5	inputfile3	qc_type3	qc_item4	-
qc_arg6	inputfile4	None	None	-

The first column is the argument name which is unique to each of the six QC outputs. Each of them could have a different QC parameter combination. The second column is the `argument_to_be_attached_to`, which is a required parameter. The third column is the `QualityMetric` item type for each QC output. The fourth column describes the `QualityMetric` itemms to be created and the fifth column describes the `QualityMetricQclist` item to be created to accommodate multiple `QualityMetric` itemms created for a given `argument_to_be_attached_to`. A `QualityMetricQclist` item is not created if there is only one `QualityMetric` item linked from the file metadata of the `argument_to_be_attached_to`, unless the same file metadata already has a `QualityMetric` item from a different workflow run.

The first row is a typical case scenario in which one QC argument to be attached to one input file and so one `QualityMetric` item will be created for this one.

workflow qc argument	qc target	qc item type	qc item	qc list item
name	(file to be attached to)	(e.g. quality_metrics_fastqc)	to be created	to be created
qc_arg1	inputfile1	qc_type1	qc_item1	-
qc_arg2	inputfile2	qc_type1	qc_item2	qc_list_item1
qc_arg3	inputfile2	qc_type2	qc_item3	qc_list_item1
qc_arg4	inputfile3	qc_type3	qc_item4	-
qc_arg5	inputfile3	qc_type3	qc_item4	-
qc_arg6	inputfile4	None	None	-

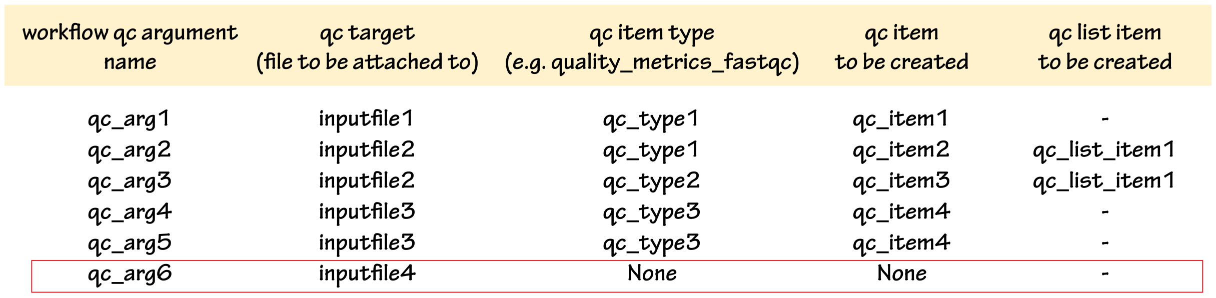
The second and third rows show two different QC arguments to be attached to the same file and each has its own `qc_type`. In this case, a `QualityMetric` item will be created for each QC argument and a `QualityMetricQclist` item will also be created to link two `QualityMetric` items to a single file item. An example could be a BAM file linked to both `bamcheck` and `bamqc` outputs, the former about the sanity check of the output file integrity and the latter about the statistics on the contents of the file including the number of reads, coverages, etc.

workflow qc argument	qc target	qc item type	qc item	qc list item
name	(file to be attached to)	(e.g. quality_metrics_fastqc)	to be created	to be created
qc_arg1	inputfile1	qc_type1	qc_item1	-
qc_arg2	inputfile2	qc_type1	qc_item2	qc_list_item1
qc_arg3	inputfile2	qc_type2	qc_item3	qc_list_item1
qc_arg4	inputfile3	qc_type3	qc_item4	-
qc_arg5	inputfile3	qc_type3	qc_item4	-
qc_arg6	inputfile4	None	None	-

The fourth and fifth rows show a similar case except that two different QC arguments have the same `qc_type`. Since a file item cannot have multiple `QualityMetric` items of the same type, this means only one `QualityMetric` item will be created and the two QC arguments will be merged into this single `QualityMetric` item. Since only one `QualityMetric` item is generated, no `QualityMetricQclist` item is created. This case may not be intuitive, but an example is a workflow that generates two separate QC output files, one in JSON format and the other in the HTML format. In such a case, we want to create a single `QualityMetric` item with the fields taken from the JSON output and a link to the HTML report.

