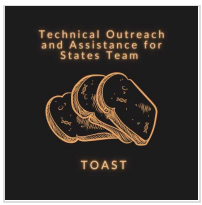

1

Oct 18, 2021

Titan-gc SARS-CoV-2 Strain Characterization Workflow for Bioconda V.1

protocol

1

Michelle Su[1], Robert A Petit III[2],
Technical Outreach and Assistance for States Team[1]

[1]Centers for Disease Control and Prevention; [2]Wyoming Public Health Laboratory

1

protocol .

Michael Weigand
Centers for Disease Control and Prevention

This protocol covers the process of using Titan on the command-line, which was developed by Robert Petit at Wyoming Public Heath Laboratories. Some laboratories have the computing infrastructure and capacity to conduct local bioinformatic analyses and wish to use the Titan workflow, but not on the Terra platform. Local implementation of Titan-gc eliminates bottlenecks such as data upload to the cloud as well as offers users the ability to incorporate the process into scripts if they desire. Provided steps include how to install dependencies via Conda, create input files for the workflow, and run the pipeline directly or using a Nextflow wrapper. Basic troubleshooting tips are also discussed.

For technical assistance, please contact: **TOAST@cdc.gov**

Michelle Su, Robert A Petit III, Technical Outreach and Assistance for States Team 2021. Titan-gc SARS-CoV-2 Strain Characterization Workflow for Bioconda. **protocols.io** https://protocols.io/view/titan-gc-sars-cov-2-strain-characterization-workfl-byu9pwz6

(ILLUMINA MENU) Protocols for SARS-CoV-2 Library Prep with ARTIC Primers, Bioinformatic Analysis, and Database Submission

SARS-CoV-2, sequencing, conda, bioconda, COVID-19

protocol ,

Oct 07, 2021

Oct 18, 2021

53889

Part of collection
(ILLUMINA MENU) Protocols for SARS-CoV-2 Library Prep with ARTIC Primers, Bioinformatic Analysis, and Database Submission

:

Set up conda environment

1 The Titan workflow and its dependencies can be installed using the Conda package manager and the Bioconda channel for bioinformatics software. **To install Conda**, follow the instructions in Step 1.1. **To update an existing Conda installation**, go to Step 1.2. **To add**

the Bioconda channel, go to Step 1.3.

A Bioconda recipe for titan-gc is available at: https://bioconda.github.io/recipes/titan-gc/README.html

With Conda running and an activated Bioconda channel on your machine, create a new Conda environment:

```
conda create -n titan-cli -c conda-forge -c bioconda titan-gc
```

This creates a new environment named **titan-cli** in which to install the titan-gc software. This Conda environment can be activated and deactivated as needed. When active, the environment name appears in parentheses to the left of the prompt. When deactivated, the normal prompt returns. For example:

```
[me@PC ~]$ conda activate titan-cli
(titan-cli) [me@PC ~]$ conda deactivate
[me@PC~]$
```

With the **titan-cli** environment activated, install and update the titan-gc software:

1.1 Miniconda is a lightweight implementation that includes Conda and Python.
Detailed installation instructions can be found at:
https://conda.io/projects/conda/en/latest/user-guide/install/index.html.
A Conda tutorial can be found at:
https://jvhagey.github.io/Tutorials/mydoc_Installation.html.

Download the Miniconda3 installer appropriate for your operating system:
https://docs.conda.io/en/latest/miniconda.html

Miniconda3 is implemented in Python 3 and the default versions of Python 3 installed in new environments depends on the installer. However, both Python 2.x and Python 3.x environments can still be installed with explicit parameter settings.

Verify the installer using SHA256, a cryptographic hash, to ensure the installer has not been tampered with or corrupted. Detailed instructions can be found at: https://conda.io/projects/conda/en/latest/user-guide/install/download.html#hash-verification

Windows (using PowerShell V4 or higher):

```
Get-FileHash filename -Algorithm SHA256
```

MacOS:

```
shasum -a 256 filename
```

Linux:

```
sha256sum filename
```

Example command and output on linux:

```
[me@PC ~]$ sha256sum Miniconda3-py39_4.9.2-Linux-x8
536817d1b14cb1ada88900f5be51ce0a5e042bae178b55
Miniconda3-py39_4.9.2-Linux-x86_64.sh
```

Hash key for Miniconda installer on conda website

If the hashes match, proceed with installation.