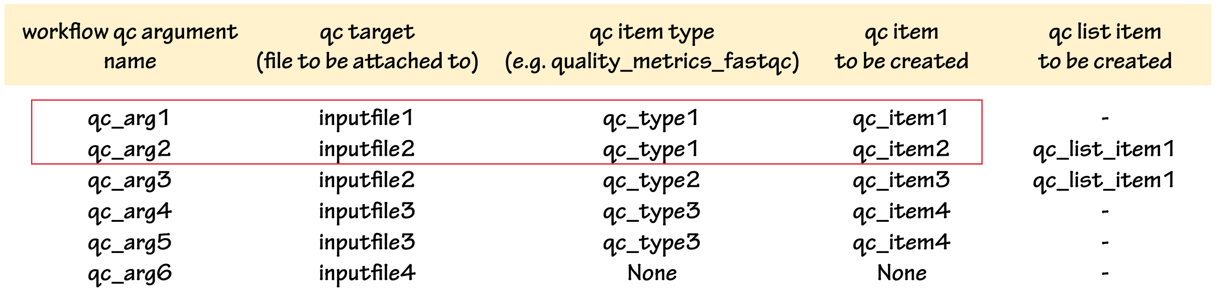
workflow qc argument	qc target	qc item type	qc item	qc list item
name	(file to be attached to)	(e.g. quality_metrics_fastqc)	to be created	to be created
qc_arg1	inputfile1	qc_type1	qc_item1	-
qc_arg2	inputfile2	qc_type1	qc_item2	qc_list_item1
qc_arg3	inputfile2	qc_type2	qc_item3	qc_list_item1
qc_arg4	inputfile3	qc_type3	qc_item4	-
qc_arg5	inputfile3	qc_type3	qc_item4	-
qc_arg6	inputfile4	None	None	-

The sixth row shows a case where `qc_type` is not set (set to `None`). In this case, no `QualityMetric` item is generated. It is used for hidden QC files that do not have a `QualityMetric` item associated with it, such as the output for `bamsnap`.



workflow qc argument name	qc target (file to be attached to)	qc item type (e.g. quality_metrics_fastqc)	qc item to be created	qc list item to be created
qc_arg1	inputfile1	qc_type1	qc_item1	-
qc_arg2	inputfile2	qc_type1	qc_item2	qc_list_item1
qc_arg3	inputfile2	qc_type2	qc_item3	qc_list_item1
qc_arg4	inputfile3	qc_type3	qc_item4	-
qc_arg5	inputfile3	qc_type3	qc_item4	-
qc_arg6	inputfile4	None	None	-

Note that QC arguments with the same `qc_type` do not lead to a merge if their `argument_to_be_attached_to` is different, since each of the file items will have their own `QualityMetric` item in this case.



workflow qc argument name	qc target (file to be attached to)	qc item type (e.g. quality_metrics_fastqc)	qc item to be created	qc list item to be created
qc_arg1	inputfile1	qc_type1	qc_item1	-
qc_arg2	inputfile2	qc_type1	qc_item2	qc_list_item1
qc_arg3	inputfile2	qc_type2	qc_item3	qc_list_item1
qc_arg4	inputfile3	qc_type3	qc_item4	-
qc_arg5	inputfile3	qc_type3	qc_item4	-
qc_arg6	inputfile4	None	None	-

Multiple QC metrics

A single workflow run may produce multiple QualityMetric objects and Tibanna Pony/Zebra supports it.

On the CGAP portal, a single File item may have multiple QualityMetric objects, but only through QualityMetricQclist. A File item cannot directly link to multiple QualityMetric objects, since the field `quality_metric` in a File object is not a list. A QualityMetricQclist object has a field `qc_list` which is a list of links to other QualityMetric objects. 4DN portal currently does not support QualityMetricQclist.

When there are multiple QC-type output, Tibanna Pony/Zebra will segregate the QC output files by `argument_to_be_attached_to`. Let's say there are three QC output files and two of them are associated with `out_bam` and the third one is associated with `out_bw`. The first two will be merged into a single `QualityMetric` object, and the third one will be its own `QualityMetric` object, i.e. Tibanna will create two `QualityMetric` objects even though there are three QC output files, because there are only two distinct groups based on `argument_to_be_attached_to`. The first two QC output files must have the same `qc_type`, but may be a different format - e.g. one of them is html and the other one is JSON. A `File` item is never associated with more than two `QualityMetric` objects of the same type.