Install.

Windows:
After download, double-click the .exe file and follow the prompts.

MacOS/Linux:
1. Open a terminal and run the installer script with the correct filename:

```
bash Miniconda-installer.sh
```

2. Restart the terminal for the changes to take effect.

3. Test the installation with command:

```
conda list
```

1.2 Conda itself can be updated with command:

```
conda update conda
```

1.3 Add the Bioconda channel, and required dependencies **in the specified order**, to an existing Conda installation with commands:

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

Create the input file

2 Analyzing sequences with the titan-gc workflow requires an input file defining:

1. Same name
2. Workflow
3. Sequencing read file locations
4. Primer bed file location

ARTIC primer bed files are available at: https://github.com/artic-network/primer-schemes/tree/master/nCoV-2019

The input file is created using the included **titan-gc-prepare.py** script in the Conda environment:

```
(titan-cli) [me@PC ~]$ titan-gc-prepare.py
usage: titan-gc-prepare [-h] [-f STR] [--fastq_separator STR]
                        [--fastq_pattern STR] [--pe1_pattern STR]
                        [--pe2_pattern STR] [-r] [--prefix STR] [--tsv]
                        [--pangolin_docker STR] [--params STR]
                        FASTQ_PATH WORKFLOW PRIMER

titan-gc-prepare - Read a directory and prepare a JSON for input to
Titan GC

optional arguments:
  -h, --help            show this help message and exit

Titan-GC Prepare Parameters:
  FASTQ_PATH            Directory where FASTQ files are stored
  WORKFLOW              The Titan-GC workflow to use for analysis.
Options:
                        clearlabs, illumina_pe, illumina_se, ont
  PRIMER                A file containing primers (bed format) used during
                        sequencing.
  -f STR, --fastq_ext STR
                        Extension of the FASTQs. Default: .fastq.gz
```

```
  --fastq_separator STR
                  Split FASTQ name on the last occurrence of the
                  separator. Default: _
  --fastq_pattern STR   Glob pattern to match FASTQs. Default:
*.fastq.gz
  --pe1_pattern STR     Designates difference first set of paired-end
reads.
                  Default: ([Aa]|[Rr]1|1) (R1, r1, 1, A, a)
  --pe2_pattern STR     Designates difference second set of paired-end
reads.
                  Default: ([Bb]|[Rr]2|2) (R2, r2, 2, AB b)
  -r, --recursive       Directories will be traversed recursively
  --prefix STR          Replace the absolute path with a given string.
                  Default: Use absolute path
  --tsv                 Output FOFN as a TSV (Default JSON)

Optional Titan-GC Workflow Parameters:
  --pangolin_docker STR
                  Docker image used to run Pangolin (takes priority over
                  --params)
  --params STR          A JSON file containing parameter values
```

The script takes three required positional inputs (**in this specific order**): fastq path, workflow, and primer bed file. The script infers sample name from the sequencing read fastq filenames.

FASTQ_PATH: relative or absolute path to directory containing fastq files
Optional parameters to modify how the script looks for and processes fastq files:
-f, --fastq_ext: .fastq.gz file extension is removed to infer sample name
--fastq_separator: underscore is used to infer where the sample name is. i.e. _1 or _2 is removed
--fastq_pattern: .fastq.gz read files are looked for. If you want to change it to look for fastqs, you would put '*.fastq'
(do not forget the single quotes or the wildcard will expand and substitute the files names)
--pe[1/2]_pattern: specifies how to differentiate between left and right reads. Accepts R1/2, r1/2, 1/2, A/B, a/b
-r, --recursive: look not only in current folder for reads files but all folders in this folder

WORKFLOW: What workflow to run
Options: clearlabs, illumina_pe, illumina_se, ont

PRIMER: relative or absolute paths accepted for location of primer bed file

Optional parameters:

--pangolin_docker: specify docker tag for version of pangolin you want to run; **not compatible with --tsv**

--params: relative or absolute paths accepted for location of params.json; **not compatible with --tsv**

example params.json: default params

--tsv: this specifies the output file format. Default is the JSON file format. JSON is used to run the titan-gc directly, and TSV is used for Nextflow

For more details about the **--tsv** option, go to **Step 3** to learn more about running titan-gc directly or with the Nextflow workflow wrapper.

Note: samples prepared together will be run with the same workflow. If you need to run different workflows, prepare separate input files for each.

Example command:

```
titan-gc-prepare.py ./reads_dir illumina_pe ./primer.bed > input.file
```

Example JSON input file creation:

Note that the "." (period) in the first line of the example below indicates that the current directory should be used as FASTQ_PATH.

```
(titan-cli) [me@PC test_seq]$ titan-gc-prepare.py . illumina_pe
/$PATH/arctic_V3_nCov-2019.bed > input.json
(titan-cli) [me@PC test_seq]$ cat input.json
{
  "titan_gc.samples": [
    {
      "sample": "SRR13422799",
      "titan_wf": "illumina_pe",
      "r1": "/$PATH/SRR13422799_1.fastq.gz",
      "r2": "/$PATH/SRR13422799_2.fastq.gz",
      "primers": "/$PATH/arctic_V3_nCov-2019.bed"
    }
  ]
}
```

Note: $PATH just a placeholder and will have to be changed to your path to the files.

Example TSV input file/FOFN (File of file names) creation:

```
(titan-cli) [me@PC test_seq]$ titan-gc-prepare.py . illumina_pe
/$PATH/arctic_V3_nCov-2019.bed --tsv > input.txt
(titan-cli) [me@PC test_seq]$ cat input.txt
sample  titan_wf      r1      r2      primers
SRR13422799     illumina_pe
/$PATH/test_seq/SRR13422799_1.fastq.gz
/$PATH/test_seq/SRR13422799_2.fastq.gz      /$PATH/arctic_V3_nCov-
2019.bed
```

$PATH just placeholder and will need to be changed with your path to the files

Running the titan-gc pipeline

3 The titan-gc pipeline can be run two different ways:

1. Directly (**Step 3.1**)
2. With Nextflow (**Step 3.2**)

Differences between the two implementations are primarily related to call-caching (how

samples are submitted to the pipeline), job submission, and STDOUT.

Call-caching: The Nextflow wrapper offers the benefits of call-caching, essentially allowing failed runs to be resumed without rerunning finished analyses. This saves time and compute resources. Running the pipeline directly (Step 3.1) uses cromwell run, and call-caching is only available from Cromwell's server mode, which will retry failed processes 10 times before the workflow is aborted.

Job submission: The Nextflow wrapper individually submits each sample to the Titan workflow. If one sample fails, downstream samples are not affected. Alternatively, when running the pipeline directly (Step 3.1), failed samples can cause the entire workflow to abort before remaining subsequently queued samples can be run to completion.

STDOUT: The Nextflow wrapper outputs little information, but provide a summary message to report the number of samples finished before merging results. Running the pipeline directly can output all of Cromwell's very extensive run logs to STDOUT.

Note: the first time you run the pipeline, it will pull all the containers, a process that can take up to 20-30 minutes.

For **Singularity**, the default cache directory is "~/.singularity/cache".
You can change this with command:
"export SINGULARITY_CACHEDIR=/path/to/newcache"
(this must be added to your "~/.bashrc" file to make the change permanent)

For **Docker**, the default cache directory is "/var/lib/docker". Changing the cache directory for Docker is more complicated, requiring edits to a config file, and will not be discussed in this protocol.