Tibanna checks if the `File` item to associate a new `QualityMetric` object already has any `QualityMetric` associated with it. If it does, Tibanna does the following.

1. If the existing `QualityMetric` object is the same type as the new `QualityMetric` object, replace the old one with the new one.
2. If the existing `QualityMetric` object is of a different type from the new `QualityMetric` object, create a new `QualityMetricQclist` object and link it to the corresponding `File` object, move the old `QualityMetric` object to the `QualityMetricQclist` object, and add a new `QualityMetric` object to the `QualityMetricQclist` object.
3. If the existing `QualityMetric` object is of type `QualityMetricQclist`, check the types of `QualityMetric` objects inthe `QualityMetricQclist` object, and if there exists a `QualityMetric`

object of the same type as the new QualityMetric object, replace this one with the new one. If not, add the new QualityMetric object to the existing QualityMetricQclist object.

In theory, a single workflow run could create multiple QualityMetric types by creating a new QualityMetricQclist and adding all of the QualityMetric objects to it, but currently Tibanna does not support it. It may be implemented if we have a case where multiple types of QC is generated by a single workflow for a single file item.

5.5.3 Report-type output handling

A report-type output is different from a QC-type output in that no QualityMetric object is created out of it. A good example of a report-type output is md5 which calculates the md5sum of an input file and the result report output file that contains the md5sum value is parsed and the value is patched to the md5sum (and content_md5sum if the file is compressed) of the input File item.

5.5.4 Handling output that becomes an extra file of an input file

An example of an Output to-be-extra-input file is the output of workflow bed2beddb where the output beddb file will be attached as an extra_file of the input bed file, instead of creating a separate processed file with the beddb format.

By default, a second run of the same workflow run fails to start, to avoid overwriting the output extra file without any metadata log, unless "overwrite_input_extra": true is set in the config of the input json.

The extra file in the input File metadata is created at the beginning of the run (through start_run) with status to be uploaded by workflow and the AWSEM instance will upload the output file to the right bucket with the right key including the right extension (the extension of the extra file). If this upload fails, check_task will throw and AWSEM error. The last step update_ffmeta will make sure that the key with the right extension exists in the right bucket, but it does *not* check that the file is new or not. If it does, it will update the status of the extra file to uploaded.

5.6 Custom fields

In case we want to pass one custom fields to WorkflowRun, FileProcessed or QualityMetric objects that are created by a workflow run, we can do that by adding custom fields to the input json. Common examples of custom field would be lab and award for pony and project and institution for zebra. One could also set genome_assembly to be passed to a FileProcessed object.

5.6.1 Custom fields for workflow run

The wfr_meta field specifies custom fields to be passed to a WorkflowRun object.

```
"wfr_meta": { "key1": "value1", "key2": "value2" ,,, }
```

In the above example, the WorkflowRun object will have field key1 with value value1 and field key2 with value value2.

5.6.2 Custom fields for processed files

The custom_pf_fields field specifies custom fields to be passed to a FileProcessed object. This field has one additional level to specify whether the field should apply to all processed files (ALL) or a specific processed file (the argument name of the specific processed file).

```
"custom_pf_fields": {
    "ALL": { "key1": "value1", "key2": "value2" },
    "out_bam": {"key3": "value3" }
}
```

In the above example, if we have two output files with argument names out_bam and out_bw, the processed file(s) associated with both out_bam and out_bw will have field key1 with value value1 and field key2 with value value2, but only the processed file(s) associated with out_bam will have field key3 with value value3.

5.6.3 Custom fields for quality metrics

The custom_qc_fields field specifies custom fields to be passed to a FileProcessed object, and all the QualityMetric objects generated (including QualityMetricWorkflowrun) will have the fields specified by custom_qc_fields.

```
"custom_qc_fields": { "key1": "value1", "key2": "value2" ,,, }
```

In the above example, all the QualityMetric objects will have field key1 with value value1 and field key2 with value value2.

CHAPTER 6

Job Description JSON Schema

The Job Description json for Tibanna Pony and Zebra are different from the json for Tibanna, but it's the same in that it defines an individual execution. The `config` part is largely the same. The Pony/Zebra input json does not have `args` but has its own set of fields.

The first step of the Pony/Zebra step function converts this input json to a Unicorn input json and pass it to the second step (`run_task`).

6.1 Example job description

```
{
    "description": [
       "This input json run a bwa-mem workflow, which is part of 4DN Hi-C pipeline",
       "on hg38 genome reference."
    ],
    "app_name": "bwa-mem",
    "_tibanna": {
      "env": "fourfront-webdev",
      "run_type": "bwa-mem"
    },
    "output_bucket": "elasticbeanstalk-fourfront-webdev-wfoutput",
    "workflow_uuid": "0fbe4db8-0b5f-448e-8b58-3f8c84baabf5",
    "parameters" :  {"nThreads": 4},
    "input_files" : [
       {
           "object_key": "4DNFIZQZ39L9.bwaIndex.tgz",
           "workflow_argument_name": "bwa_index",
           "uuid": "1f53df95-4cf3-41cc-971d-81bb16c486dd",
           "bucket_name": "elasticbeanstalk-fourfront-webdev-files",
           "rename": "hg38.tar.gz"
       },
       {
           "workflow_argument_name": "fastq1",
```

(continues on next page)

(continued from previous page)

```
            "bucket_name": "elasticbeanstalk-fourfront-webdev-files",
            "uuid": "1150b428-272b-4a0c-b3e6-4b405c148f7c",
            "object_key": "4DNFIVOZN511.fastq.gz"
        },
        {
            "workflow_argument_name": "fastq2",
            "bucket_name": "elasticbeanstalk-fourfront-webdev-files",
            "uuid": "f4864029-a8ad-4bb8-93e7-5108f462ccaa",
            "object_key": "4DNFIRSRJH45.fastq.gz"
        }
  ],
  "config": {
    "instance_type": "t3.large",
    "EBS_optimized": true,
    "ebs_size": 30,
    "ebs_type": "gp2",
    "shutdown_min": 30,
    "password": "",
    "log_bucket": "tibanna-output",
    "key_name": "4dn-encoded",
    "spot_instance": true,
    "spot_duration": 360,
    "behavior_on_capacity_limit": "wait_and_retry",
    "overwrite_input_extra": false,
    "cloudwatch_dashboard", false,
    "email": true,
    "public_postrun_json" : true
  },
  "custom_pf_fields": {
    "out_bam": {
        "genome_assembly": "GRCh38"
    }
  },
  "wfr_meta": {
    "notes": "a nice workflow run"
  },
  "custom_qc_fields": {
    "award": "/awards/5UM1HL128773-04/",
    "lab": "/labs/bing-ren-lab/"
  },
  "push_error_to_end": true
  "dependency": {
    "exec_arn": [
        "arn:aws:states:us-east-1:643366669028:execution:tibanna_unicorn_default_
↪7412:md5_test",
        "arn:aws:states:us-east-1:643366669028:execution:tibanna_unicorn_default_
↪7412:md5_test2"
    ]
  }
}
```

- The `description` field is an optional field for humans and they are ignored by Tibanna.
- The `app_name` field contains the name of the workflow.
- The `output_bucket` field specifies the bucket where all the output files go to. It is not required if `_tibanna` specifies `env`. The bucket name is auto-determined from the `env` value.