3.1 Running pipeline directly:

```
(titan-cli) [me@PC ~]$ titan-gc --help
usage: titan-gc [-h] [-i STR] [--inputs STR] [-o STR] [--outdir
STR] [--options STR] [--quiet]

titan-gc-cli - Run Titan GC on a set of samples.

required arguments:
  -i STR, --inputs STR  The JSON file to be used with
Cromwell for inputs.
  -o STR, --outdir STR  Output directory to store the final
results in.

optional arguments:
  -h, --help            Show this help message and exit
  --options STR         JSON file containing Cromwell options
  --profile STR         The backend profile to use [options:
docker, singularity]
  --config STR          Custom backend profile to use
  --cromwell_jar STR    Path to cromwell.jar (Default use
conda install)
  --quiet               Silence all STDOUT from Cromwell and
titan-gc-organize
```

The command takes two required inputs: **input JSON file** and **output directory**

-i: relative or absolute path to input JSON file

-o: relative or absolute path to output directory

Optional parameters:

--profile: what container software to use.

options: docker(default), singularity.

--options: provide JSON file to modify cromwell run attributes

More info about config options can be found at https://cromwell.readthedocs.io/en/stable/Configuring/

--quiet: Silence output from running the pipeline

Example command:

```
titan-gc -i input.json -o titan-out --profile singularity
```

The running pipeline should produce extensive text output of Cromwall's run information.

If a sample fails, because Cromwell provides no call-caching, there are no options except to rerun the sample. **If the input.json file included multiple samples some may have successfully completed before the workflow aborted and a new input.json should be prepared for rerunning only the failed samples.**

3.2 Acquire the Nextflow wrapper here: https://gitlab.com/steamboat/titan-nf/-/tree/main.

DISCLAIMER: The titan-nf Nextflow wrapper is not officially supported by the Titan development team and is a separate activity supported by Robert Petit in Wyoming. This workflow is under active development and frequent updates are expected.

First, pull the Nextflow wrapper, which is not included in the Conda installation. Nexflow itself should already be installed by default. This step only needs to be completed once or whenever the Nextflow wrapper is updated. The workflow will be stored at "$HOME/.nextflow/assets/steamboat/titan-nf".

Note: execute while your Conda environment is active unless your system has Nextflow.

```
nextflow pull -hub gitlab -r main steamboat/titan-nf
```

Running the Nextflow wrapper:

```
(titan-cli) [me@PC ~]$ nextflow run -hub gitlab -r main
steamboat/titan-nf --help
N E X T F L O W  ~  version 21.04.0
Launching `steamboat/titan-nf` [berserk_ekeblad] -
revision: e36cccd145 [main]

Required Parameters:
    --samples FILE          File produced by titan-gc-prepare.py
(--tsv used)

Optional Parameters:
    --titan_opts FILE       A JSON file with Titan GC parameters
to use.
                            Default: Use Titan-GC's defaults

    --pangolin_docker STR   The pangolin docker image to use
                            Default: Use Titan-GC's defaults

    --profile STR           Profile for Titan GC to use
                            Options: docker or singularity
                            Default: singularity

    --outdir DIR            Directory to write results to
                            Default: ./

    --run_name STR          Name to use for the results folder
                            Default: titan-gc
```

The command takes one required input: **input TSV file**

--samples: relative or absolute paths taken for input TSV file (created using --tsv option of titan-gc-prepare.py)

Optional parameters:

--titan_opts: relative or absolute paths accepted for location of JSON parameter file

example params.json: default params

--pangolin_docker: specify docker tag for version of pangolin you want to run

--profile: what container software to use.

options: docker, singularity(default).

--outdir: directory in which output folder is created. Default is the current working directory.

--run_name: name of output folder. Default is titan-gc.

Example command:

```
nextflow run -hub gitlab -r main steamboat/titan-nf --samples input.txt
```

Example of running nextflow wrapper:

```
(titan-cli) [me@PC test_seq]$ nextflow run -hub gitlab -r main steamboat/titan-nf --samples input.txt
N E X T F L O W  ~  version 21.04.0
Launching `steamboat/titan-nf` [clever_mercator] - revision: 3af325405e [main]
executor >  local (2)
[6e/1c5eb5] process > titan_gc (SRR13422799) [100%] 1 of 1 ✔
[41/fccd35] process > merge_results         [100%] 1 of 1 ✔
Completed at: 16-Jul-2021 12:33:51
Duration    : 3m 59s
CPU hours   : 0.3
Succeeded   : 2
```

If a sample fails, use the **-resume** option to retry the sample beyond the initial number of Nextflow attempts (default: 5 times). Some failures are not reproducible, and simply rerunning may solve the issue. Completed samples will not be rerun, thanks to call-caching, so the same input file can be reused without excluding the completed sampled.