- The `workflow_uuid` field contains the uuid of the 4DN workflow metadata.
- The `parameters` field contains a set of workflow-specific parameters in a dictionary.
- The `additional_benchmarking_parameters` field contains a set of additional parameters that are not required for workflow runs but is required for a benchmarking function (e.g. resource usage depends on number of reads which is not a parameter for workflow run)
- The `input_files` field specifies the argument names (matching the names in CWL), the input file metadata uuid and its bucket and object key name.
  - `workflow_argument_name` and `uuid` are required fields.
  - `bucket_name` and `object_key` are required only if the content is a list.
  - `rename` (optional) can be used to rename a file upon download from s3 to an instance where the workflow will be executed.
- The `config` field is directly passed on to the second step, where instance_type, ebs_size, EBS_optimized are auto-filled, if not given.
  - The `spot_instance` field (optional), if set `true`, requests a spot instance instead of an on-demand instance.
  - The `spot_duration` field (optional), if set, requests a fixed-duration spot instance instead of a regular spot instance. The value is the duration in minutes. This field has no effect if `spot_instance` is either `false` or not set.
  - The `behavior_on_capacity_limit` field (optional) sets the behavior of Tibanna in case AWS instance Limit or Spot instance capacity limit is encountered. Default value is `fail`. If set to `wait_and_retry`, Tibanna will wait until the instance becomes available and rerun (10 min interval, for 1 week). If `spot_instance` is `true` and `behavior_on_capacity_limit` is set to `retry_without_spot`, when the spot instance is not available, it will automatically switch to a regular instance of the same type (applicable only when `spot_instance` is `true`).
  - The `overwrite_input_extra` (optional) allows overwriting on an existing extra file, if the workflow hasan output of type `Output to-be-extra-input file` (i.e., creating an extra file of an input rather than creating a new processed file object). Default `false`.
  - The `cloudwatch_dashboard` field (optional), if set `true`, creates a cloudwatch dashboard for the job, which allows users to trace memory, disk and CPU utilization during and after the run.
  - The `email` field (optional), if set `true`, sends a notification email to `4dndcic@gmail.com` when a workflow run finishes.
  - The `public_postrun_json` field (optional) is recommended to be set `true`. This way the postrun json files become publicly available when they're created.
  - The `key_name` field is recommended to be set `4dn-encoded` which is the key used by the 4DN DCIC team.
- The `push_error_to_end` field (optional), if set true, passes any error to the last step so that the metadata can be updated with proper error status. (default true)
- The `custom_pf_fields` field (optional) contains a dictionary that can be directly passed to the processed file metadata. The key may be either `ALL` (applies to all processed files) or the argument name for a specific processed file (or both).
- The `wfr_meta` field (optional) contains a dictionary that can be directly passed to the workflow run metadata.
- The `custom_qc_fields` field (optional) contains a dictionary that can be directly passed to an associated Quality Metric object.

- The `dependency` field (optional) sets dependent jobs. The job will not start until the dependencies successfully finish. If dependency fails, the current job will also fail. The `exec_arn` is the list of step function execution arns. The job will wait at the run_task step, not at the start_task step (for consistenty with unicorn). This field will be passed to run_task as `dependency` inside the `args` field.

CHAPTER 7

Permissions

Permissions for Tibanna Pony/Zebra users are set up in a way similar to the way it is set up for Unicorn, but there are differences.

7.1 With usergroup

One can set up usergroups while deploying a pony/zebra like the way a usergroup is set up while deploying a unicorn (using `-g` option). They work the same way. Bucket permission doesn't have to be specified. By default, all the buckets listed in the `IAM_BUCKETS` variables defined in `tibanna_4dn/vars.py` or `tibanna_cgap/vars.py` are added to the user group. These buckets are those used for 4DN and CGAP **test environments**, respectively. Tibanna deployed with usergroup does not have permission to production buckets for 4DN and CGAP, unless explicitly specified using the `-b` option. It is recommended to use the `without usergroup` option to access production buckets.

```
tibanna_4dn deploy_pony -g <usergroupname>

# e.g.
tibanna_4dn deploy_pony -g default_luisa
```

```
tibanna_cgap deploy_zebra -g <usergroupname>
```

7.2 Without usergroup

If a pony/zebra is deployed without usergroup, it gives the pony/zebra access to all the buckets. You have to be an admin in order to use this kind of pony/zebra. It applies to cases where you deploy a pony/zebra without any other options, or with a suffix (`-s` option).

```
tibanna_4dn deploy_pony
```

```
tibanna_4dn deploy_pony -s dev
```

```
tibanna_cgap deploy_zebra
```

```
tibanna_cgap deploy_zebra -s dev
```

CHAPTER 8

Tests

Whenever changes are made, we need to run tests.

8.1 Prerequisites

To run tests, first do

```
pip install -r reqirements-test.txt
```

Also, before running any tests, make sure to first build based on the current repo content.

```
python setup.py install
```

8.2 Test portals

Currently `fourfront-webdev` is used as a main Tibanna test portal for 4DN, and `fourfront-cgap` is for CGAP.

8.3 Local tests

Local tests run test scripts in the `tests` directory using `pytest`.

Local tests do not involve spinning up any EC2 instance, but some of them involve uploading small files to S3 and posting minimal objects to test portals. After each such test, the files and posted items are deleted.

Local tests are a good starting point, but they're not comprehensive, because it doesn't involve any real EC2 instance, and the code is not run on an actual AWS Lambda environment. So this way, we cannot catch any Lambda deployment error or IAM permission setup error.

Running all local tests can be done by `invoke test`. Some `flake8` tests are going to fail that we're not going to fix, so it is recommended to run this test without `flake8`.

```
invoke test --no-flake
```

To run your code through `flake8`, it would be nice to do it script by script after modification, e.g.

```
flake8 tibanna_4dn/check_task.py
```

To run individual test (to save time for testing while still working on the code), one could directly use `pytest`, e.g.

```
pytest tests/tibanna/pony/test_fourfrontupdater.py
```

8.4 Post-deployment test

Testing on a dev tibanna requires spinning up EC2 instances and it costs $$. So this test should be done only when we're confident about our modifications and after all the local tests already passed.

A post-deployment test first deploys dev tibanna pony and zebra and then tests are submitted to the newly deployed dev Tibanna step functions.

The dev tibanna suffixes are currently specified in `tasks.py` as `pre`. (The post-deployment tests run on `tibanna_pony_pre` and `tibanna_zebra_pre`).

To run the full post-deployment tests,

```
invoke test --deployment
```

8.4.1 Four different output types

Pony

- `md5 (report)`, `fastqc (QC)`, `bwa-mem (processed)` (with a small reference index file), `bedGraphToBigWig (to-be-extra-input`, not yet included)
- `QCList` is not available for Fourfront.

Zebra

- `md5 (report)`, `fastqc (QC)`, `bwa-check (processed)` (with a small reference index file)
  - We don't have to-be-extra-input type on CGAP yet.
- `QCList` test by running `bamqc` on top of `bwa-check` and then rerunning `bamqc` (must replace the first one). (not yet included)

8.4.2 Input array types

Pony

- `merge_fastq` with very small fastq files (1D array) (not yet included)
- `merge_and_cut` test workflow item (3D array) (not yet included)

Zebra

- `merge_bam` with very small bam files (1D array) (not yet included)
- `merge_and_cut` test workflow item (3D array) (not yet included)

8.4.3 Reruns

Pony

- md5 conflict test (not yet included)
  - rerun the same File item with a different md5 content (must fail)
  - rerun a different File item with the same md5 content (must fail)
- overwrite_extra test (not yet included)
  - rerun the same `bedGraphToBigWig` job with different file content with overwrite_extra = True (must overwrite)
  - rerun the same `bedGraphToBigWig` job with overwrite_extra = False (must fail)

Zebra

- md5 conflict test
  - rerun the same File item with a different md5 content (must fail) (not yet included)
  - rerun a different File item with the same md5 content (must fail) (not yet included)

8.4.4 WDL

Pony

- `merge` WDL test workflow item (also 2D array test) (not yet included)

Zebra

- `merge` WDL test workflow item (also 2D array test) (not yet included)

8.4.5 Workflow Run QC

- check html & tsv (not yet included)

8.4.6 EC2 test

- EC2 unintended termination test (force kill externally)
- EC2 idle test (sleep for 1hr) (Not yet included)

8.5 Travis test

Travis test is currently set up to run at every push and every PR. Travis test currently runs only local tests for most cases. It runs post-deployment tests when there is a `git push` to the `production` branch. This can take longer and $$ (actually launching EC2 instances) and we should do this only when we're fairly confident, usually after we merge things to the `master` branch, we can push it to `production`. After the post-deployment test succeeds, Travis auto-deploys production pony and zebra.

8.6 Other tests that we should include in the future

The following tests are currently not set up and is done manually. Ideally they should be automated in the future.

- CLI test
- md5/fastqc trigger test
- initiator test
- permission tests

CHAPTER 9

Production Deployment

After all the tests pass, we should deploy production tibanna as below.

9.1 Pony

```
tibanna_4dn deploy_pony
tibanna_4dn deploy_pony -s tmp_md5  # md5/fastqc triggers
tibanna_4dn deploy_pony -g default_luisa  # luisa's tibanna, with different permission
```

9.2 Zebra

```
tibanna_cgap deploy_zebra
tibanna_cgap deploy_zebra -s tmp_md5  # md5/fastqc triggers
```