Read more about Nextflow -resume at:
https://www.nextflow.io/blog/2019/demystifying-nextflow-resume.html

Example of -resume option in Nextflow:

```
(titan-cli) [me@PC test_seq]$ nextflow run -hub gitlab -r
main steamboat/titan-nf --samples input.txt -resume
N E X T F L O W  ~  version 21.04.0
Launching `steamboat/titan-nf` [ecstatic_bhabha] -
revision: 3af325405e [main]
[6e/1c5eb5] process > titan_gc (SRR13422799) [100%] 1 of
1, cached: 1 ✔
[41/fccd35] process > merge_results        [100%] 1 of 1,
cached: 1 ✔
```

Output

4 The output directory will include a structure similar to that shown below. Running the titan-gc pipeline using the Nextflow wrapper produces an additional "nf-info" folder. Read more about the Nextflow outputs at: https://www.nextflow.io/docs/latest/tracing.html

```
titan-out
├── alignments
│   ├── SRR13422799.flagstat.txt
│   ├── SRR13422799.primertrim.sorted.bam
│   ├── SRR13422799.primertrim.sorted.bam.bai
│   └── SRR13422799.stats.txt
├── assemblies
│   └── SRR13422799.ivar.consensus.fasta
├── cromwell
│   ├── SRR13422799-metadata.json
│   ├── SRR13422799-stderr.txt
│   └── SRR13422799-stdout.txt
├── dehosted_reads
│   ├── SRR13422799_R1_dehosted.fastq.gz
│   └── SRR13422799_R2_dehosted.fastq.gz
├── kraken2_reports
│   ├── SRR13422799_dehosted_kraken2_report.txt
│   └── SRR13422799_kraken2_report.txt
├── nextclade
│   ├── SRR13422799.ivar.consensus.nextclade.auspice.json
│   ├── SRR13422799.ivar.consensus.nextclade.json
│   └── SRR13422799.ivar.consensus.nextclade.tsv
├── nf-info
│   ├── titan-report.html
│   ├── titan-report.html.1
│   ├── titan-timeline.html
│   ├── titan-timeline.html.1
│   ├── titan-trace.txt
│   ├── titan-trace.txt.1
│   └── titan-trace.txt.2
├── pangolin_reports
│   └── SRR13422799.pangolin_report.csv
├── results
│   ├── SRR13422799.json
│   └── SRR13422799.tsv
├── titan-results.json
├── titan-results.tsv
└── vadr_alerts
    └── SRR13422799.ivar.consensus.vadr.alt.list
```

Troubleshooting

5 There are two places to investigate when troubleshooting:

1. Cromwell STDOUT: the STDOUT messages can indicate which step failed and where in the working directory to look
2. Cromwell working directory: this holds the STDOUT and STDERR log files for each task run

Cromwell STDOUT:

- Running pipeline directly: cromwell-stdout.txt will be in your specified output directory
- Running Nextflow wrapper: cromwell-stdout.txt will be in the titan-gc folder in the unique hash identifier folders in the work folder (which is by default in the directory you ran the pipeline from), e.g.:

```
$PWD/work/0a/250b0f5bc0014f2192c06d661bd210/titan-gc/cromwell-stdout.txt
```

Cromwell work directory:

- Running pipeline directly: There will be a cromwell-executions folder (which is by default in the directory you ran the pipeline from) which contains a titan-gc folder which contains unique hash identifier folders, e.g.:

```
$PWD/cromwell-executions/titan_gc/3458e2f1-4566-4762-80ee-4f4f02d16c5b/
```

- Running nextflow wrapper: In the nextflow work folder under the unique hash identifier, there will be a cromwell-executions folder which contains a titan-gc folder which contains unique hash identifier folders, e.g.:

```
$PWD/work/0a/250b0f5bc0014f2192c06d661bd210/cromwell-executions/titan_gc/5f303e3d-e2bd-46d0-9541-1006a960f3f8
```